

Monitoring MariaDB Server with bpftrace on Linux

Problems and Solutions

Valerii Kravchuk, Principal Support Engineer, MariaDB

valerii.kravchuk@mariadb.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://mysqlextomologist.blogspot.com> - my blog about MariaDB and MySQL (including some **HowTos**, not only bugs marketing)
- <https://www.facebook.com/valerii.kravchuk> - my Facebook page
- [@mysqlbugs](#) #bugoftheday
- **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 am 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

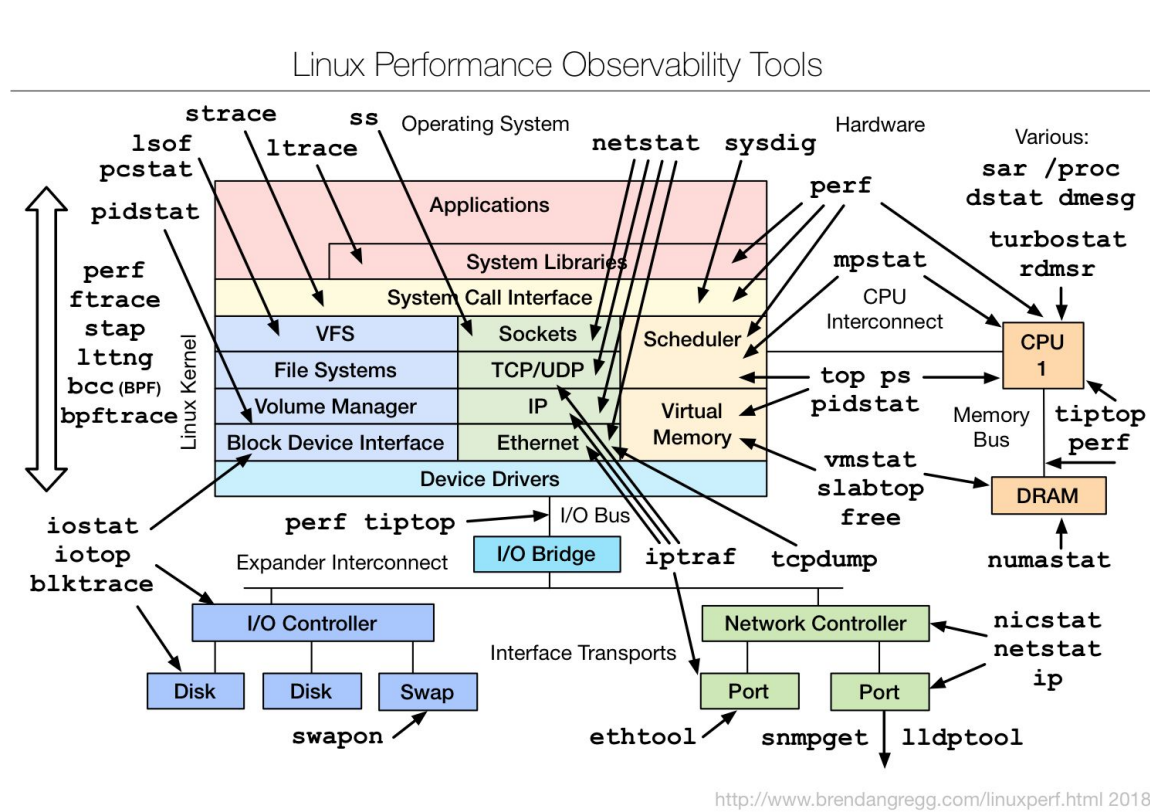
- Trace files from **-debug** binaries
- Extended slow query log
- **show [global] status, show engine innodb status\G**
- InnoDB-related tables in the INFORMATION_SCHEMA
- userstat - operations per user, client, table or index
- **show profiles;**
- PERFORMANCE_SCHEMA (MariaDB 10.5 added memory instrumentation and some more)
- **OS-level tracing and profiling tools:**
 - **/proc** sampling
 - **ftrace** and **perf** profiler
 - eBPF, **bcc** tools and **bpfftrace**
- tcpdump analysis

What is this session about?

