MySQL Server Memory Usage

By Nilnandan Joshi, Alexey Stroganov
and Roel Van de Paar

Copyright © 2006-2014 Percona LLC

Table of Contents

About Percona

Percona was founded in August 2006 by Peter Zaitsev and Vadim Tkachenko and now employs a global network of experts with a staff of more than 100 people. Our customer list is large and diverse, including Fortune 50 firms, popular websites, and small startups. We have over 1,800 customers and, although we do not reveal all of their names, chances are we're working with every large MySQL user you've heard about. To put Percona's MySQL expertise to work for you, please contact us.

Contact Us 24 Hours A Day

Is this an emergency? Get immediate assistance from Percona Support 24/7. Click here

Skype: oncall.percona
GTalk: oncall@percona.com
AIM (AOL Instant Messenger): oncallpercona
Telephone direct-to-engineer: +1-877-862-4316 or
UK Toll Free: +44-800-088-5561
Telephone to live operator: +1-888-488-8556
Customer portal: https://customers.percona.com/

Sales North America	(888) 316-9775 or (208) 473-2904
Sales Europe	+44-208-133-0309 (UK) 0-800-051-8984 (UK) 0-800-181-0665 (GER)
Training	(855) 55TRAIN or (925) 271-5054

Copyright © 2006-2014 Percona LLC

Chapter 1: MySQL server memory usage troubleshooting tips

By Nilnandan Joshi

As a **Percona support engineer**, I see many issues related to heavy server loads – OR **OOM killer** got invoked and killed MySQL server due to **high Memory usage…** OR a question such as: "I don't know why mysql is taking so much memory. How do I find where exactly memory is allocated? please help!"

There are many ways to check memory consumption of MySQL. So, I'm just trying here to explain it by combining all details that I know of in this post.

- **Check memory related Global/Session variables.**

If you are using **MyISAM** then you need to check for **Key_buffer_size**, while using **InnoDB**, you can check i**nnodb_buffer_pool_size, innodb_additional_memory_pool_size, innodb_log_buffer_size, innodb_sort_buffer_size (used only for sorting data while creating index in innodb, introduced from 5.6). max_connections, query_cache_size and table_cache** are also important variables to check**.**

We know that whenever a thread is connected to MySQL, it will need it's own buffers when they are doing some complex operations like FTS, sorting, creating temp tables etc. So we also need to check the size of **read_buffer_size, sort_buffer_size, read_rnd_buffer_size and tmp_table_size.**

There is a very good quote from High Performance MySQL, 3rd Edition: ***" You can think of MySQL's memory consumption as falling into two categories: the memory you can control, and the memory you can't. You can't control how much memory MySQL uses merely to run the server, parse queries, and manage its internals, but you have a lot of control over how much memory it uses for specific purposes."*** So it seems we have to understand the purpose for configuring any variable… either it is Global or Session level. I would like to explain more about that here.

For the Global variables like **key_buffer_size, query_cache_size etc,** MySQL always allocates and initializes the specified amount of memory all at once when the server starts. But it's not happened for those who are global default but can be set as per-session variables, i.e
For **read_buffer_size, sort_buffer_size, join_buffer_size**, MySQL doesn't allocate any memory for these buffers until query needs. But when a query needs, it immediately allocates the entire chunk of memory specified. So if there are even small sorts, full buffer size will be allocated which is just waste of memory. Even some buffers can be used multiple times. For example on queries that join several tables **join_buffer** can be allocated once per joined table. also some complicated queries including sub-queries can use multiple **sort_buffers** at the same time which can lead to

high memory consumption. In some scenario, query didn't even use sort_buffer whatever size is, as it select by primary key which will not allocate it. So it depends on the nature of your environment but I would say it's always better to start with a safe variable value that can be larger than default if needed but not as large as it can consume all of the server's memory.

One more thing, not **all** per thread memory allocation is configured by variables. Some of memory allocation per thread is done by MySQL itself for running complex processes/queries like **"stored procedures"** and it can take unlimited amount of memory while running. And sometimes, optimizer can also take a lot of memory working with highly complex queries which generally we can't control by any configuration parameter.

Even **innodb_buffer_pool_size** is not a hard limit, usually innodb uses **10%** more memory than the one specified. Many people do not recommend using both storage engine MyISAM and InnoDB at the same time on production server. Because both have individual buffers which can eat all server memory.

For detailed information related to this topic, I would suggest reading this post from Peter Zaitsev titled "MySQL Server Memory Usage."

- Check "**SHOW ENGINE INNODB STATUS**" for section "**BUFFER POOL AND MEMORY**"

```
1  ----------------------
2  BUFFER POOL AND MEMORY
3  ----------------------
4  Total memory allocated 137363456; in additional pool allocated 0
5  Dictionary memory allocated 76056
6  Buffer pool size 8191
7  Free buffers 7804
8  Database pages 387
9  Old database pages 0
10 Modified db pages 0
```

Above one is from Native MySQL but if you'll check the same with Percona Server you'll get some more information.

```
1  ----------------------
2  BUFFER POOL AND MEMORY
3  ----------------------
4
5  Total memory allocated 137756672; in additional pool allocated 0
6  Total memory allocated by read views 88
7  Internal hash tables (constant factor + variable factor)
8  Adaptive hash index 2217584 (2213368 + 4216)
9  Page hash 139112 (buffer pool 0 only)
10 Dictionary cache 597885 (554768 + 43117)
```

```
File system 83536 (82672 + 864)
Lock system 333248 (332872 + 376)
Recovery system 0 (0 + 0)
Dictionary memory allocated 43117
Buffer pool size 8191
Buffer pool size, bytes 134201344
Free buffers 7760
Database pages 431
Old database pages 0
Modified db pages 0
```

This will give you information regarding how much memory is allocated by InnoDB. You can see here **“Total Memory Allocated”, “Internal Hash Tables”, “Dictionary Memory Allocated”, “Buffer Pool Size”** etc.

