Monitoring and Tracing MySQL or MariaDB Server With Bpftrace

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://mysqlentomologist.blogspot.com** - my blog about MariaDB and MySQL (including some **HowTo**s, not only MySQL bugs marketing)
- I often write about **bpftrace** in my blog...
- **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 information for MySQL, MariaDB or Percona servers

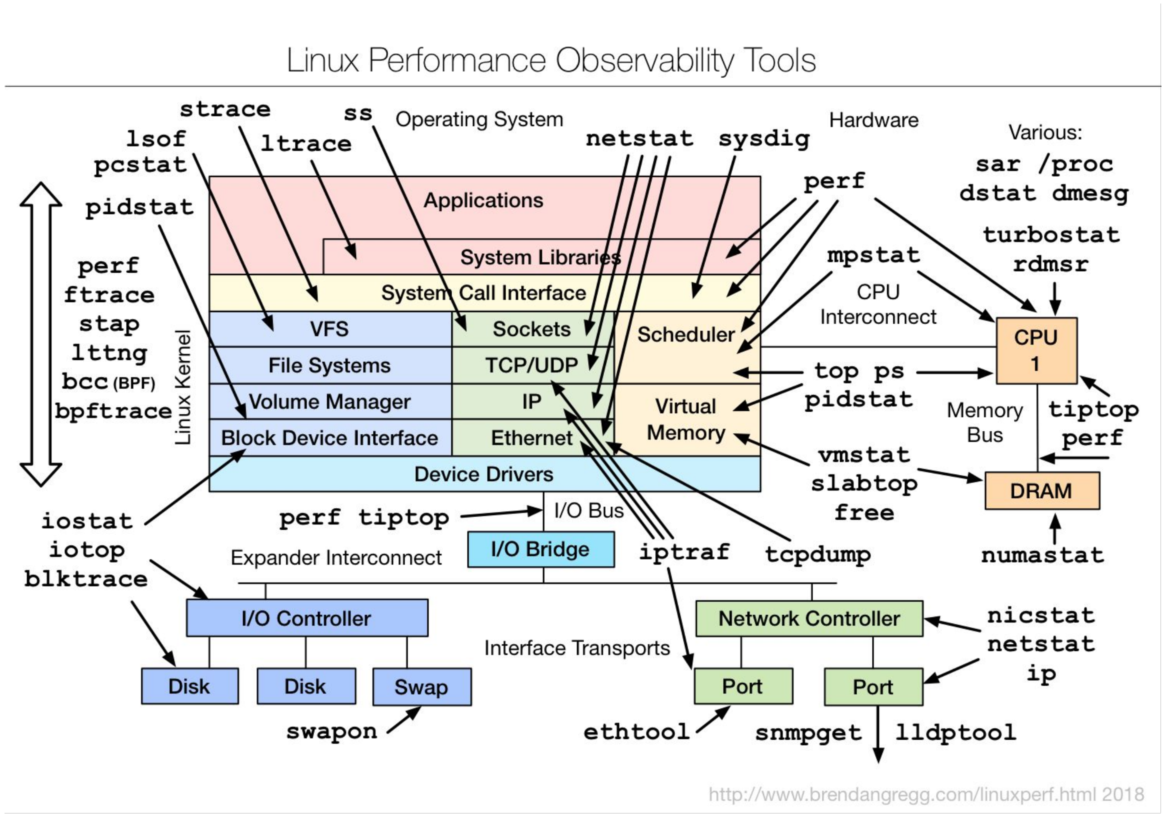
- Trace files from **-debug** binaries
- Extended slow query log (Percona, MariaDB)
- **show** [**global**] **status**, **show engine innodb status**\G
- InnoDB-related tables in the INFORMATION_SCHEMA
- **userstat** - operations per user, client, table or index (Percona, MariaDB)
- **show profiles;**
- **PERFORMANCE_SCHEMA** that was supposed to be an ultimate solution
- OS-level tracing and profiling tools:
  - **/proc** sampling
  - **ftrace** and **perf** profiler
  - eBPF, **bcc** tools and **bpftrace**
- tcpdump analysis

What is this session about?