- It's about tracing and profiling MariaDB server in **production** on recent Linux versions with eBPF-based **bpftrace** tool
- I plan to present and discuss some (mostly resolvable) dynamic tracing problems one may hit with MariaDB server
- Performance impact of tracing and profiling in production matters
- Why not about Performance Schema?
- Why not about **perf** and **bcc** tools?

So, what do I suggest?

- Use modern Linux tracing tools while troubleshooting MariaDB server!
- Yes, all that kernel and user probes and tracepoints, with **bpftrace** if Linux kernel version allows to use it
- **Brendan D. Gregg** explained the role of **bpftrace** in the global picture:



Tracing events sources

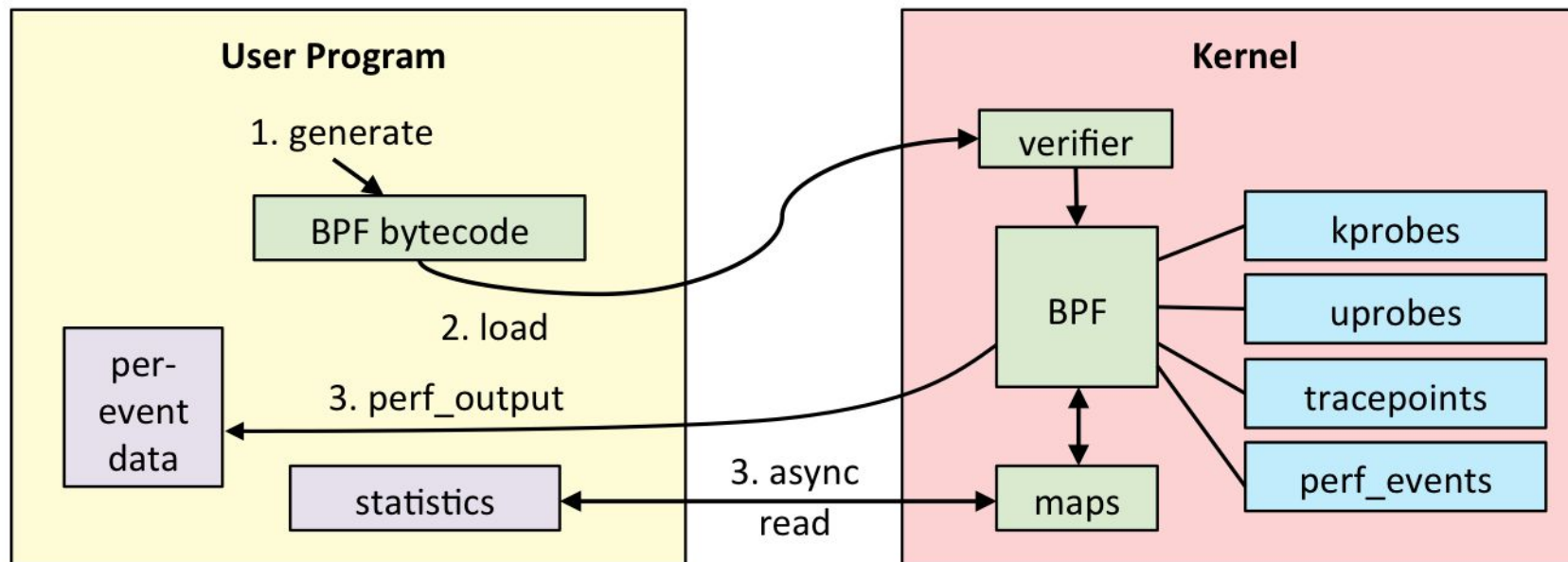
- So, *tracing* is basically *doing something* whenever specific *events* occur
- Event data can come from the kernel or from userspace (apps and libraries). Some of them are automatically available without further upstream developer effort, others require manual annotations:

	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*

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 packet 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](#):

