



Introduction to using DTrace with MySQL

Vince Carbone – Performance
Technology Group, Sun
MC Brown - MySQL

A graphic consisting of several interlocking puzzle pieces. The background of the puzzle is a blue and white image of the Earth from space. The puzzle pieces are arranged in a way that some are missing, creating a fragmented effect.

Innovation Everywhere



Agenda

- Quick DTrace Overview
- Tracing User Applications
- User Process Tracing Case Study
- MySQL Static probes
- What you can do with MySQL Static probes



Quick DTrace Overview

- Dynamic Tracing
- Supported Operating Systems
 - OpenSolaris
 - Mac OSX
- Allows tracing of live processes
- Trace anything
 - Application
 - OS Kernel
 - System Libraries
- Doesn't require special builds, debugger, etc
- Zero overhead when not used

DTrace Providers and Probes

- provider:module:function:name
- provider – name of DTrace provider that publishes this probe
- module – specific program or system library where probe is located
- function – program function where probe is located
- name – name of probe (entry, return)
- Example
 - Syscall count
 - `syscall:::entry { @num[probefunc] = count(); }`
 - Read Syscall by process
 - `syscall::read:entry { @num[pid] = count(); }`

How to use it

- dtrace (1m)
- dtrace -l
 - list all probes available for tracing
- Execute from command line
 - dtrace one-liner
 - `dtrace -n 'syscall:::entry { @num[probefunc] = count(); }'`
- “D” scripting language
 - awk and C like
 - Selective processing
 - Aggregations
 - Formatting



You've enabled a probe, now what?

- Report
 - Counts
 - Timings
 - Capture function arguments
- Aggregations
 - Count
 - Sum
 - Average
- DTrace follows thread of execution
 - Thread local variables
 - self->

D Script Example

```
#!/usr/sbin/dtrace -qs

pid$1::my_thread_setprio:entry
{
    self->trace = 1;
}

pid$1::my_thread_setprio:return
{
    self->trace = 0;
}

syscall::priocntlsys:entry
/ self->trace /
{
    self->ts = timestamp;
}

syscall::priocntlsys:return
/ self->ts /
{
    @count[tid] = count();
    @time[tid] = avg(timestamp - self->ts);
    @sum[tid] = sum(timestamp - self->ts);
    self->ts = 0;
}

END
{
    printf("\nMySQL 5.1.26 on SNV87 X86\n");
    printf("priocntlsys called via my_thread_setprio\n");
    printf("Syscall Count\n");
    printa(@count);
    printf("Average Time\n");
    printa(@time); printf("Total Time\n");
    printa(@sum);
}
```

Tracing User Applications

- DTrace provides two methods of tracing user applications
- PID provider
 - Dynamically creates probes on any function
 - Need to know functions, names what they do
 - Helps if you have access to source code, otherwise hope that function names are meaningful

```
file: trace_my_thread_setprio.d

#pragma D option flowindent

pid$target::my_thread_setprio:entry
{
    self->trace = 1;
}

pid$target::my_thread_setprio:return
/ self->trace /
{
    self->trace = 0;
}

Execute with:

dtrace -s trace_my_thread_setprio.d -p `pgrep 'mysqld'`;
```

Tracing User Applications

- Statically Defined Application Probes
 - Probes added to source code
 - Provide easier hooks into the information you want
 - Identify specific operation
 - Access specific information
 - Much easier to use
 - User doesn't need to know detail of application code

```
mysql*:::query-cache-hit
{
  self->qc = 1;
}

mysql*:::query-cache-miss
{
  self->qc = 0;
}
```

User Process Tracing Case Study

- Investigating MySQL thread scheduling on Solaris
 - Only OS where this is supported
 - Will not be supported in future versions of MySQL
- Sysbench Read-Only OLTP
 - 10,000,000 rows
 - 32 sysbench threads
 - Solaris Nevada X86
 - 4 x Intel Xeon Quad Core (16 cores)

