eBPF and Dynamic Tracing for MariaDB DBAs

(ftrace, bcc tools and bpftrace)

Valerii Kravchuk, Principal Support Engineer, MariaDB

valerii.kravchuk@mariadb.com

Who am I?

Valerii (aka **Valeriy**) **Kravchuk**:

- MySQL Support Engineer in MySQL AB, Sun and Oracle, 2005-2012
- Principal Support Engineer in Percona, 2012-2016
- Principal Support Engineer in MariaDB Corporation since March 2016
- **http://mysqlentomologist.blogspot.com** - my blog about MySQL and MariaDB (a lot about MySQL bugs, but some **HowTo**s as well)
- **https://www.facebook.com/valerii.kravchuk** - my Facebook page
- **http://bugs.mysql.com** - my personal playground
- **@mysqlbugs #bugoftheday**
- **MySQL Community Contributor of the Year 2019**
- I like FOSDEM (but this year my talks were not accepted…)

Sources of tracing and profiling info for MariaDB

- Trace files from **-debug** binaries, optimizer trace files
- Extended slow query log
- **show** [**global**] **status**;
- **show engine innodb status**\G
- **show engine innodb mutex**;
- InnoDB-related tables in the INFORMATION_SCHEMA
- **userstat**
- **show profiles;**
- PERFORMANCE_SCHEMA
- Profilers (even simple like **pt-pmp** or real like **perf**)
- OS-level tracing and profiling tools
- tcpdump analysis

What is this session about?

- It's about tracing and profiling MariaDB, and some OS level tools MariaDB DBA can use for tracing and profiling **in production** on recent Linux versions:
  - Some details about **perf** and adding dynamic probes
  - Few words about **ftrace**
  - Mostly about eBPF, bcc tools and **bpftrace**
- Why not about gprof, Callgrind, Massif, dtrace, SystemTap?
- Why not about Performance Schema? It's disabled by default in MariaDB, to begin with...
- Performance impact of tracing and profiling

Why not about Performance Schema?

- It may be NOT enabled when server was started (the case for MariaDB by default)
- Too much memory used (see MDEV-20216)
- Specific instruments may not be enabled at startup and then it's too late (see Bug #68097)
- Sizing instruments properly may be problematic
- Part of the code or 3rd party plugins may not be instrumented at all or in enough details (see Bug #83912)
- It does not give you a system-wide profiling, just for selected parts of MariaDB server code
- MariaDB Developers do not consider it much useful and prefer to get stack traces...
- Not easy to use (large and complex queries)

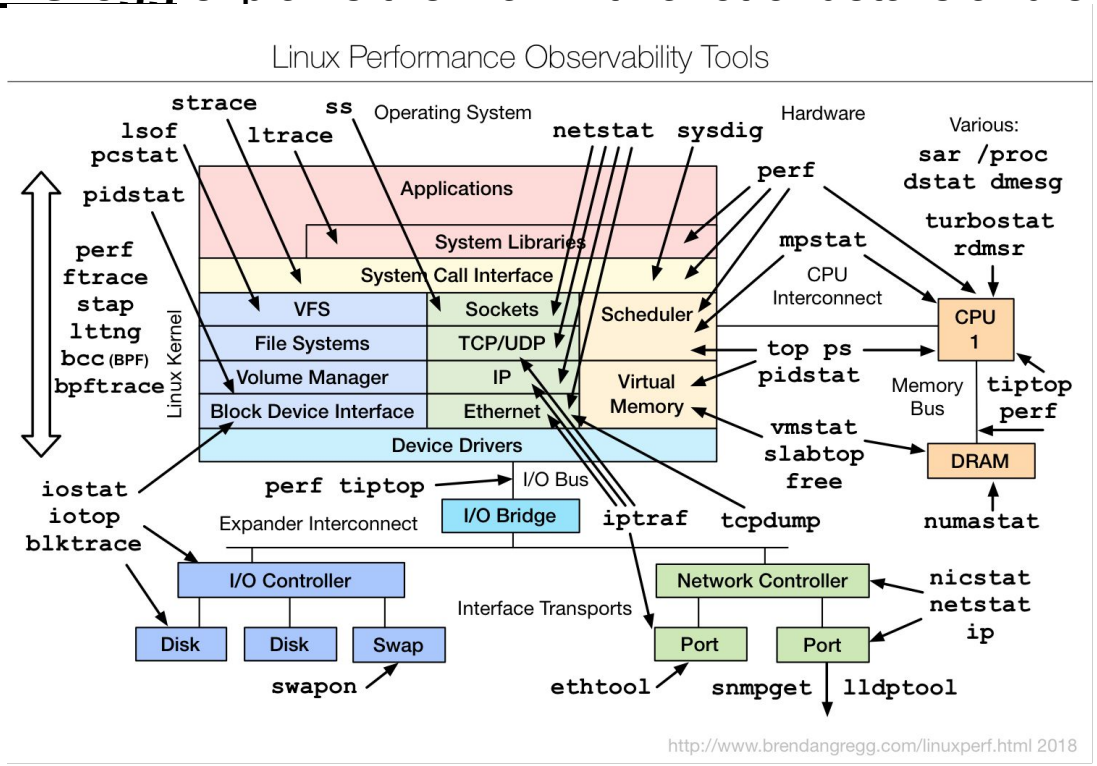
Typical “profiling” query to Performance Schema