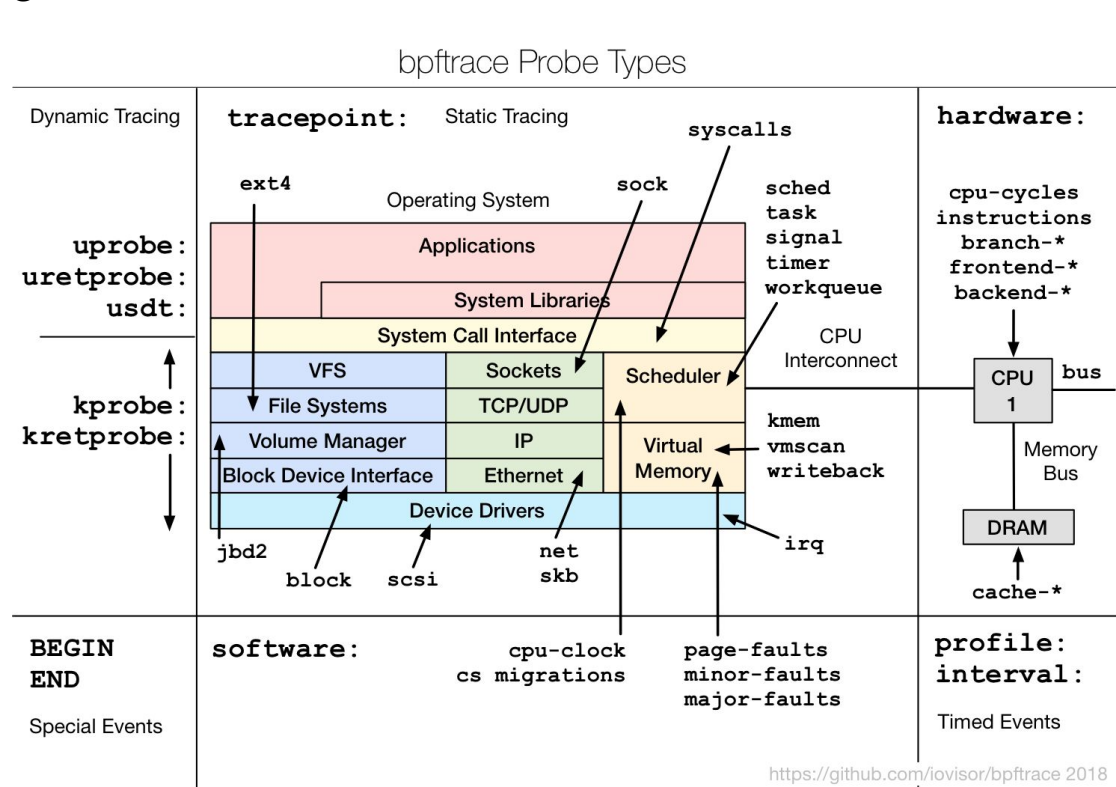


More about eBPF

- **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 (kernel 4.9+):
 - 4.1 - kprobes
 - 4.3 - uprobes
 - 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
- DBAs usually use eBPF via some existing **bcc** frontend. Check some [here](#).
- **Recently a very convenient **bpftool** frontend was added**

bpfftrace as a frontend for eBPF

- bpfftrace (frontend with programming language) allows to define actions for probes presented below in easy and flexible way
- How to start using **bpfftrace**? You need recent enough kernel 5.x.y, install the package or build it from GitHub source and then...



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)); }'`
- The program consists of one or more of the following sequences:
`probe [, probe, ...] [/ filter /] { action }`
- For me the language resembles **awk** (see the [Reference Guide](#))
- More from **Brendan Gregg** (as of August 2019) on it is [here](#)
- ["Bpfftrace is wonderful! Bpfftrace is the future!"](#)

Check and use existing bpftrace programs

- They are in the tools subdirectory:

```
[root@fc31 tools]# ls
bashreadline.bt          loads_example.txt      syscount_example.txt
bashreadline_example.txt mdflush.bt             tcpaccept.bt
biolatency.bt          mdflush_example.txt   tcpaccept_example.txt
biolatency_example.txt naptime.bt            tcpconnect.bt
biosnoop.bt           naptime_example.txt   tcpconnect_example.txt
biosnoop_example.txt  oomkill.bt            tcpdrop.bt
biostacks.bt         oomkill_example.txt   tcpdrop_example.txt
biostacks_example.txt opensnoop.bt         tcplife.bt
bitesize.bt         opensnoop_example.txt tcplife_example.txt
bitesize_example.txt    pidpersec.bt           tcpretrans.bt
capable.bt              pidpersec_example.txt tcpretrans_example.txt
capable_example.txt     runqlat.bt             tcpsynbl.bt
```

...