- **Profiling MySQL Memory usage with Valgrind Massif**

Recently, I used this tool and surprisingly I got very good statistics about memory usage. Here the only problem is you have to shutdown the mysql, start it with valgrind massif and after collecting statistics, you again have to shutdown and normal start.

```
$ /etc/init.d/mysql stop
$ valgrind --tool=massif --massif-out-file=/tmp/massif.out /usr/sbin/mysqld
$ /etc/init.d/mysql restart
```

After getting massif.out file, you have to read it with ms_print command. You will see pretty nice graph and then statistics. i.e

```
[root@percona1 ~]# ms_print /tmp/massif.out
--------------------------------------------------------------------------------
Command:            /usr/sbin/mysqld
Massif arguments:   --massif-out-file=/tmp/massif.out
ms_print arguments: /tmp/massif.out
--------------------------------------------------------------------------------

    MB
50.22^         ##
     |         #
     |         #
     |         #
     |         #
     |         #
     |    :    #
     |    ::   #                      ::::::@:::::::::::::@:::@::::@:::@:::::
     |    : @::# :::::@@::::::::::::::::::: @::: ::: :::::@:::@::: @:::@:::::
     |    : @::# :  ::@ :::::::::: :: ::::: @::: ::: :::::@:::@::: @:::@:::::
     |    : @::# :  ::@ :::::::::: :: ::::: @::: ::: :::::@:::@::: @:::@:::::
     |    : @::# :  ::@ :::::::::: :: ::::: @::: ::: :::::@:::@::: @:::@:::::
     |    : @::# :  ::@ :::::::::: :: ::::: @::: ::: :::::@:::@::: @:::@::::@
     |    : @::# :  ::@ :::::::::: :: ::::: @::: ::: :::::@:::@::: @:::@::::@
     |   :: @::# :  ::@ :::::::::: :: ::::: @::: ::: :::::@:::@::: @:::@::::@
     |  ::: @::# :  ::@ :::::::::: :: ::::: @::: ::: :::::@:::@::: @:::@::::@
     | :::: @::# :  ::@ :::::::::: :: ::::: @::: ::: :::::@:::@::: @:::@::::@:
     | :::: @::# :  ::@ :::::::::: :: ::::: @::: ::: :::::@:::@::: @:::@::::@:
     | :::: @::# :  ::@ :::::::::: :: ::::: @::: ::: :::::@:::@::: @:::@::::@:
     | :::: @::# :  ::@ :::::::::: :: ::::: @::: ::: :::::@:::@::: @:::@::::@:
   0 +-------------------------------------------------------------------------->Mi
     0                                                                   575.9
```

```
Number of snapshots: 96
 Detailed snapshots: [1, 7, 11 (peak), 16, 35, 48, 58, 68, 78, 88]

--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0                0             0            0
  1      6,090,089          195,648          194,590         1,058            0
99.46% (194,590B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->48.79% (95,458B) 0x7A1D20: my_malloc (my_malloc.c:38)
| ->25.08% (49,060B) 0x6594F1: read_texts(char const*, char const*, char const***, unsigned int) (derror.cc:160)
| | ->25.08% (49,060B) 0x6597C2: init_errmessage() (derror.cc:69)
| |   ->25.08% (49,060B) 0x506232: init_common_variables() (mysqld.cc:3414)
| |     ->25.08% (49,060B) 0x508CBB: mysqld_main(int, char**) (mysqld.cc:4461)
| |       ->25.08% (49,060B) 0x5B2CD1B: (below main) (in /lib64/libc-2.12.so)
| |
| ->09.36% (18,317B) 0x789571: my_read_charset_file (charset.c:364)
| | ->09.36% (18,317B) 0x789DEC: init_available_charsets (charset.c:458)
| |   ->09.36% (18,317B) 0x4E35D31: pthread_once (in /lib64/libpthread-2.12.so)
| |     ->09.36% (18,317B) 0x789C80: get_charset_by_csname (charset.c:644)
| |       ->09.36% (18,317B) 0x5062E9: init_common_variables() (mysqld.cc:3439)
| |         ->09.36% (18,317B) 0x508CBB: mysqld_main(int, char**) (mysqld.cc:4461)
| |           ->09.36% (18,317B) 0x5B2CD1B: (below main) (in /lib64/libc-2.12.so)
| |
| ->08.37% (16,384B) 0x79DEEF: my_set_max_open_files (my_file.c:105)
| | ->08.37% (16,384B) 0x506169: init_common_variables() (mysqld.cc:3373)
| |   ->08.37% (16,384B) 0x508CBB: mysqld_main(int, char**) (mysqld.cc:4461)
| |     ->08.37% (16,384B) 0x5B2CD1B: (below main) (in /lib64/libc-2.12.so)
| |
| ->04.36% (8,536B) 0x788DB4: init_dynamic_array2 (array.c:70)
| | ->02.45% (4,800B) 0x5CD51A: add_status_vars(st_mysql_show_var*) (sql_show.cc:2062)
| | | ->02.45% (4,800B) 0x505E68: init_common_variables() (mysqld.cc:3245)
| | |   ->02.45% (4,800B) 0x508CBB: mysqld_main(int, char**) (mysqld.cc:4461)
| | |     ->02.45% (4,800B) 0x5B2CD1B: (below main) (in /lib64/libc-2.12.so)
| | |
```

You can see from the output that where memory is allocated, to which function etc. You can use this tool to find memory leaks. You can get more information here for how to install and use it. Here's another related post by Roel Van de Paar titled: "Profiling MySQL Memory Usage With Valgrind Massif."