- This is how it may look like:

```
SELECT thread_id, event_id, nesting_event_id, CONCAT( CASE WHEN event_name
LIKE 'stage%' THEN
CONCAT(' ', event_name) WHEN event_name LIKE 'wait%' AND
nesting_event_id IS NOT NULL THEN CONCAT(' ', event_name) ELSE
IF(digest_text IS NOT NULL, SUBSTR(digest_text, 1, 64), event_name) END,
' (',ROUND(timer_wait/1000000000, 2),'ms) ') event
FROM (
  (SELECT thread_id,
   event_id, event_name, timer_wait, timer_start, nesting_event_id,
   digest_text FROM events_statements_history_long)
UNION
  (SELECT
   thread_id, event_id, event_name, timer_wait, timer_start,
   nesting_event_id, NULL FROM events_stages_history_long)
UNION
  (SELECT
   thread_id, event_id, event_name, timer_wait, timer_start,
   nesting_event_id, NULL FROM events_waits_history_long)
) events
ORDER BY thread_id, event_id;
```

So, what do I suggest?

- Use modern(!) Linux tracing tools!
- Yes, all that kernel and user probes and tracepoints, **ftrace**, and **perf**, and eBPF (via bcc tools and **bpftrace**), depending on Linux kernel version
- **Julia Evans** explains and illustrates them all here
- **Brendan D. Gregg** explains them all with a lot of details and examples:



Few words on strace: take care!

- **strace** may help MariaDB DBA to find out:
  - what files are accessed by the **mysqld** process or utilities, and in what order
  - why some command (silently) fails or hangs
  - why some commands end up with permission denied or other errors
  - what signals MariaDB server and tools get
  - what system calls could took a lot of time when something works slow
  - when files are opened and closed, and how much data are read
  - where the error log and other logs are really located (we can look for system calls related to writing to stderr, for example)
  - how MariaDB really works with files, ports and sockets
- See my blog post for more details
- **Use in production as a last resort (**2 interrupts per system call, even not those we care about, may leave traced process hanged**)**
- **strace** surely **slows server down**

Few words on DTrace: forget about it...

- **DTrace** is "the father of tracing"**...** But probes are **not in MariaDB** by default
- full-system dynamic tracing framework originally developed by Sun Microsystems (for Solaris)
- Until recently license issues prevented direct use of DTrace on Linux. Not any more since 2018 (Oracle released it as GPL)
- As of Linux 4.9, the Linux kernel finally has similar raw capabilities as DTrace. This is the culmination of many tracing projects and technologies that were merged in the Linux kernel over the years, including: profiling and perf_events, kernel static tracing (*tracepoints*), kernel dynamic tracing (*kprobes*), and user dynamic tracing (*uprobes*).
- There is some "DTrace compatibility". The Linux tracing ecosystem developers decided to stay source-compatible with the DTrace API, so any DTRACE_PROBE macros are automatically converted to *USDT probes*
- If you use Oracle Linux you can try it. Making it work on Fedora 29 took me too much time to complete last year...

