Dynamic Tracing for Finding and Solving MySQL Performance Problems on Linux

Valerii Kravchuk, Principal Support Engineer, MariaDB

vkravchuk@gmail.com

Who am I and What Do I Do?

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 MariaDB and MySQL (including some **HowTo**s, used to be mostly bugs marketing)
- **https://www.facebook.com/valerii.kravchuk** - my Facebook page
- **http://bugs.mysql.com** - used to be my personal playground
- **@mysqlbugs** **#bugoftheday** - links to interesting MySQL bugs, few er week
- **MySQL Community Contributor of the Year 2019**
- I speak about MySQL and MariaDB in public. Some slides from previous talks are here and there…
- "I solve problems", "I drink and I know things"

Disclaimers

- Since September, 2012 I act as an Independent Consultant providing services to different companies
- All views, ideas, conclusions, statements and approaches in my presentations and blog posts are mine and may not be shared by any of my previous, current and future employees, customers and partners
- All examples are either based on public information or are truly fictional and has nothing to do with any real persons or companies. Any similarities are pure coincidence :)
- The information presented is true to the best of my knowledge

Sources of tracing and profiling info for MariaDB

- Trace files from **-debug** binaries
- Extended slow query log
- **show [global] status**;
- **show engine innodb status**\G
- **show engine innodb mutex**;
- InnoDB-related tables in the INFORMATION_SCHEMA
- **userstat** - per user, client, table or index
- **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?

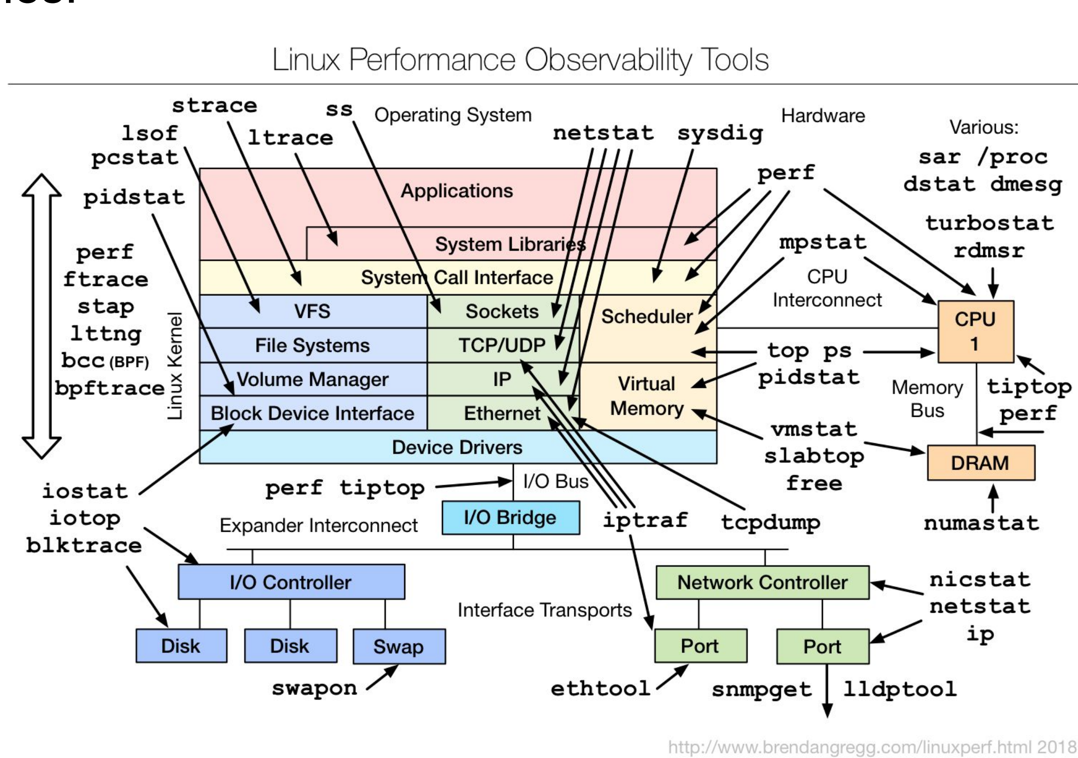
- It's about tracing and profiling MySQL server (or MariaDB server, or any other processes, or even kernel), and some tools for dynamic tracing and profiling **in production** on recent Linux versions:
  - Few words about **ftrace** that is "always there"
  - A lot about **perf** and adding dynamic probes
  - Mostly about eBPF, bcc tools and **bpftrace**
- I plan to present and discuss some (mostly resolvable) dynamic tracing problems one may hit with MySQL server
- Why not about Performance Schema?
- Performance impact of tracing and profiling

Why not about Performance Schema?