> If possible Valgrind massif should not be used on busy production server as it can degrade the performance. Generally it's used to find memory leak by creating mirror environment on test/stage server and run on it. It needs debug binary to run so it decreases performance a lot. So it can be used for investigating some cases but not for regular use.

- **Check Plot memory usage by monitoring ps output.**

This also useful when you want to check how much virtual(VSZ) and real memory (RSS) is used by mysqld. You can either simply run some bash script for monitoring it like

```
while true
do
  date >> ps.log
  ps aux | grep mysqld >> ps.log
  sleep 60
done
```

Or you can also check when needed from shell prompt with “ps aux | grep mysqld” command.

- **Memory tables in MySQL 5.7**

With MySQL 5.7, some very interesting memory statistics tables are introduced to check memory usage in **performance_schema.** There is no any detailed documentation available yet but you can check some details here.

http://dev.mysql.com/doc/refman/5.7/en/memory-summary-tables.html

In P_S, there are five memory summary tables.

```
mysql> show tables like '%memory%';
+----------------------------------------+
| Tables_in_performance_schema (%memory%) |
+----------------------------------------+
| memory_summary_by_account_by_event_name |
| memory_summary_by_host_by_event_name |
| memory_summary_by_thread_by_event_name |
| memory_summary_by_user_by_event_name |
| memory_summary_global_by_event_name |
+----------------------------------------+
5 rows in set (0.00 sec)
```

So with every event you can get summarized memory consumption for a particular account, host, thread and user. While checking more, I found that there are around 209 different events to check. I have just tried to check one event related to join buffer size.

```
mysql> select * from memory_summary_by_account_by_event_name where SUM_NUMBER_OF_BYTES_ALLOC
<> 0 and user = 'msandbox' and event_name = 'memory/sql/JOIN_CACHE' \G

*************************** 1. row ***************************
USER: msandbox
HOST: localhost
EVENT_NAME: memory/sql/JOIN_CACHE
COUNT_ALLOC: 2
COUNT_FREE: 2
SUM_NUMBER_OF_BYTES_ALLOC: 524288
SUM_NUMBER_OF_BYTES_FREE: 524288
LOW_COUNT_USED: 0
CURRENT_COUNT_USED: 0
HIGH_COUNT_USED: 1
LOW_NUMBER_OF_BYTES_USED: 0
CURRENT_NUMBER_OF_BYTES_USED: 0
HIGH_NUMBER_OF_BYTES_USED: 262144
1 row in set (0.00 sec)

mysql> show global variables like 'join%';
+------------------+--------+
| Variable_name | Value |
+------------------+--------+
| join_buffer_size | 262144 |
+------------------+--------+
1 row in set (0.00 sec)

mysql>
```

Here, **COUNT_ALLOC, COUNT_FREE** are aggregate the number of calls to malloc-like and free-like functions. **SUM_NUMBER_OF_BYTES_ALLOC and SUM_NUMBER_OF_BYTES_FREE** are indicate the aggregate size of allocated and freed memory blocks. **CURRENT_COUNT_USED** is the aggregate number of currently allocated blocks that have not been freed yet. **CURRENT_NUMBER_OF_BYTES_USED** is the aggregate size of currently allocated memory blocks that have not been freed yet. LOW_ and HIGH_ are low and high water marks corresponding to the columns.

- **Memory sections in output of** pt-summary **and** pt-mysql-summary

If you are aware about these scripts then, these are giving very good summary about overall server memory consumption as well as related to MySQL.

Like in output of pt-summary,

```
# Memory ################################################
Total | 11.8G
Free | 143.7M
Used | physical = 11.6G, swap allocated = 4.0G, swap used = 0.0, virtual = 11.6G
Buffers | 224.9M
Caches | 6.2G
Dirty | 164 kB
UsedRSS | 4.8G
```

In output of pt-mysql-summary.

```
# Query cache #################################################
query_cache_type | OFF
Size | 0.0
Usage | 0%
HitToInsertRatio | 0%

# InnoDB ######################################################
Version | 5.5.30-rel30.2
Buffer Pool Size | 4.0G
Buffer Pool Fill | 35%
Buffer Pool Dirty | 1%

# MyISAM ######################################################
Key Cache | 32.0M
Pct Used | 20%
Unflushed | 0%
```

Conclusion:

It is really important for us to know where MySQL allocates memory and how it affects the overall load on the MySQL server and performance. I have just tried here to describe a few ways but I still think that we should have some sort of script or something that can combine all of these results and gives us some truthful output of memory usage in MySQL.

The impact of memory allocators on MySQL performance

By Alexey Stroganov

MySQL server intensively uses dynamic memory allocation so a good choice of memory allocator is quite important for the proper utilization of CPU/RAM resources. Efficient memory allocator should help to improve scalability, increase throughput and keep memory footprint under the control. In this post I'm going to check impact of several memory allocators on the performance/scalability of MySQL server in the read-only workloads.