Few words on SystemTap: forget about it!

- **SystemTap** is a scripting language and tool for dynamically instrumenting running Linux systems
- SystemTap files are written in the SystemTap language (saved as **.stp** files) and run with the **stap** command-line Scripts (after some checks) are usually compiled into a l**oadable kernel module** (!). Consider risk here...
- SystemTap is a powerful tracer. It can do everything: profiling, tracepoints, kprobes, uprobes (which came from SystemTap), USDT, in-kernel programming, etc
- kernel debuginfo seems to be needed to use all features
- It helped a lot in some practical cases, like finding a process that sent signal to MySQL server. Percona blog provides nice examples.

A lot about (tracing) events sources

- So, *tracing* is basically *doing something* whenever specific *events* occur
- Event data can come from the kernel or from userspace (apps and libraries). Some of them are automatically available without further upstream developer effort, others require manual annotations:

	Automatic	**Manual annotations**
Kernel	kprobes	Kernel tracepoints
Userspace	uprobes	USDT

- *Kprobe* - the mechanism that allows tracing any function call inside the kernel
- *Kernel tracepoint* - tracing custom events that the kernel developers have defined (with TRACE_EVENT macros).
- *Uprobe* - for tracing userspace function calls
- *USDT* (e.g. DTrace probes) stands for *Userland Statically Defined Tracing*

On frontends to events sources

- *Frontends* are tools that allow users to easily make use of the event sources
- Frontends basically operate like this:
  a. The kernel exposes a mechanism – typically some ***/proc*** or ***/sys*** file that you can write to – to register an intent to trace an event and what should happen when an event occurs
  b. Once registered, the kernel looks up the location in memory of the kernel/userspace function/tracepoint/USDT-probe, and modifies its code so that *something else* happens. Yes, **the code is modified on the fly!**
  c. The result of that "something else" can be collected later through some mechanism (like reading from files).
- Usually you don't want to do all these by hand (with **echo**, **cat** and text processing tools via **ftrace**)! Frontends do all that for you
- **perf** is a frontend
- **bcc** and related tools are frontends
- **bpftrace** is a frontend

Few words about ftrace: do not bother much...

- **ftrace** - “a kind of janky interface which is a pain to use directly”. Basically there’s a filesystem at **/sys/kernel/debug/tracing/** that lets you get various tracing data out of the kernel. It supports kprobes, uprobes, kernel tracepoints and UDST can be hacked.
- The way you fundamentally interact with **ftrace** is:
  - Write to files in **/sys/kernel/debug/tracing/**
  - Read output from files in **/sys/kernel/debug/tracing/**

```
[openxs@fc29 ~]$ sudo mount -t tracefs nodev /sys/kernel/tracing
[openxs@fc29 ~]$ sudo ls /sys/kernel/tracing/
available_events              kprobe_profile       stack_trace
available_filter_functions    max_graph_depth      stack_trace_filter
…
[openxs@fc29 ~]$ sudo cat /sys/kernel/tracing/uprobe_events
p:probe_mysqld/dc /home/openxs/dbs/maria10.3/bin/mysqld:0x00000000005c7c93
p:probe_mysqld/dc_1
/home/openxs/dbs/maria10.3/bin/mysqld:0x00000000005c7bb0
```

- Usually is used via some tool (like **trace-cmd**), not directly

Few words about ftrace: if you want to… go for it!