- Discussed elsewhere a lot
- It may be NOT enabled when server was started (the case for MariaDB by default) or built (performance impact?)
- Specific instruments may not be enabled at startup and then it's too late (see Bug #68097) - not dynamic enough!
- 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 MySQL server
- MariaDB Developers do not consider it that useful and prefer to get stack traces… (see **pt-pmp**)
- Not easy to use (large and complex queries), **sys** helps...

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
- **Brendan D. Gregg** explains them all (also in a good book) with a lot of details and examples:



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 user space 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
```

- 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/maria10.5/bin/mariadbd |
grep dispatch_command
000000000070a170 g    DF .text  000000000000289b  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/maria10.5/bin/mariadbd:0x000000000070a170
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
        mariadbd-22196 [000] d... 260006.251430: dc: (0x5592713fa170) query="select
1+3"
        mariadbd-22196 [001] d... 260008.899395: dc: (0x5592713fa170) query="select
version(), connection_id()"
```

- You can try to do this even with 2.6.27+ kernels, CentOS 6, (but better 4.x+)
- More details in my blog post
- Or just check/use **uprobe** from Brendan Gregg's **ftrace**-based perf-tools

Dynamic tracing with ftrace: problems

- You need **sudo** or even **root** access
- How to to get the probe address? Consider complex cases of C++ class methods, like in this post with **perf**
- How to work with function arguments? What if they are complex classes or structures like **THD**?
- Return probes: this is easy: **r** instead of **p** and you can access **$retval**
- Don't forget to remove the probe when done:

```
root@ao756:~# echo '-:dc' > /sys/kernel/debug/tracing/uprobe_events
```

- In some cases it may help to add probe with **perf** and then check the details in **/sys/kernel/tracing/uprobe_events**
- A lot of extra reading here..
- I plan to write more blog posts on **ftrace** when I find out something...

A lot about perf

- If you are interested in details presented nicely...
- Or even more details…
- But basically with Linux 2.6.31+ (since 2009) install **perf** package and try it:

```
# perf
 usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]
 The most commonly used perf commands are:
...
   record          Run a command and record its profile into perf.data
   report          Read perf.data (created by perf record) and display the
profile
   sched           Tool to trace/measure scheduler properties (latencies)
   script          Read perf.data (created by perf record) and display
trace output
   stat            Run a command and gather performance counter statistics
...
   top             System profiling tool.
   probe           Define new dynamic tracepoints
   trace           strace inspired tool

 See 'perf help COMMAND' for more information on a specific command.
```

Adding uprobe to MariaDB 10.5 with perf

- The idea is still to add dynamic probe to capture SQL queries
- This was done on Ubuntu 16.04 with recent (at the moment) MariaDB 10.5.6
- First I had to find out with **gdb** where is the query
- Then it’s just as easy as follows:

```
openxs@ao756:~$ sudo perf probe -x
/home/openxs/dbs/maria10.5/bin/mariadbd 'dispatch_command packet:string'

openxs@ao756:~$ sudo perf record -e probe_mariadbd:dispatch_command -aR
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.948 MB perf.data (2 samples) ]

openxs@ao756:~$ sudo perf script >/tmp/queries.txt

openxs@ao756:~$ sudo cat /sys/kernel/tracing/uprobe_events
p:probe_mariadbd/dispatch_command
/home/openxs/dbs/maria10.5/bin/mariadbd:0x000000000070a170
packet_string=+0(%dx):string

openxs@ao756:~$ sudo perf probe --del dispatch_command
```

Adding uprobe to MariaDB 10.5 with perf

- We have queries captured with probe added on previous slide:

```
openxs@ao756:~$ cat /tmp/queries.txt
    mariadbd 11063 [001]  2492.149290: probe_mariadbd:dispatch_command:
(556a2921d170) packet_string="select 1"
    mariadbd 11063 [000]  2496.934324: probe_mariadbd:dispatch_command:
(556a2921d170) packet_string="select version()"
```