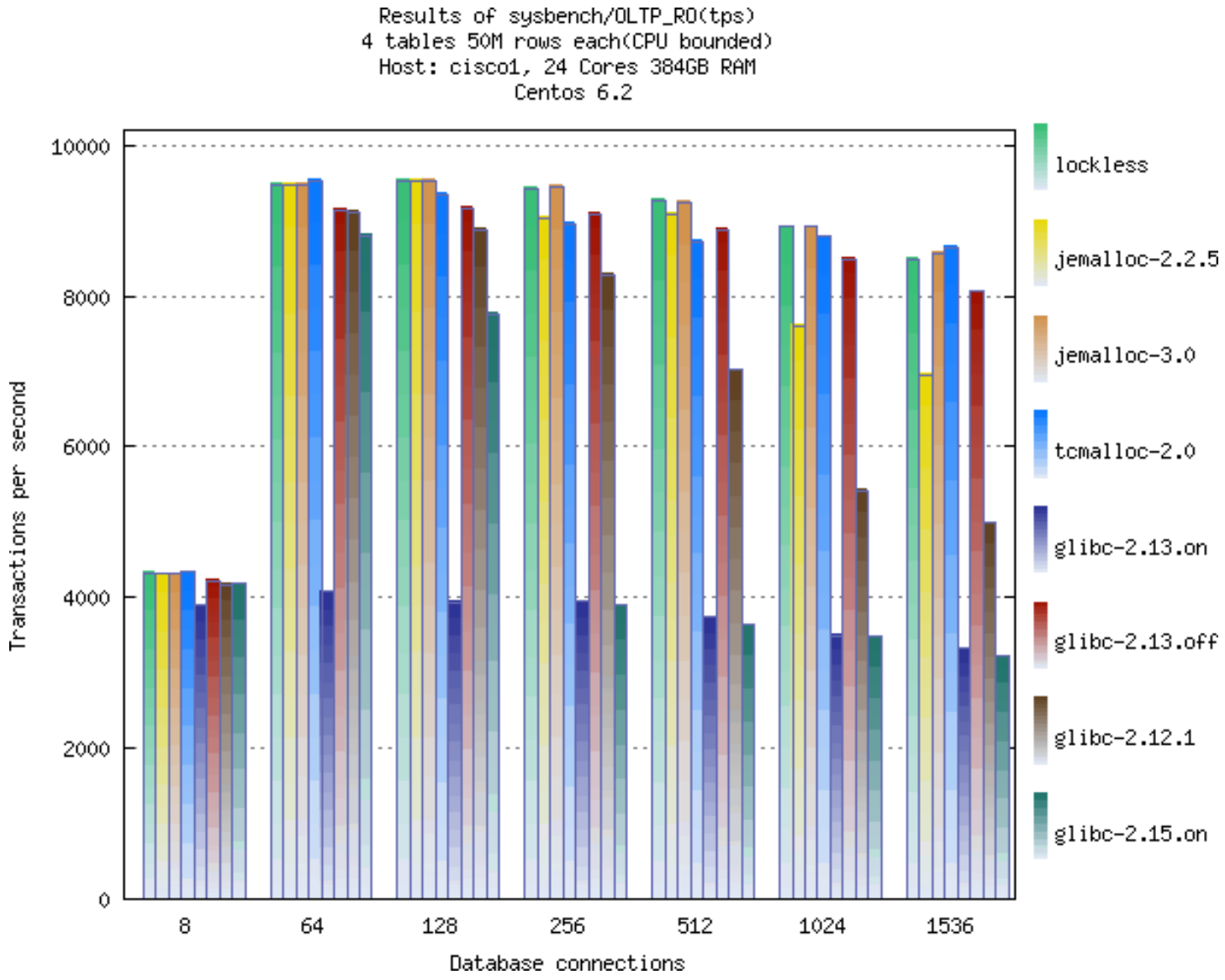
For my testing i chose following allocators: lockless, jemalloc-2.2.5, jemalloc-3.0, tcmalloc(gperftools-2.0), glibc-2.12.1(new malloc)(CentOS 6.2), glibc-2.13(old malloc), glibc-2.13(new malloc), glibc-2.15(new malloc).

Let me clarify a bit about malloc in glibc. Starting from glibc-2.10 it had two malloc implementations that one can choose with configure option *–enable-experimental-malloc*. (You can find details about new malloc here). Many distros switched to this new malloc in 2009. From my experince this new malloc behaved not always efficiently with MySQL so i decided to include old one to comparison as well. I used glibc-2.13 for that purpose because later *–enable-experimental-malloc* option was removed from glibc sources.

I built all allocators from sources(except system glibc 2.12.1) with stock CentOS gcc(version 4.4.6 20110731). All of them were built with -O3. I used LD_PRELOAD for lockless, jemalloc-2.2.5, jemalloc-3.0, tcmalloc and for glibc I prefixed mysqld with:

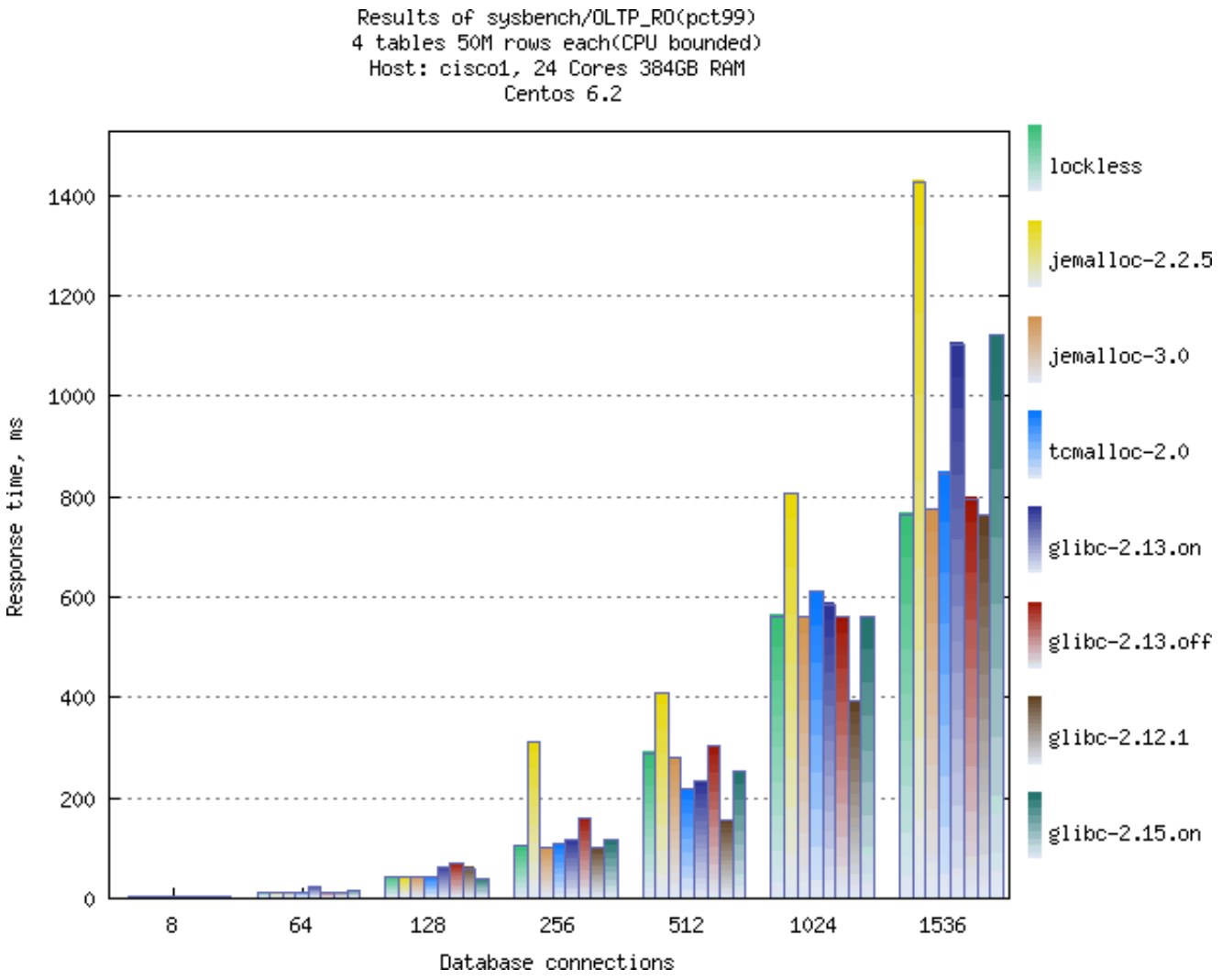
```
/[path]/glibc-[X].root/lib/ld-[X].so --library-path /[path]/glibc-[X].root/lib:/lib64:/usr/lib64
```

- Testing details:
  - Cisco USC_C250 box
  - Percona Server 5.5.24
  - 2 read only scnearios: OLTP_RO and POINT_SELECT from the latest sysbench-0.5
  - dataset consists of 4 sysbench tables(50M rows each) ~50G data / CPU bound case
  - innodb_buffer_pool_size=52G
- For every malloc allocator perform the following steps:
  - start Percona server either with LD_PRELOAD=[allocator_lib.so] or glibc prefix(see above)/get RSS/VSZ size of mysqld
  - warmup with 'select avg(id) from sbtest$i FORCE KEY (PRIMARY)' and then OLTP_RO for 600sec
    run OLTP_RO/POINT_SELECT test cases, duration 300 sec and vary number of threads: 8/64/128/256/512/1024/1536
  - stop server/get RSS/VSZ size of mysqld