- **ftrace** - let's try to add uprobe for **dispatch_command()** that prints SQL
- Tricky steps are to get probe address (it may be more complex):

```
openxs@ao756:~$ objdump -T /home/openxs/dbs/maria10.3/bin/mysqld | grep dispatch_command
0000000000587b90 g    DF .text  000000000000236e  Base        _Z16dispatch_command19enum_server_commandP3THDPcjbb
```

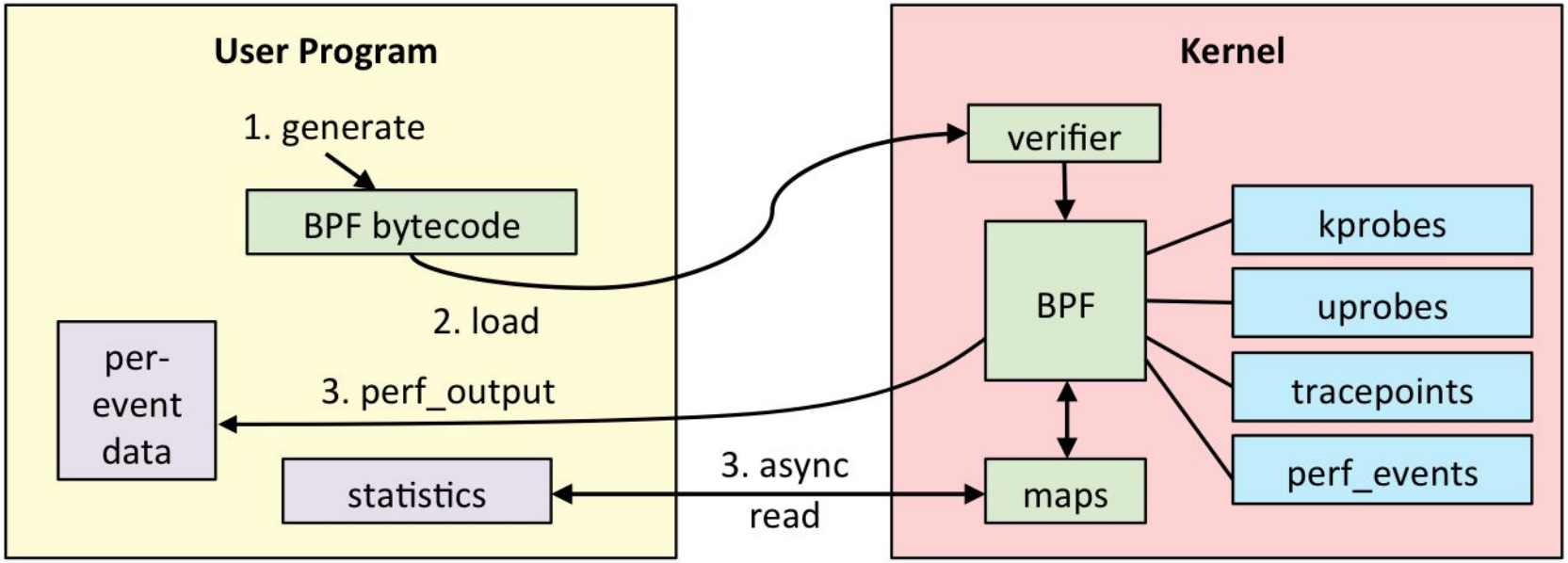
- ...and to work with function arguments (do you know how they are passed?)

```
root@ao756:~# echo 'p:dc /home/openxs/dbs/maria10.3/bin/mysqld:0x0000000000587b90 query=+0(%dx):string' > /sys/kernel/debug/tracing/uprobe_events
root@ao756:~# echo 1 > /sys/kernel/debug/tracing/events/uprobes/dc/enable
root@ao756:~# echo 1 > /sys/kernel/debug/tracing/tracing_on
root@ao756:~# cat /sys/kernel/debug/tracing/trace_pipe
          mysqld-1082  [000] d... 273258.971401: dc: (0x560d8a20fb90) query="select @@version_comment limit 1"
          mysqld-1082  [001] d... 273269.128542: dc: (0x560d8a20fb90) query="select version()"
```

- You can try to do this even with 2.6.27+ kernels (but better 4.x+)
- More details in my recent blog post
- Or just check/use **uprobe** from Brendan Gregg's **ftrace**-based perf-tools

A lot about eBPF: extended Berkeley Packet Filter

- **eBPF** is a tiny language for a VM that can be executed inside Linux Kernel. *eBPF* instructions can be JIT-compiled into a native code. *eBPF* was originally conceived to power tools like *tcpdump* and implement programmable network packed dispatch and tracing. Since Linux 4.1, *eBPF* programs can be attached to *kprobes* and later - *uprobes*, enabling efficient programmable tracing
- **Brendan Gregg** explained it here:



A lot about eBPF

- **Julia Evans** explained it here:
  1. You write an "*eBPF program*" (often in C, Python or use a tool that generates that program for you) for LLVM. It's the "probe".
  2. You ask the kernel to attach that probe to a kprobe/uprobe/tracepoint/dtrace probe
  3. Your program writes out data to an eBPF map / ftrace / perf buffer
  4. You have your precious preprocessed data exported to userspace!
- eBPF is a part of any modern Linux (4.9+):

  4.1 - kprobes
  4.3 - uprobes (so they can be used on Ubuntu 16.04+)
  4.6 - stack traces, **count** and **hist** builtins (use PER CPU maps for accuracy and efficiency)
  4.7 - tracepoints
  4.9 - timers/profiling

- You don't have to install any kernel modules
- You can define your own programs to do any fancy aggregation you want so it's really powerful
- You'd usually use it with some existing **bcc** frontend.
- Recently a very convenient **bpftrace** frontend was added

Examples of bcc tools in action: tplist

- **https://github.com/iovisor/bcc/blob/master/tools/tplist.py**
- This tool displays kernel tracepoints or *USDT probes* and their formats
- Let me apply it to current MariaDB 10.3.x on Fedora 29 (**Fedora build!**):

```
[openxs@fc29 mysql-server]$ sudo /usr/share/bcc/tools/tplist -l
/usr/libexec/mysqld | more

b'/usr/libexec/mysqld' b'mysql':b'connection__done'
b'/usr/libexec/mysqld' b'mysql':b'net__write__start'
b'/usr/libexec/mysqld' b'mysql':b'net__write__done'
b'/usr/libexec/mysqld' b'mysql':b'net__read__start'
b'/usr/libexec/mysqld' b'mysql':b'net__read__done'
b'/usr/libexec/mysqld' b'mysql':b'query__exec__start'
b'/usr/libexec/mysqld' b'mysql':b'query__exec__done'
...
```

- We get these USDT as they were added to the code when DTrace static probes were added. See also **readelf -n**.
- MariaDB does NOT care about DTrace any more, but probes are there (**--DENABLE_DTRACE=1**). Not in MySQL 8.0.1+ it seems

Examples of bcc tools in action: mysqld_qslower

- **https://github.com/iovisor/bcc/blob/master/tools/mysqld_qslower.py**
- Depends on **query__start** and **query__done** UDST probes!
- USAGE: mysqld_qslower PID [min_ms]
- By defaults logs queries slower than 1 millisecond. Set to 0 to have all queries logged. Does not seem to work with prepared statements!
- Let me apply it to current MariaDB 10.3.18 on Fedora 29:

```
[openxs@fc29 tmp]$ sudo /usr/share/bcc/tools/mysqld_qslower `pidof mysqld`
Tracing MySQL server queries for PID 4642 slower than 1 ms...
TIME(s)      PID          MS QUERY
...
0.698114     5955   3546.324 INSERT INTO sbtest3(k, c, pad)
VALUES(501130,
'64733237507-56788752464-03679578678-53343296505-31167207241-1060305090
1-641486789
4.251413     5955161.330 INSERT INTO sbtest3(k, c, pad) VALUES(503408,
'77033721128-77169379656-02480595704-40686156292-96586631730-5187073598
2-037677765
```

- Now run sysbench and measure the impact of this logging vs other options :)