- We can control output format, but basically we see binary, PID, CPU where uprobe was executed on, timestamp (milliseconds since start of record), probe and variables with format we specified
- **perf** is easier to use than **ftrace** directly, but we can use it to see the way to add probes with **ftrace** too.
- We do not need to find addresses, understand the way parameters are passed, and usually can access structure fields etc, but studying the source code of the specific version and **gdb** checks are still essential

Adding uprobe with perf: problems

- The idea of the time was still to add dynamic probe to capture SQL queries
- This was done on Fedora 29 with recent MariaDB 10.3.x and something went wrong (event though it worked with **trace** bcc tool). I was NOT able to add uprobe to print **com_data->com_query.query** in a probe on **dispatch_command**
- So I tried with another function, **do_command()**, and its local variable, **packet**::

```
[openxs@fc29 ~]$ sudo perf probe -x
/home/openxs/dbs/maria10.3/bin/mysqld 'do_command packet'

Sorry, we don't support this variable location yet.

  Error: Failed to add events.
```

- I could print **THD *thd** parameter, but got no luck with structure members...

Adding uprobe with perf: problems

- **--vars** option shows what we can access:

```
[openxs@fc29 ~]$ sudo perf probe -x
/home/openxs/dbs/maria10.3/bin/mysqld --vars do_command

Available variables at do_command

    @<do_command+0>

            THD* thd
```

- But **--vars** did not allow to directly access local variables, moreover, at function entry their values may be of no interest
- So I needed a way to create the probe for the specific line of code, where some local variable already had the value set
- **--line** option shows what lines we can “probe”

Adding uprobe with perf: --line option

- **--line** option shows what lines we can “probe”. You add probe for function entry and then:

```
[openxs@fc29 ~]$ sudo perf probe -x /home/openxs/dbs/maria10.3/bin/mysqld
--line do_command
<do_command@/mnt/home/openxs/git/server/sql/sql_parse.cc:0>
     0  bool do_command(THD *thd)
     1  {
     2    bool return_value;
     3    char *packet= 0;
 ...
    10    enum enum_server_command command;
    11    DBUG_ENTER("do_command");
...
   154  packet[packet_length]= '\0';                /* safety */


   157  command= fetch_command(thd, packet);
...
```

Adding uprobe to the line inside function with perf

- So I tried to attach probe to line 157 (relative to **do_command** start) and print local variable as it is there::

```
[openxs@fc29 ~]$ sudo perf probe -x /home/openxs/dbs/maria10.3/bin/mysqld
'do_command:157 packet:string'

Added new events:

  probe_mysqld:do_command (on do_command:157 in
/home/openxs/dbs/maria10.3/bin/mysqld with packet:string)

...

[openxs@fc29 ~]$ sudo perf record -e 'probe_mysqld:do_command*' -aR
    ^C[ perf record: Woken up 1 times to write data ]

[ perf record: Captured and wrote 1.254 MB perf.data (14 samples) ]

[openxs@fc29 ~]$ sudo perf script > /tmp/trace.out
```

It works!

- Let me check the **script** (raw data) output from **perf**::

```
[openxs@fc29 ~]$ cat /tmp/trace.out
        mysqld 15268 [002] 24115.673783:   probe_mysqld:do_command:
(556c4297eb11) packet_string="select 2"
        mysqld 15268 [002] 24118.361743:   probe_mysqld:do_command:
(556c4297eb11) packet_string="select 1"
        mysqld 15245 [001] 24125.281206:   probe_mysqld:do_command:
(556c4297eb11) packet_string="SET time_zone='+00:00'"
        mysqld 15245 [001] 24125.281431:   probe_mysqld:do_command:
(556c4297eb11) packet_string="SHOW STATUS LIKE 'Uptime'"
        mysqld 15245 [001] 24125.282427:   probe_mysqld:do_command:
(556c4297eb11) packet_string=""
        mysqld 15268 [002] 24127.401669:   probe_mysqld:do_command:
(556c4297eb11) packet_string="select user, host from mysql.user"
...
```

- Now, when you’ll get every other row of code instrumented in P_S?
- Use **perf** and dynamic probes in the meantime...