System Level Monitoring

- Start with high level monitoring tools and drill down
- mpstat reports per cpu statistics
 - High system time (26%)
 - 40K system calls/sec/cpu !!!!

```

CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl set
0  2  0  2 1138 159 8229 1975 1000 3289  0 39814  70 27  0  3  0
1  0  0 3047  796 102 8313 1998  984 2979  0 39928  70 26  0  3  0
2  0  0  2  842  1 8703 2090  986 2991  0 41188  70 26  0  3  0
3  0  0  2  843  0 8668 2094  962 2977  0 40924  70 26  0  3  0
4  0  0  72  867  4 8886 2160  953 3002  0 41742  70 26  0  3  0
5  0  0  2  778  2 8409 2123  999 3126  0 41529  72 26  0  2  0
6  0  0  2  891  2 8708 1938 1086 2937  0 40115  70 26  0  4  0
7  0  0  2  826  0 8736 2120  960 3008  0 41348  70 26  0  3  0
8  0  0  42  798  34 8346 2081 1016 3121  0 41200  72 26  0  2  0
9  1  0  46  879  0 8664 1927 1085 2916  0 39859  70 26  0  4  0
10 0  0  2  839  0 8881 2158  975 3024  0 41947  70 26  0  3  0
11 1  0  2  807  0 8630 2135 1032 3102  0 42141  71 26  0  2  0
12 1  0  3  934  0 8934 1984 1096 2882  0 40758  69 27  0  4  0
13 0  0  5  815  0 8651 2123  962 3000  0 41312  70 26  0  3  0
14 0  0  2  788  0 8550 2159 1010 3148  0 42061  72 26  0  2  0
15 0  0  2  905  0 8845 1968 1095 2959  0 40621  69 26  0  4  0

```

Syscall Profile of mysqld

- Drill down using 'truss -c' to profile system calls made by mysqld
 - prcntlsys called 28,994
- What does prcntlsys do?
 - sets scheduling parameters for a thread

```
> $ truss -c -p 21927
> ^C
> syscall      seconds  calls  errors
> read         1.143  29253  259
> write        .698   14497
> time         .092   2823
> lseek        .000    5
> fcntl        .017   519
> lwp_park     .079   1918
> lwp_unpark   .082   1916
> prcntlsys    1.233  28994
> yield       .161   3592
> pwrite       .076   693
> pollsys     .000    13
>
> sys totals:  3.585  84229  259
> usr time:    2.495
> elapsed:    5.040
```

Find priocntlsys in mysqld

- Where in mysqld is priocntlsys being called?
- Use syscall provider and ustack()

```
syscall::priocntlsys:entry
/ execname == 'mysqld' /
{
    @count[ustack(10)] = count();
}

END
{
    printa(@count);
}
```

```
libc.so.1`__priocntlset+0xa
libc.so.1`set_priority+0x72
libc.so.1`setparam+0x6e
libc.so.1`_thr_setparam+0x99
libc.so.1`pthread_setschedparam+0xb
mysqld`my_thread_setprio+0x30
mysqld`_Z18mysql_stmt_executeP3THDPcj+0x268
mysqld`_Z16dispatch_command19enum_server_commandP3THDPcj+0x457
mysqld`_Z10do_commandP3THD+0xc0
mysqld`handle_one_connection+0x5df
libc.so.1`_thr_setup+0x89
libc.so.1`_lwp_start
162468
```



Turn off Thread Scheduling

- What happens when MySQL Thread Scheduling is turned off?
 - `--skip-thread-priority`
- Results
 - With Thread Scheduling
 - 5027 TPS
 - `--skip-thread-priority`
 - 3740 TPS
 - What the `?*&*!*`

Measure Thread Scheduling Impact

- `my_thread_setprio` alters user priority
- What is the impact? 6.8s of 300s run per thread

```
#!/usr/sbin/dtrace -qs

pid$1::my_thread_setprio:entry
{
    self->trace = 1;
}

pid$1::my_thread_setprio:return
{
    self->trace = 0;
}

syscall::priocntlsys:entry
/ self->trace /
{
    self->ts = timestamp;
}

syscall::priocntlsys:return
/ self->ts /
{
    @count[tid] = count();
    @time[tid] = avg(timestamp - self->ts);
    @sum[tid] = sum(timestamp - self->ts);
    self->ts = 0;
}

END
{
    printf("\nMySQL 5.1.26 on SNV87 X86\n");
    printf("priocntlsys called via my_thread_setprio\n");
    printf("Syscall Count\n");
    printa(@count);
    printf("Average Time\n");
    printa(@time);
    printf("Total Time\n");
    printa(@sum);
}
```

MySQL 5.1.26 on SNV87 X86
priocntlsys called via my_thread_setprio

Syscall Count

.	.
.	.
.	.
136	1013548
135	1013630
131	1015086

Average Time

.	.
.	.
.	.
134	6715
135	6739
136	6742

Total Time

.	.
.	.
.	.
134	6781345029
135	6831815241
136	6833573153

What's up in the Scheduler

- Is something weird going on in the scheduler?
 - TS (Timeshare) default scheduling class
 - Priority 0-60, changes based number of factors
 - In General
 - cpu intensive threads get lower priority
 - I/O intensive threads get higher priority



Observe impact of Scheduler

- DTrace 'sched' provider makes available probes related to CPU scheduling
- Who preempts who?

```
#!/usr/sbin/dtrace -s
#pragma D option quiet

sched:::preempt
{
    self->preempt = 1;
}

sched:::remain-cpu
/self->preempt/
{
    self->preempt = 0;
}

sched:::off-cpu
/self->preempt/
{
    /*
     * If we were told to preempt ourselves, see who we ended up giving
     * the CPU to.
     */
    @[stringof(args[1]->pr_fname), args[0]->pr_pri, execname,
     curlwpsinfo->pr_pri] = count();
    self->preempt = 0;
}

END
{
    printf("%30s %3s %30s %3s %5s\n", "PREEMPTOR", "PRI",
        "PREEMPTED", "PRI", "#");
    printa("%30s %3d %30s %3d %5@d\n", @);
}
```

Who is Preempting Who?