Examples of bcc tools in action: trace

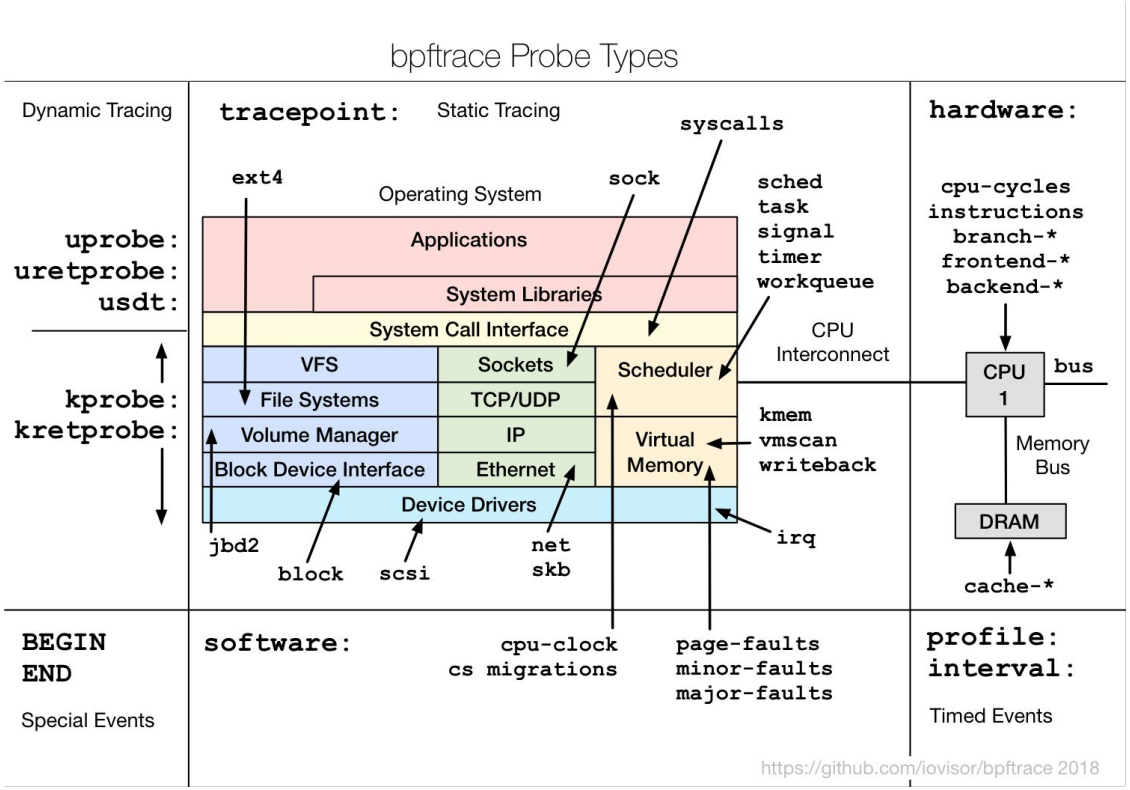
- **https://github.com/iovisor/bcc/blob/master/tools/trace.py**
- Trace a function and print a trace message based on its parameters, with an optional filter.
- Let me apply it to current MariaDB 10.3.x on Fedora 29 to get queries without any UDST used (by adding *uprobe*). I'll attach to function (dispatch_command) and print its 3rd parameter:

```
nm -na /home/openxs/dbs/maria10.3/bin/mysqld | grep dispatch_command
…
00000000005c5180 T _Z16dispatch_command19enum_server_commandP3THDPcjbb
sudo /usr/share/bcc/tools/trace
'p:/home/openxs/dbs/maria10.3/bin/mysqld:_Z16dispatch_command19enum_serv
er_commandP3THDPcjbb "%s" arg3'
PID      TID      COMM         FUNC             -
26140   26225    mysqld
_Z16dispatch_command19enum_server_commandP3THDPcjbb b'select 2'
```

- It seems you have to use mangled name and access to structures may not work easily. See this my blog post for some more details.

What about bpftrace?

- **https://github.com/iovisor/bpftrace**
- **bpftrace** (frontend with programming language) allows to do what I did with trace utility above, but easier and more flexible
- You need recent enough kernel (not available on Ubuntu 16.04), 5.x.y ideally



Study at least one-liner bpftrace examples

- **https://github.com/iovisor/bpftrace/blob/master/docs/tutorial_one_liners.md**
- Command line options
  -l | -e ‘program’ | -p PID | -c CMD | --unsafe | -d | -dd | -v
- Listing probes that match a template:
  ```
  bpftrace -l 'tracepoint:syscalls:sys_enter_*'
  ```
- Tracing file opens may look as follows:
  ```
  # bpftrace -e 'tracepoint:syscalls:sys_enter_openat \
  { printf("%s %s\n", comm, str(args->filename)); }'
  ```
- Syntax is basic:
  ```
  probe[,probe,...] [/filter/] { action }
  ```
- For me the language resembles **awk** and I like it
- More from **Brendan Gregg** (as of August 2019) on it is here
- “Bpftrace is wonderful! Bpftrace is the future!”