- It’s about tracing and profiling MySQL or MariaDB server **in production** on recent Linux versions (kernel 5.x.y, the newer the better) with eBPF-based **bpftrace** tool
- I plan to present and discuss some (mostly resolvable) dynamic tracing problems one may hit while working with MySQL or MariaDB servers
- 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 MySQL 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

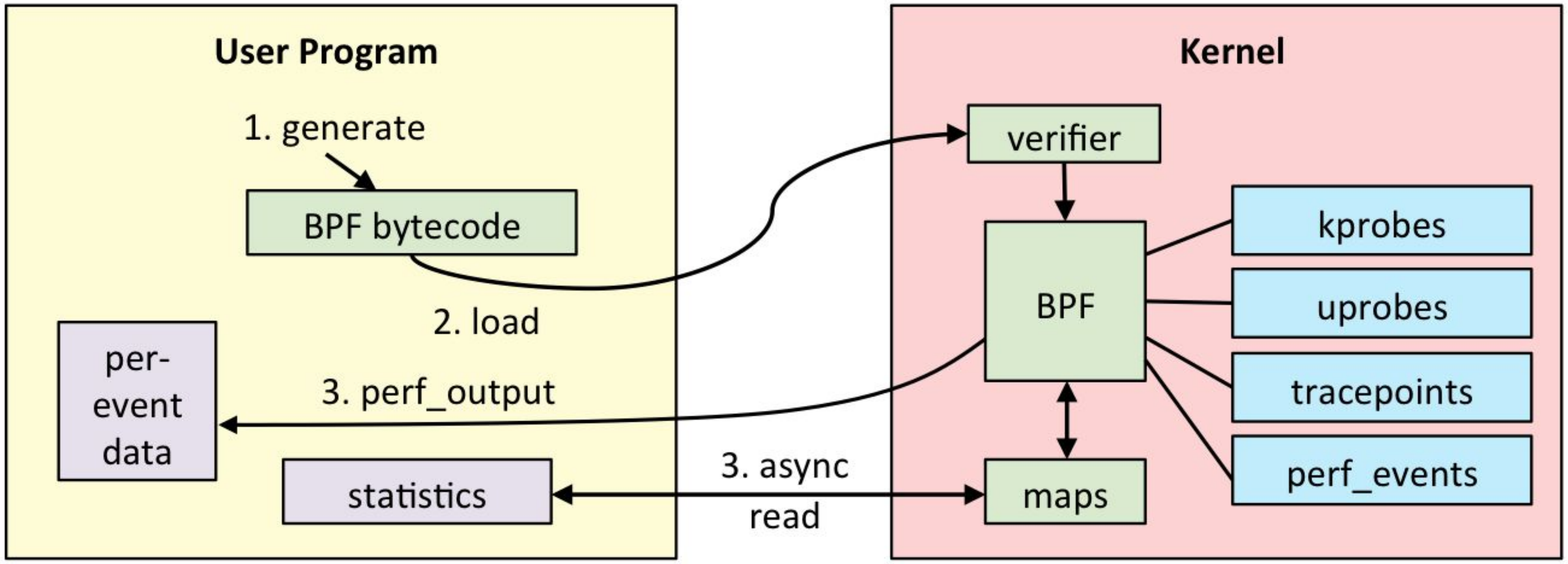
- So, *tracing* is basically *doing something* whenever specific *events* occur
- Event data can come from the kernel or from userspace (apps and libraries). Some of them are automatically available without further upstream developer effort, others require manual annotations:

	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 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:



More about eBPF

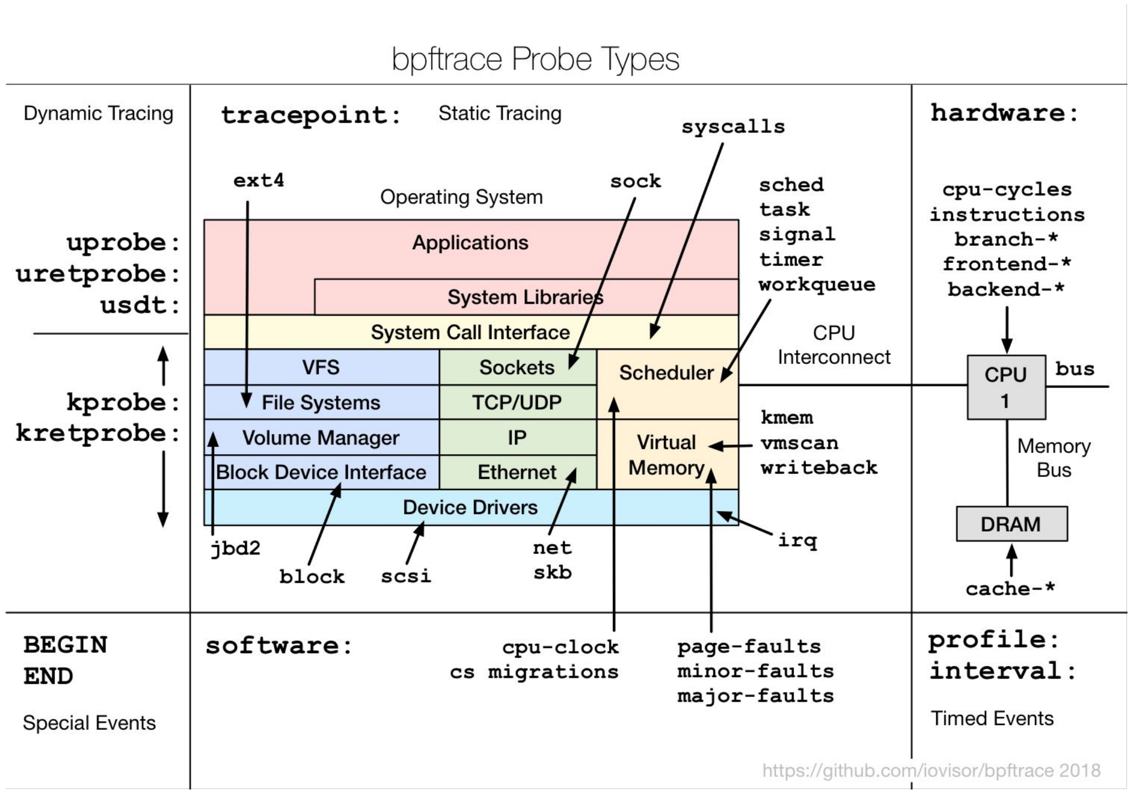
- **Julia Evans** explained it here:
  1. You write an "*eBPF program*" (often in C, Python or use a tool that generates that program for you) for LLVM. It's the "probe".
  2. You ask the kernel to attach that probe to a kprobe/uprobe/tracepoint/dtrace probe
  3. Your program writes out data to an eBPF 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 **bpftrace** frontend was added

bpftrace as a frontend for eBPF

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



Check bpftrace --help output

```
openxs@ao756:~$ bpftrace --version
bpftrace v0.12.0-35-g4dc3
openxs@ao756:~$ bpftrace --help
USAGE:
   bpftrace [options] filename
   bpftrace [options] - <stdin input>
   bpftrace [options] -e 'program'

OPTIONS:
   -B MODE        output buffering mode ('full', 'none')
   -f FORMAT      output format ('text', 'json')
   -o file        redirect bpftrace output to file
   -d             debug info dry run
   -dd            verbose debug info dry run
   -e 'program'    execute this program
   -h, --help     show this help message
```

Check bpftrace --help output (more)

```
    -I DIR          add the directory to the include search path
    --include FILE add an #include file before preprocessing
    -l [search]     list probes
    -p PID          enable USDT probes on PID
    -c 'CMD'        run CMD and enable USDT probes on resulting
process
    --usdt-file-activation
                    activate usdt semaphores based on file path
    --unsafe        allow unsafe builtin functions (and more)
    -q              keep messages quiet
    -v              verbose messages
    --info          Print information about kernel BPF support
    -k              emit a warning when a bpf helper returns an
error (except read functions)
    -kk             check all bpf helper functions
    -V, --version    bpftrace version
    --no-warnings    disable all warning messages
```

Environment variables for bpftrace (--help output)

```
ENVIRONMENT:
    BPFTRACE_STRLEN             [default: 64] bytes on BPF stack per
str()
    BPFTRACE_NO_CPP_DEMANGLE    [default: 0] disable C++ symbol
demangling
    BPFTRACE_MAP_KEYS_MAX       [default: 4096] max keys in a map
    BPFTRACE_CAT_BYTES_MAX      [default: 10k] maximum bytes read by
cat builtin
    BPFTRACE_MAX_PROBES         [default: 512] max number of probes
    BPFTRACE_LOG_SIZE           [default: 1000000] log size in bytes
    BPFTRACE_PERF_RB_PAGES      [default: 64] pages per CPU to allocate
for ring buffer
    BPFTRACE_NO_USER_SYMBOLS    [default: 0] disable user symbol
resolution
    BPFTRACE_CACHE_USER_SYMBOLS [default: auto] enable user symbol
cache
    BPFTRACE_VMLINUX            [default: none] vmlinux path used for
kernel symbol resolution
    BPFTRACE_BTF                [default: none] BTF file
```