- MySQL Thread Scheduling
- `--skip-thread-priority`
- `mysqld` runs at higher priority than `sysbench`
- `mysqld` preempts `sysbench`
- `mysqld` tends to run at lower priority and is preempted by `sysbench`

PREEMPTOR PRI	PREEMPTED PRI #
mysqld 59	mysqld 50 124472
mysqld 59	mysqld 51 125171
mysqld 59	sysbench 37 168411
mysqld 59	sysbench 47 644789

PREEMPTOR PRI	PREEMPTED PRI	Count
sysbench 47	mysqld 9	438072
sysbench 47	mysqld 19	457455
sysbench 47	mysqld 39	459220
sysbench 37	mysqld 0	477657
sysbench 42	mysqld 9	490111
sysbench 37	mysqld 9	573309
mysqld 19	mysqld 9	596944
mysqld 9	mysqld 0	989618

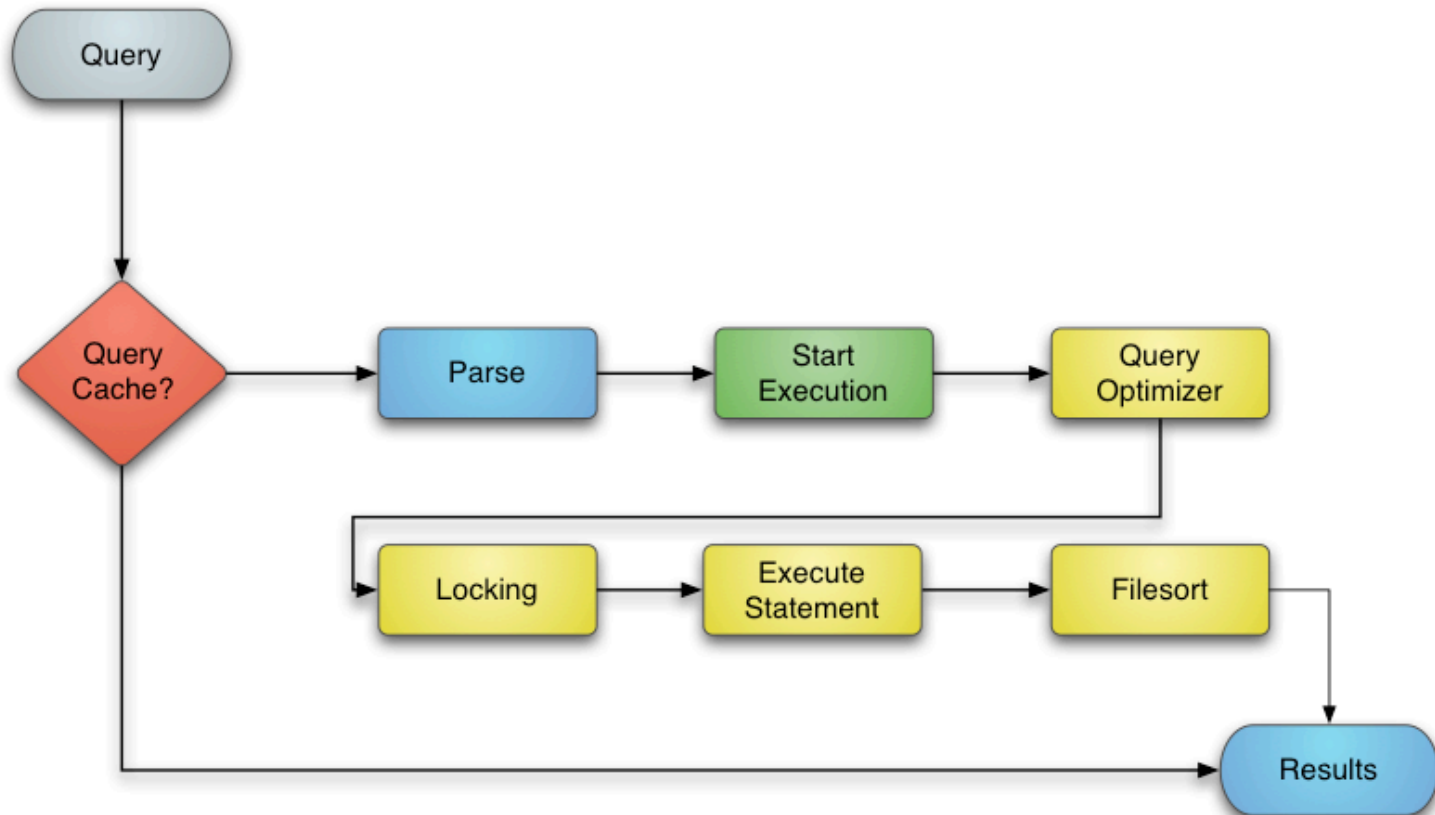


So.....

- As a side effect of calling `pricntlsys` millions of times
 - `mysqld` gets priority increased to 59
- What if?
 - set `mysqld` to class FX priority 59
 - `--skip-thread-priority`
 - Results: 5375 TPS
- For this workload `mysqld` runs best with higher priority than client process

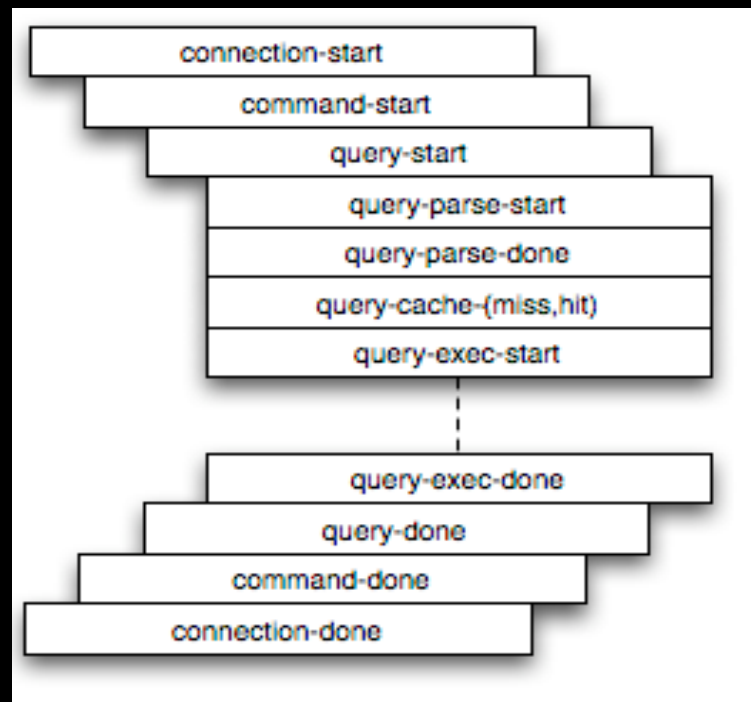
MySQL Static Probes

- Available in MySQL 5.4 and MySQL 6.0.8 and later
- Probes in MySQL are organized to follow the execution path of a query



Probe Hierarchy

- Most are part of the execution sequence with the level getting deeper
- But designed that you don't need to traverse the entire hierarchy to get what you want





Probe Sets

- Connection
- Command
- Query
- Query Parsing
- Query Caching