Getting stack traces with bpftrace

- See **ustack()** etc in the Reference Guide
- This is how we can use **bpftrace** as a poor man's profiler:

```
sudo bpftrace -e 'profile:hz:99 /comm == "mysqld"/
{printf("# %s\n", ustack(perf));}' > /tmp/ustack.txt
```

- We get output like this by default (**perf** argument adds address etc):

```
...
mysqld_stmt_execute(THD*, char*, unsigned int)+37
dispatch_command(enum_server_command, THD*, char*,
unsigned int, bool, bool)+5123
do_command(THD*)+368
tp_callback(TP_connection*)+314
worker_main(void*)+160
start_thread+234
```

- See my recent blog post for more details on what you may want to do next :)

Performance impact of pt-pmp vs perf vs bpftrace

- Consider **sysbench** (I/O bound) test on Q8300 @ 2.50GHz Fedora 29 box:

```
sysbench /usr/local/share/sysbench/ oltp_point_select.lua
--mysql-host=127.0.0.1 --mysql-user=root --mysql-port=3306  --threads=12
--tables=4 --table-size=1000000 --time=60 --report-interval=5 run
```

- I've executed it without tracing and with the following (compatible?) data collections working for same 60 seconds:

1. **sudo pt-pmp --interval=1 --iterations=60 --pid=`pidof mysqld`**
2. **sudo perf record -F 99 -a -g -- sleep 60**

```
[ perf record: Woken up 17 times to write data ]
[ perf record: Captured and wrote  5.464 MB perf.data (23260 samples) ]
```

3. **sudo bpftrace -e 'profile:hz:99 { @[ustack] = count(); }' > /tmp/bpftrace-stack.txt**

```
[openxs@fc29 tmp]$ ls -l /tmp/bpftrace-stack.txt

-rw-rw-r--. 1 openxs openxs  2980460 Jan 29 12:24 /tmp/bpftrace-stack.txt
```

- Average QPS: 27272 | 15279 (**56%**) | 26780 (98.2%) | 27237 (**99.87%**)