Read the Reference Guide - Probes

- The program consists of one or more of the following sequences:

```
probe[,probe,...] [/filter/] { action }
```

- Probes:
  - **BEGIN** - triggered before all other probes are attached
  - **kprobe** - kernel function start
  - **kretprobe** - kernel function return
  - **uprobe** - **user-level function start**
  - **uretprobe** - **user-level function return**
  - **tracepoint** - kernel static tracepoints
  - **usdt** - user-level static tracepoints (not used in MySQL code any more)
  - **profile** - timed sampling
  - **interval** - timed output
  - **software** - kernel software events (see **perf**, like **page-faults** or **cs**)
  - **hardware** - processor-level events (see **perf**, like **cycles** or **cache-misses**)
  - **watchpoint** - memory (by address) watchpoints provided by the kernel (**r**:**w**:**x**)
  - **END** - triggered after all other probes are detached

Read the Reference Guide - Language

- For me the **bpftrace** “language” resembles **awk** (see the **Reference Guide**):
  - **{...}**: Action Blocks - you can group statements separated by **;**
  - **/.../**: Filtering - optional, you can specify any condition using arguments, variables...
  - //, /*: Comments - single line and multi-line comments
  - **->**: C Struct Navigation - you can refer to struct fields with **->** operator
  - **struct**: Struct Declaration - you can declare structures like in C
  - **? :**: ternary operators - like in C
  - **if () {...} else {...}** - conditional execution
  - **unroll () {...}**: repeat the body N times. For example, try this:
    `'BEGIN { $i = 1; unroll(5) { printf("i: %d\n", $i); $i = $i + 1; } exit() }'`
  - **++** and **--:** increment operators - increment or decrement counters in maps or vars
  - **[]**: Array access - to access one-dimensional, constant arrays
  - Integer casts of 64 bit signed integers: **uint8**, **int8**, **uint16**, … **uint64**, **int64**
  - Looping constructs - C style **while** loops, with **continue** and **break**, 5.3+
  - **return:** Terminate Early - exist the current probe (vs **exit()** function)
  - **(,)**: Tuples - you can define variables with tuples, access with . index (**$t.2**)

Read the Reference Guide - Variables

- See “**Variables**”:
  - Builtins - **pid**, **tid**, **uid**, **gid**, **cpu**, **comm**, **retval**, **func**, **probe**, **rand**, **cgroup, arg0**, **arg1**, ..., **argN.** - arguments to the traced function; assumed to be 64 bits wide, and more …
  - **nsecs, elapsed**: - timestamp and elapsed time since **bpftrace** initialization, in nanoseconds
  - **@**, **$**: Basic Variables - global (**@a**), per-thread (**@a[tid]**) and scratch (**$a**)
  - **@[]**: Associative Arrays - BPF map, **@start[tid] = nsecs; @memory[tid,ustack] = retval;**
  - **kstack**: Stack Traces, Kernel - alias for **kstack()** function
  - **ustack**: Stack Traces, User - alias for **ustack()** function
  - **$1**, ..., **$N, $#**: Positional Parameters - command line arguments, may be used in **probe** too

Read the Reference Guide - Functions

- See “**Functions**”:
  - **printf(char *fmt, ...)** - Print formatted
  - **print(...)** - Print a non-map value with default formatting
  - **str(char *s [, int length])** - Returns the string pointed to by **s**
  - **system(char *cmd)** - Execute shell command
  - **exit()** - Quit **bpftrace**
  - **cgroupid(char *path)** - Resolve cgroup ID
  - **kstack([StackMode mode,][int level])** - Kernel stack trace
  - **ustack([StackMode mode,][int level])** - User stack trace
  - **strncmp(char *s1, char *s2, int n)** - Compare first **n** characters of two strings
  - **sizeof(...)** - Return size of a type or expression
  - ...