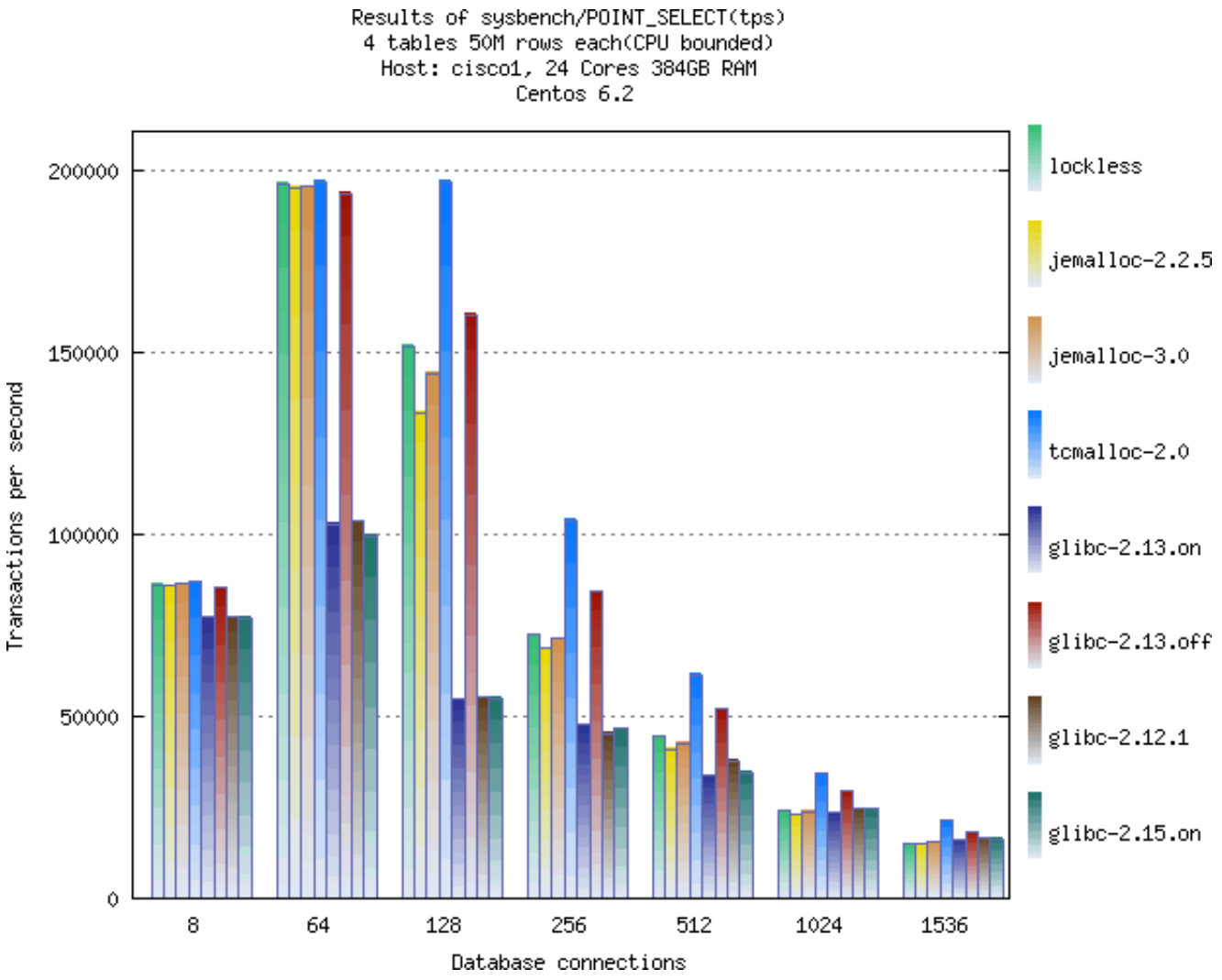


The best throughput/scalability we have with lockless/jemalloc-3.0/tcmalloc. jemalloc-2.2.5 slightly drops with higher number of threads. On the graph with response time(see below) there are spikes for it that may be caused by some contention in the lib.

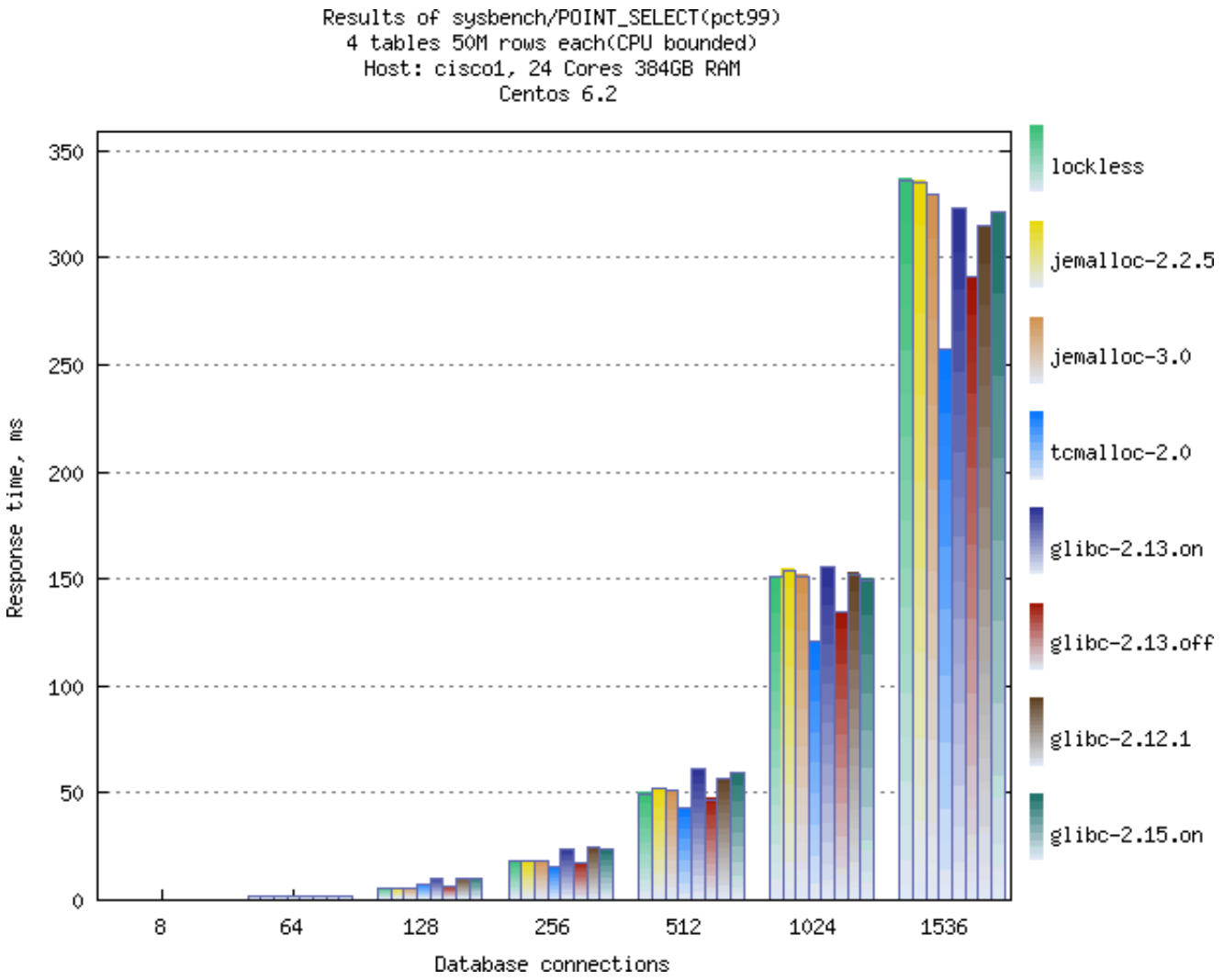
All variations of glibc that are based on new
malloc with increasing concurrency demonstrate notable drops – almost two times at high threads. In the same time glibc-2.13 built with old malloc looks good, results are very similar to lockless/jemalloc-3.0/tcmalloc.