Dynamic tracing with perf: practical examples

- Dynamic tracing of disk I/O with **perf**:
  - What to trace: file I/O, block device I/O, fsync calls, something else?
    ```
    sudo perf record -e block:block_rq_insert -a -g -- sleep 30
    ```
  - More details (by **Brendan Gregg**) here
- Dynamic tracing of memory allocations with **perf**:
  - What if we want to trace all calls to **malloc** and record size and pointer? Depends on allocator library used, but more or less:
    ```
    sudo perf probe -x /lib/x86_64-linux-gnu/libc.so.6 'malloc
    size=%di:s64'
    ```
  - There are problems. More details in this blog post
- Dynamic tracing of C++ class members:
  - What if we want to trace *return* value of **ha_heap::records_in_range**?
  - More details in this blog post
  - Mangled names and, shit happens, virtual member functions.
- Dynamic tracing of **pthread_mutex_lock**, see the blog post

perf - basic usage as a profiler

- Check my post, "perf Basics for MySQL Profiling", for details and references, but basic minimal steps are:
  - Make sure **perf**-related packages are installed (**perf** with RPMs) for your kernel:
    `sudo apt-get install linux-tools-generic`
  - *Make sure debug symbols are installed and software is built with **-fno-omit-frame-pointer***
  - Start data collection for some time using **perf record**:
    `sudo perf record -a [-g] [-F99] [-p `pidof mysqld`] sleep 30`
    Run your problematic load against MySQL server
  - *Samples are collected in **`pwd`/perf.data** by default*
  - Process samples and display the profile using **perf report**:
    `sudo perf report [-n] [-g] --stdio`
- Alternatively, run in foreground and interrupt any time with Ctrl-C:

```
[root@centos ~]# perf record -ag
^C
```

- Or run in background and send **-SIGINT** when done:

```
[root@centos ~]# perf record -ag &
[1] 2353
[root@centos ~]# kill -sigint  2353
```

- See also **perf top** and this blog post by **Daniel Black**

Perf - Call Graphs (hanging at “statistics” case)

- This real life case got me converted from P_S to profiling with **perf** in **2016**...
- See my blog post for details and outputs (**perf record -a -g -F99 sleep 60**):

```
|         |--71.70%-- srv_conc_enter_innodb(trx_t*)
|         |           ha_innobase::index_read(...)
|         |           handler::index_read_idx_map(...)
|         |           handler::ha_index_read_idx_map(...)
|         |           join_read_const(st_join_table*)
|         |           join_read_const_table(THD*, ...)
|         |           make_join_statistics(JOIN*, ...)
|         |           JOIN::optimize_inner()
|         |           JOIN::optimize()
|         |           mysql_select(THD*, ...)
...
```

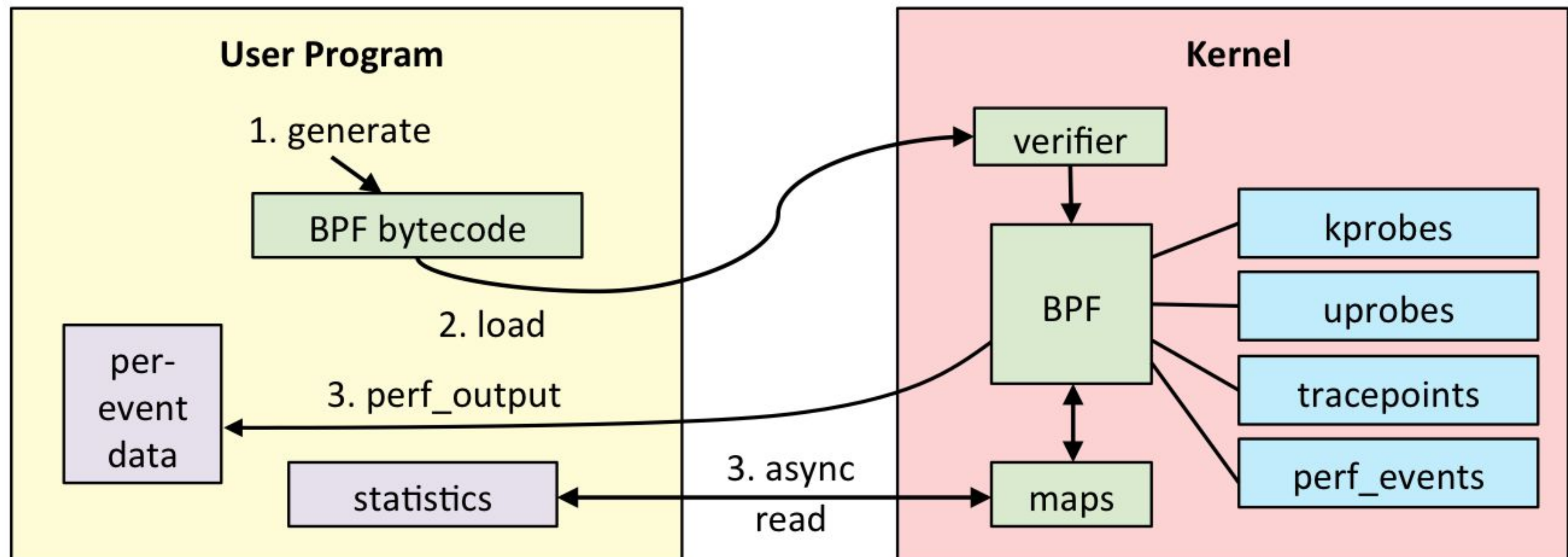
- We can see that time to do **SELECT** is mostly spent waiting to enter InnoDB queue while reading data via index (dive) to get statistics for the optimizer (see **Bug #83912** for what P_S shown me back then)
- We can see where the time is spent by kernel and other processes (**-a**)