Read the Reference Guide - Map Functions

- Maps are special BPF data types that can be used to store counts, statistics, and histograms.
- When **bpftrace** exits all maps are printed
- See "**Map Functions**":
  - **count()** - Count the number of times this function is called
  - **sum(int n)**, **avg(int n)**, **min(int n)**, **max(int n)** - sum, average etc
  - **stats(int n)** - Return the count, average, and total for this value
  - **hist(int n)** - Produce a log2 histogram of values of n
  - **lhist(int n, int min, int max, int step)** - Produce a linear histogram of values of n
  - **delete(@x[key])** - Delete the map element passed in as an argument
  - **print(@x[, top [, div]])** - Print the map, optionally the top entries only and with a divisor
  - **clear(@x)** - Delete all keys from the map
  - **zero(@x)** - Set all map values to zero

Study at least one-liner bpftrace examples

- **https://github.com/iovisor/bpftrace/blob/master/docs/tutorial_one_liners.md**
- 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)); }'
```
- Syscall count by program:
```
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm]
= count(); }'
```
- Read size distribution by process:
```
bpftrace -e 'tracepoint:syscalls:sys_exit_read { @[comm]
= hist(args->ret); }'
```
- More from **Brendan Gregg** (as of August 2019) on it is here

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

Code review of biosnoop.bt

- File starts with shebang line and **#include** directive:

```
#!/usr/bin/env bpftrace
#include <linux/blkdev.h>
```

- Comments:

```
/*
* biosnoop.bt  Block I/O tracing tool, showing per I/O latency.
*              For Linux, uses bpftrace, eBPF.
*
* TODO: switch to block tracepoints. Add offset and size columns.*/
```

- **BEGIN** probe to print the header line:

```
BEGIN
{
    printf("%-12s %-7s %-16s %-6s %7s\n", "TIME(ms)", "DISK", "COMM",
"PID", "LAT(ms)");
}
```

Code review of biosnoop.bt (more…)

- Kernel probes for start and end of block I/O:

```
kprobe:blk_account_io_start
{
    @start[arg0] = nsecs;
    @iopid[arg0] = pid;
    @iocomm[arg0] = comm;
    @disk[arg0] = ((struct request *)arg0)->rq_disk->disk_name;
}

kprobe:blk_account_io_done
/@start[arg0] != 0 && @iopid[arg0] != 0 && @iocomm[arg0] != ""/
{
    $now = nsecs;
    printf("%-12u %-7s %-16s %-6d %7d\n",
        elapsed / 1e6, @disk[arg0], @iocomm[arg0], @iopid[arg0],
        ($now - @start[arg0]) / 1e6);

    delete(@start[arg0]);
    delete(@iopid[arg0]);
    delete(@iocomm[arg0]);
    delete(@disk[arg0]);
}
```

Code review of biosnoop.bt (END probe etc)

- We do not want to print maps content at the end, as this is a monitoring tool for interactive use
- So, we clear all the maps in the **END** probe:

```
END
{
    clear(@start);
    clear(@iopid);
    clear(@iocomm);
    clear(@disk);
}
```

- This approach is typical for interactive monitoring tools
- Other option (**syscount.bt**) is to collect the data and print at the end:

```
END {
    printf("\nTop 10 syscalls IDs:\n");
    print(@syscall, 10);
    clear(@syscall); … }
```

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/mariadbd:_Z16dispatch_com
mand19enum_server_commandP3THDPcjbb { @sql[tid] = str(arg2);
@start[tid] = nsecs; }
uretprobe:/home/openxs/dbs/maria10.5/bin/mariadbd:_Z16dispatch_
command19enum_server_commandP3THDPcjbb /@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
- Versions 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 == "mariadbd"/
{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 strack traces that led to them, then output the summary:

```
[openxs@fc31 ~]$ ldd /home/openxs/dbs/maria10.5/bin/mariadbd | grep thread
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f3d957bf000)
[openxs@fc31 ~]$ sudo bpftrace -e
'uprobe:/lib64/libpthread.so.0:pthread_mutex_lock /comm == "mariadbd"/ {
@[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
...
```

Tracing time spent in __lll_lock_wait with bpftrace

- It turned out that tracing uncontended **pthread_mutex_lock** calls may give false alarms. So we changed the approach to measuring time spent waiting...
- Main parts of the new **lll_lock_wait2.bt** tool:

```
...
interval:s:$1 { exit(); }

uprobe:/lib64/libpthread.so.0:__lll_lock_wait
/comm == "mariadbd"/ {
    @start[tid] = nsecs;
    @tidstack[tid] = ustack(perf); }

uretprobe:/lib64/libpthread.so.0:__lll_lock_wait
/comm == "mariadbd" && @start[tid] != 0/ {
    $now = nsecs;
    $time = $now - @start[tid];
    @futexstack[@tidstack[tid]] += $time;
    print(@futexstack);
    delete(@futexstack[@tidstack[tid]]);
    delete(@start[tid]);
    delete(@tidstack[tid]); }
```

- Call stacks and times spent are printed, they are summarized externally!

Tracing MySQL memory allocations with bpftrace

- You can trace and monitor almost anything wth **bpftrace**. For example, you can easily figure out when (and where) and how large memory areas are really allocated:

```
openxs@ao756:~$ sudo bpftrace -e
'uprobe:/lib/x86_64-linux-gnu/libc.so.6:malloc / comm == "mysqld" / {
printf("Allocating %d bytes in thread %u...\n", arg0, tid); }'
Attaching 1 probe...
Allocating 32 bytes in thread 105713...
Allocating 32 bytes in thread 105713...
Allocating 40 bytes in thread 107448...
```

- Now when we run this:

```
select repeat('a',3000000);
```

- We see:

```
Allocating 3000040 bytes in thread 107448...
Allocating 3000048 bytes in thread 107448...
Allocating 40 bytes in thread 107448...
Allocating 16416 bytes in thread 107448...
Allocating 32 bytes in thread 105713...
^C
```

Tracing memory allocations with bpftrace (lame...)

- You can trace and monitor almost anything wth **bpftrace**, but beware of lame approaches that work, but with a cost that makes them impractical:

```
uprobe:/lib64/libc.so.6:malloc /comm == "mariadbd"/ {
    @size[tid] += arg0;
/*  printf("Allocating %d bytes in thread %u...\n", arg0, tid);  */
}

uretprobe:/lib64/libc.so.6:malloc /comm == "mariadbd" && @size[tid] > 0/
{
    @memory[tid,retval] = @size[tid];
    @stack[ustack(perf)] += @size[tid];

    print(@stack);
    clear(@stack);
    delete(@size[tid]); }

uprobe:/lib64/libc.so.6:free / comm == "mariadbd" / {
    delete(@memory[tid, arg0]);
/*  printf("Freeing %p...\n", arg0); */
}
...
```

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 (performance impact**?**)
- How to add dynamic probe to some line inside the function? (possible, probe on offset within function, but had not tried myself yet)
- C++ (mangled names, class members, virtual member functions) and access to complex structures (**bpftrace** needs headers), no stable "API"
- 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**, had to do this even on Ubuntu 20.04), 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).

References to bpftrace in MySQL, MariaDB and Percona public bugs databases...

- MySQL (**site://bugs.mysql.com bpftrace**): zero hits! For **perf** - 288 hits
- MariaDB (**site://jira.mariadb.org bpftrace**):
  - **MDEV-24316** - on migration pre-check tool, “bpftrace script that hooks mysql and counts when incompatible functions are used”, **Daniel Black**
  - **MDEV-20931** - on the lack of information about shutdown progress, “...no way to find this out without OS level tools like gdb, perf or bpftrace”, yours truly…
  - **MDEV-19552** - on DROP TABLE locking SHOW etc, incomplete. “Could you try perf or BPF trace + https://github.com/brendangregg/FlameGraph?”, **Eugene Kosov**
  - **MDEV-23326** - Aria TRANSACTIONAL=1 significantly slow on timezone initialisation, in review already. **Daniel Black**

    ```
    bpftrace -e 'tracepoint:syscalls:sys_enter_fdatasync {
    @start[args->fd] = nsecs; @fd = args->fd}
    tracepoint:syscalls:sys_exit_fdatasync { @us[ustack, @fd] =
    hist((nsecs - @start[@fd]) / 1000); delete(@start[@fd])  } ' -p
    331087
    ```
- Percona (**site://jira.percona.com bpftrace**): zero hits! For **perf** - 147 hits

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 **bpftrace**-based tools, MariaDB engineers starts to “think” in terms of **bpftrace**...
- 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 MySQL or 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 **bpftrace** 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
- https://jira.percona.com
- https://bugs.mysql.com