(Please procede to the next page)



For POINT_SELECT test with increasing concurrency we have two allocators that handle load very well – tcmalloc and only slightly behind … glibc-2.13 with old malloc. Then we have jemalloc-3.0/lockless/jemalloc-2.2.5 and last ones are glibc allocators based on new malloc. Along with the best throughput/scalability runs with tcmalloc also demonstrate best response time (30-50 ms at the high threads).



Besides throughput and latency there is one more factor that should be taken into account – memory footprint.

memory allocator	mysqld RSS size grow(kbytes)	mysqld VSZ size grow(kbytes)
lockless	6.966.736	105.780.880
jemalloc-2.2.5	214.408	3.706.880
jemalloc-3.0	216.084	5.804.032
tcmalloc	456.028	514.544
glibc-2.13-new-malloc	210.120	232.624
glibc-2.13-old-malloc	253.568	1.006.204
glibc-2.12.1-system	162.952	215.064
glibc-2.15-new-malloc	5.106.124	261.636

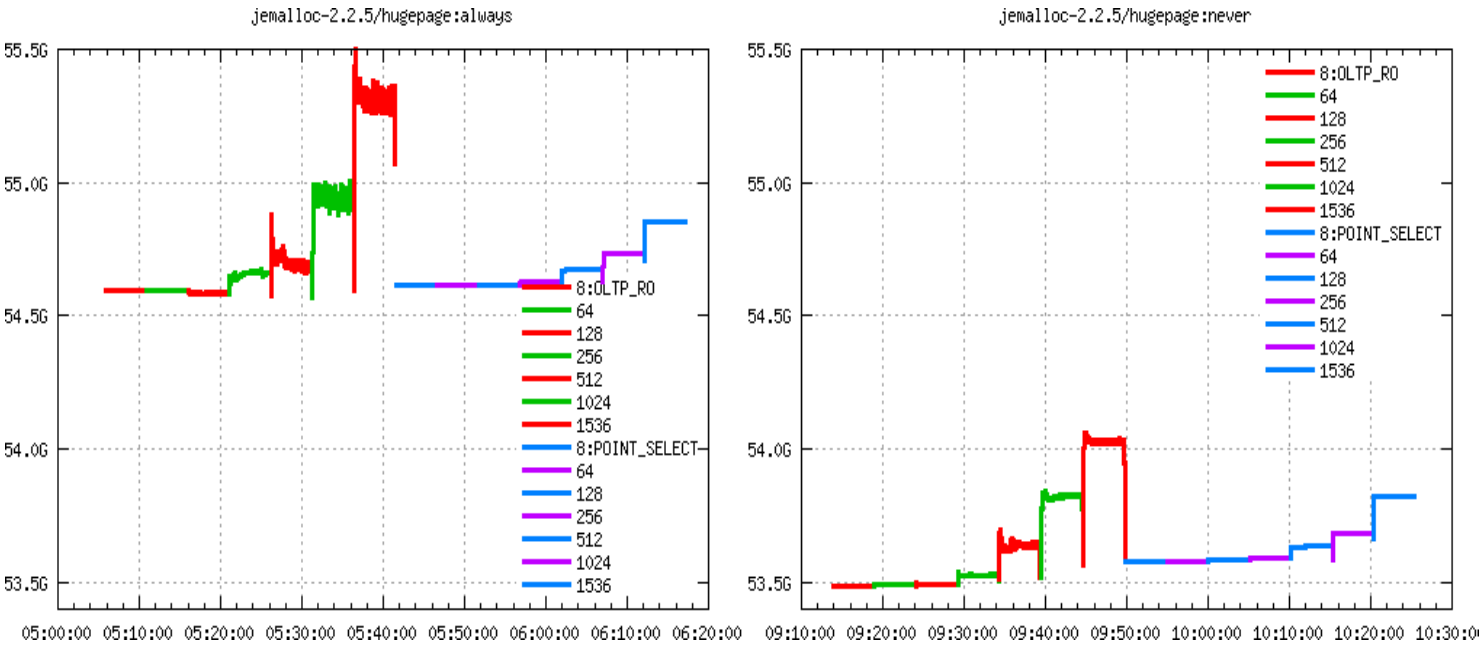
The only two allocators lockless and glibc-2.15-with new malloc notably incressed RSS memory footprint of mysqld server – more than on 5G. Memory usage for others allocators looks more or less acceptable.