- Query Execution
- Locks
- Statements
- Row-Level
- Filesort
- Network
- <http://dev.mysql.com/doc/refman/6.0/en/dba-dtrace-server.html>

Getting information from probe

- Probe set
 - -start: information associated w/ probe
 - -done: status
- DTrace Arguments
 - query-start(arg0, arg1, arg2, arg3, arg4)
- Documenation
 - query-start(query, connectionid, database, user, host)
- Mapping args to probe information

```
mysql*:::query-start
{
  self->query = copyinstr(arg0);
  self->connid = arg1;
  self->db = copyinstr(arg2);
  self->who = strjoin(copyinstr(arg3),strjoin("@",copyinstr(arg4)));
  self->querystart = timestamp; self->qc = 0;
}
```



-start probe information

- Providing key information
 - user, host
 - connection-start
 - query-start
 - query-exec start
 - database, query
 - query-start
 - query-exec-start

What you can do with MySQL

Static probes

- Average duration of MySQL connections
- Monitor query execution time
- Determine effectiveness of query cache
- Monitor row operations within a query
- Monitor locking operations
- Monitor statement operations



More Info

- Brendan Gregg's website:
 - <http://blogs.sun.com/brendan/category/DTrace>
- OpenSolaris DTrace group:
 - <http://opensolaris.org/os/community/dtrace/dtracetoolkit/>
- MC Brown's Blog:
 - <http://coalface.mcslp.com>
- MySQL Docs:
 - <http://dev.mysql.com/doc/refman/6.0/en/dba-dtrace-server.html>