- Ready to use for ad hoc OS level tracing and monitoring
- Good examples on how to use kprobes and tracepoints, clean up everything, use **hist()** and other built in functions
- See [my blog post](#) for a lot more details

Adding a uprobe to MariaDB 10.5 with bpftrace

- The idea is to add dynamic probes to capture SQL queries (and their execution times)
- This was done on Fedora 31, see [my blog post](#) for the details
- First I had to find out with **gdb** or code where is the query stored/passed
- I already know that it is in the third argument in this call:
`dispatch_command(enum_server_command, THD*, char*, ...)`
- Then it's just as easy as follows (note the mangled function name):

```
[openxs@fc31 ~]$ sudo bpftrace -e '  
uprobe:/home/openxs/dbs/maria10.5/bin/mariadb:_Z16dispatch_  
command19enum_server_commandP3THDPcjobb { @sql[tid] = str(arg2);  
@start[tid] = nsecs; }  
uretprobe:/home/openxs/dbs/maria10.5/bin/mariadb:_Z16dispatch_  
command19enum_server_commandP3THDPcjobb /@start[tid] != 0/ {  
printf("%s : %u %u ms\n", @sql[tid], tid, (nsecs -  
@start[tid])/1000000); } '
```

Adding a uprobe to MariaDB 10.5 with bpftrace

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

```
Attaching 2 probes...
select sleep(1) : 4029 1000 ms
  : 4029 0 ms
select sleep(2) : 4281 2000 ms
  : 4281 0 ms
select sleep(3) : 4283 3000 ms
  : 4283 0 ms
select sleep(4) : 4282 4000 ms
  : 4282 0 ms
^C
...
```

- We do not need to find addresses, understand the way parameters are passed via CPU registers, and usually can access structure fields etc, but studying the source code of the specific version is still essential
- Version 0.11 understands non-mangled C++ function signatures...

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 == "mariadb"/  
{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 :)

Tracing pthread_mutex_lock with bpftrace

- You can find more details in my [recent blog post](#)
- But basically we need to trace **pthread_mutex_lock** calls in the **libpthread.so.*** and count different stack traces that led to them, then output the summary:

```
[openxs@fc31 ~]$ ldd /home/openxs/dbs/maria10.5/bin/mariadb | grep thread
    libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f3d957bf000)
[openxs@fc31 ~]$ sudo bpftrace -e
'uprobe:/lib64/libpthread.so.0:pthread_mutex_lock /comm == "mariadb"/ {
@[ustack] = count(); }' > /tmp/bpfmutex.txt
^C
```

- Take care about the **performance impact** for tracing frequent events!

```
...
[ 10s ] thds: 32 tps: 658.05 qps: 13199.78 (r/w/o:
9246.09/2634.40/1319.30) lat (ms,95%): 227.40 err/s: 0.00 reconn/s: 0.00
[ 20s ] thds: 32 tps: 737.82 qps: 14752.19 (r/w/o:
10325.44/2951.30/1475.45) lat (ms,95%): 193.38 err/s: 0.00 reconn/s: 0.00
[ 30s ] thds: 32 tps: 451.18 qps: 9023.16 (r/w/o: 6316.56/1804.03/902.57)
lat (ms,95%): 320.17 err/s: 0.00 reconn/s: 0.00
[ 40s ] thds: 32 tps: 379.09 qps: 7585.24 (r/w/o: 5310.19/1516.87/758.18)
lat (ms,95%): 390.30 err/s: 0.00 reconn/s: 0.00
...
```


Performance impact of pt-pmp vs perf vs bpftrace

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

Problems of dynamic tracing with bpftrace

- **root/sudo** access is required
- Limit memory and CPU usage while in kernel context
- Do as much aggregations as possible in the probes
- 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 BTF+CO-RE etc may help.
- I had not (yet) used **bpftrace** 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**....
- 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 there is *no good reason* NOT to try to use dynamic probes (at least while Performance Schema instrumentations are not on every other line of the code :)
- **eBPF** with **bpfftrace** makes it easier (to some extent) and *safer* to do this in production

Thank you!

Questions and Answers?

Please, search and report bugs at: <https://jira.mariadb.org>