perf - more problems and challenges

- Large size of **perf.data** at high sampling rates:
  `-rw------- 1 openxs openxs  33553324 тра 15 00:25 perf.data_io`
  The above is for block I/O only, not I/O bound case:
  `sudo perf record -e block:block_rq_insert -a -g -- sleep 60`
- Overhead of writing by **perf record** to the file (what if you profile I/O and are already I/O bound?)
- The answer is… eBPF and safe summarizing in the probe (in kernel context)
- For complex software like MariaDB **perf** produces too large data sets to study efficiently
- The answer is filtering (with **grep**) and … visualisation as Heat Maps or (for **-g**) Flame Graphs

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. Check some here.
- 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
- It was applied it to MariaDB 10.3.x on Fedora 29 (**Fedora package!**):

```
[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: uprobe with trace

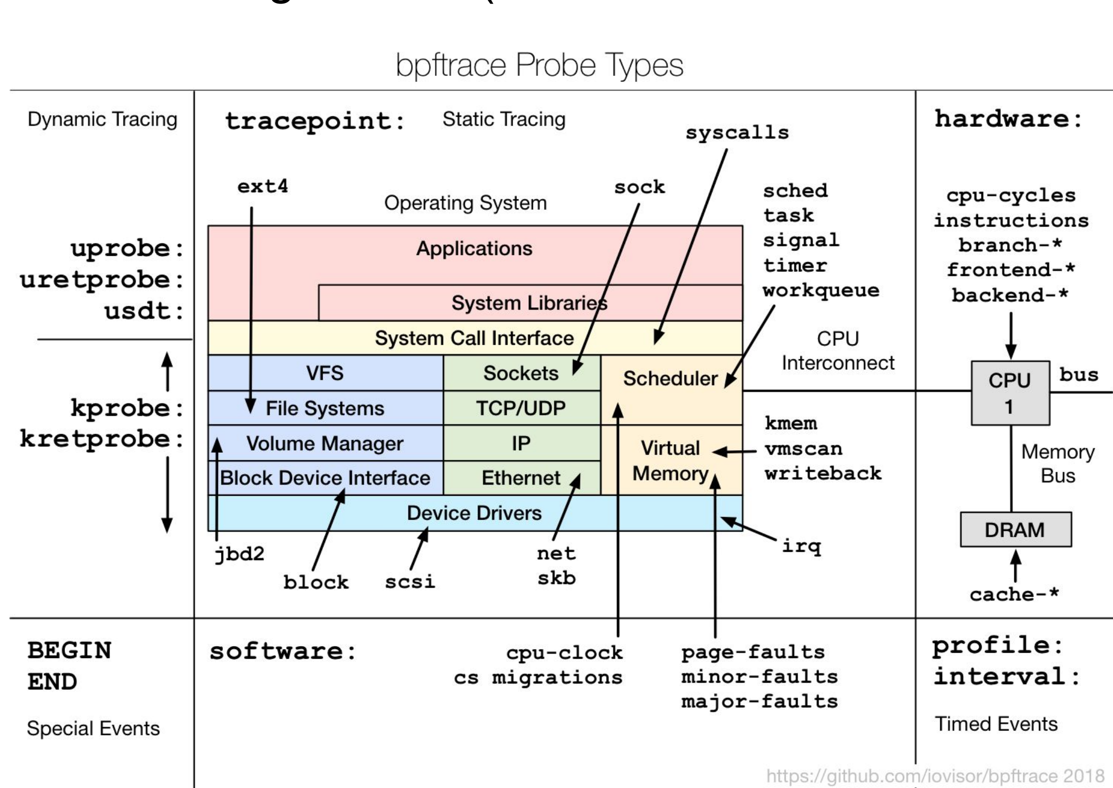
- **https://github.com/iovisor/bcc/blob/master/tools/trace.py**
- Trace a function and print a trace message based on its parameters, with an optional filter.
- It was applied to MariaDB 10.3.x on Fedora 29 to get queries without any UDST used (by adding *uprobe*). As usual I attached to function (d**ispatch_command**) and printed 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%**)

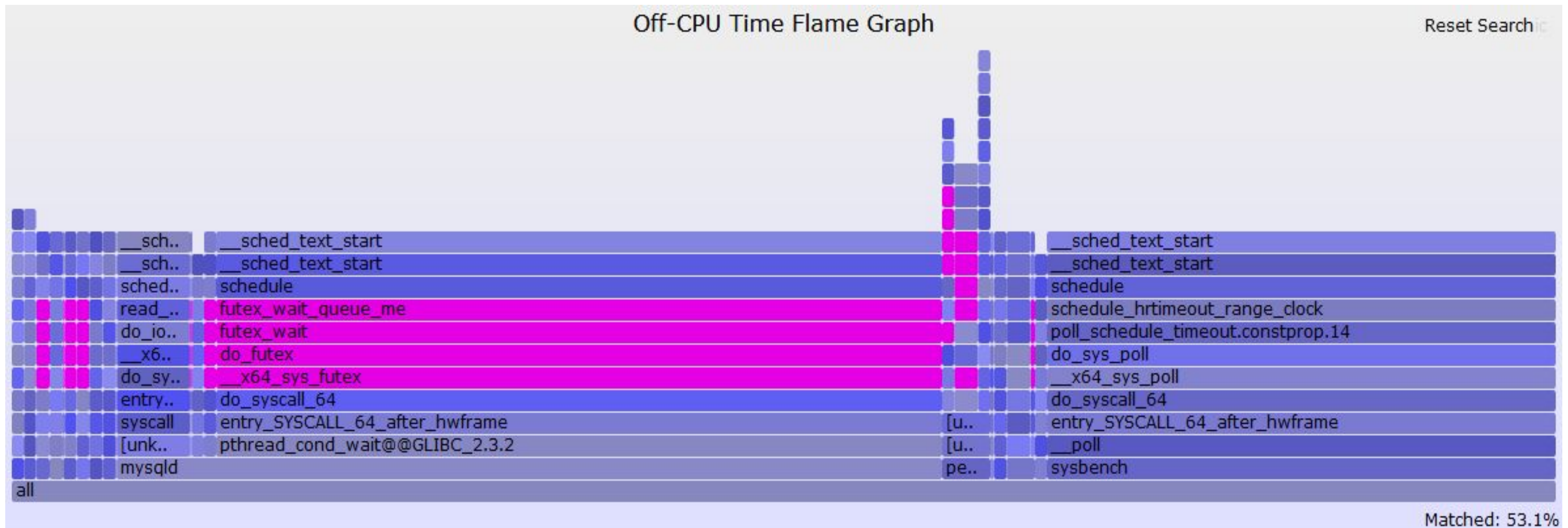
Flame Graphs

- **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: summary

- **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 (doable in **perf**)?
- C++ (mangled names, class members, virtual member functions) and access to complex structures (**bpftrace** needs headers)
- eBPF tools rely on recent Linux kernels (4.9+). Use **perf** for older versions!
- **-fno-omit-frame-pointer** must be used everywhere to see reasonable stack traces
- **-debuginfo** packages, symbolic information for binaries?
- More tools to install (and maybe build from source), but **ftrace** is there… In case of eBPF, there are solutions too: BTF & CO-RE
- 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… **Markos Albe** also speaks about **perf** and **eBPF/bcc** tools, **Daniel Black** writes and speaks about **perf**....
- **perf** probes are used for tracing Oracle RDBMS! There is enough instrumentation there for almost everything, but still...
- Dynamic tracers are proven tools for **instrumenting OS calls** (probes for measuring I/O latency at microsecond precision, for example)
- Dynamic tracing of RDBMS **userspace** is a topic of growing interest, with a lot of RAM and workloads that are often CPU-bound these days.
- For open source RDBMS like MariaDB or MySQL 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 + BTF and CO-RE and **bpftrace**) in long term makes it easier (to some extent) and *safer* to do this in production