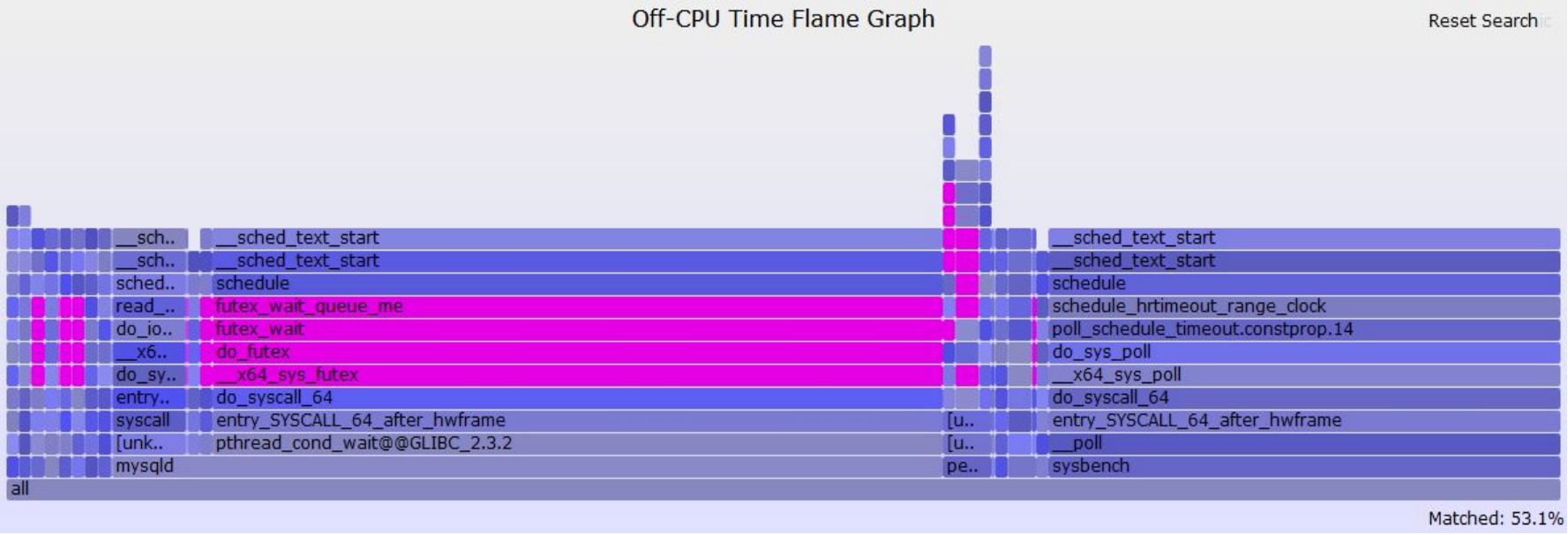
Flame Graphs

- **http://www.brendangregg.com/flamegraphs.html**
- Flame graphs are a visualization (as **.svg** file to be checked in browser) of profiled software, allowing the most frequent code-paths to be identified quickly and accurately.
- The x-axis shows the stack profile population, sorted *alphabetically* (it is not the passage of time), and the y-axis shows stack depth. Each rectangle represents a stack frame. The wider a frame is, the more often it was present in the stacks. Check some examples (on screen :)
- **CPU Flame Graphs** ← profiling by sampling at a fixed rate. Check this post.
- **Memory Flame Graphs** ← tracing **malloc()**, **free()**, **brk()**, **mmap()**, **page_fault**
- **Off-CPU Flame Graphs** ← tracing file I/O, block I/O or scheduler
- More (Hot-Cold, Differential, **pt-pmp**-based etc),
- **https://github.com/brendangregg/FlameGraph** + **perf** + ... or **bcc** tools like **offcputime.py**

Flame Graphs - simple example for off-CPU

- Created based on these steps (while **oltp_update_index.lua** was running):

```
[openxs@fc29 FlameGraph]$ sudo /usr/share/bcc/tools/offcputime -df 60 >
/tmp/out.stacks
WARNING: 459 stack traces lost and could not be displayed.
[openxs@fc29 FlameGraph]$ ./flamegraph.pl --color=io --title="Off-CPU
Time Flame Graph" --countname=us < /tmp/out.stacks > ~/Documents/out.svg
```



Problems of dynamic tracing (with eBPF)

- **root/sudo** access is required
- Debugging the program that is traced …
- Limit memory and CPU usage while in kernel context
- How to add dynamic probe to some line inside the function?
- C++ (mangled names) and access to complex structures (needs headers)
- eBPF tools rely on recent Linux kernels (but Debian 9 uses 4.9+ and RHEL 8 and Ubuntu 18.04.02+ use 4.18 already). Use **perf** for older versions!
- **-fno-omit-frame-pointer** must be used everywhere to see reasonable stack traces
- **-debuginfo**, symbolic information for binaries?
- More tools to install (and maybe build from source)
- Lack of knowledge and practical experience with anything but **gdb** and **perf**
- I had not (yet) used eBPF tools for real life Support issues at customer side (**gdb** and **perf** are standard tools for many customers already).

Am I crazy trying these and suggesting to DBAs?

- Quite possible, maybe I just have too much free time :)
- Or maybe I do not know how to use Performance Schema properly :)
- But I am not alone…
- For open source RDBMS like MariaDB there is *no good reason* NOT to try to use dynamic probes (at least while UDST or Performance Schema instrumentations are not on every other line of the code :)
- **eBPF** (with **bcc** tools and **bpftrace**) makes it easier (to some extent) and *safer* to do this in production

Thank you!

- Thanks to **MariaDB Foundation** and **Anna Widenius** for this “MariaDB Day During FOSDEM” event!
- Questions and Answers?
- Please, report bugs at: https://jira.mariadb.org