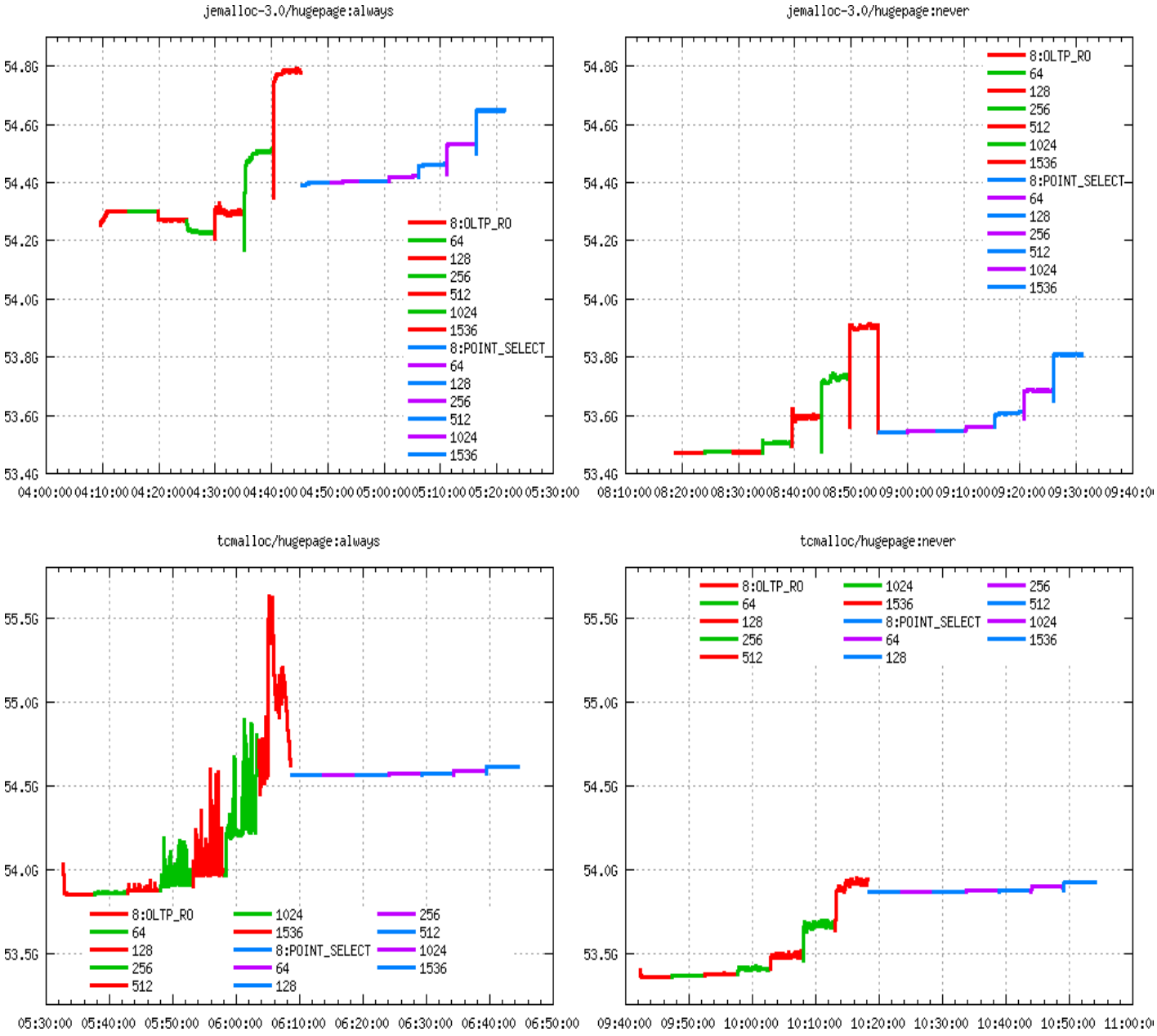
Taking into account all 3 factors – throughput, latency and memory usage for above POINT_SELECT/OLTP_RO type of workloads the most suitable allocators are tcmalloc, jemalloc-3.0 and glibc-2.13 with old malloc.

Important point to take is that new glibc with new malloc implementation may be NOT suitable and may show worse results than on older platforms.

UPDATE:

To cover some questions raised in the comments I rerun OLTP_RO/POINT_SELECT tests with jemalloc-2.2.5/jemalloc-3.0/tcmalloc, varied /sys/kernel/mm/transparent_hugepage/enabled(always|never) and gathered mysqld size with ‘ps –sort=-rss -eopid,rss,vsz,pcpu’ during the test run. Just to remind whole test run cycle looks like following:
start server, warmup, OLTP_RO test, POINT_SELECT test. So on charts below you will see how mysqld footprint is changed during the test cycle and what is the impact of disabling of hugepages.




jemalloc-3.0/hugepage:always
54.8G
54.6G
54.4G
54.2G
54.0G
53.8G
53.6G
53.4G
04:00:00 04:10:00 04:20:00 04:30:00 04:40:00 04:50:00 05:00:00 05:10:00 05:20:00 05:30:00
8:OLTP_RO
64
128
256
512
1024
1536
8:POINT_SELECT
64
128
256
512
1024
1536
jemalloc-3.0/hugepage:never
54.8G
54.6G
54.4G
54.2G
54.0G
53.8G
53.6G
53.4G
08:10:00 08:20:00 08:30:00 08:40:00 08:50:00 09:00:00 09:10:00 09:20:00 09:30:00 09:40:00
8:OLTP_RO
64
128
256
512
1024
1536
8:POINT_SELECT
64
128
256
512
1024
1536
tcmalloc/hugepage:always
55.5G
55.0G
54.5G
54.0G
53.5G
05:30:00 05:40:00 05:50:00 06:00:00 06:10:00 06:20:00 06:30:00 06:40:00 06:50:00
8:OLTP_RO
64
128
256
512
1024
1536
8:POINT_SELECT
64
128
256
512
1024
1536
tcmalloc/hugepage:never
55.5G
55.0G
54.5G
54.0G
53.5G
09:40:00 09:50:00 10:00:00 10:10:00 10:20:00 10:30:00 10:40:00 10:50:00 11:00:00
8:OLTP_RO
64
128
256
512
1024
1536
8:POINT_SELECT
64
128
256
512
1024
1536

MySQL performance: Impact of memory allocators (Part 2)

By Alexey Stroganov

In the previous chapter I wrote about memory allocators and how they can affect MySQL performance in general. This time I would like to explore this topic from a bit different angle: What impact does the number of processor cores have on different memory allocators and what difference we will see in MySQL performance in this scenario?

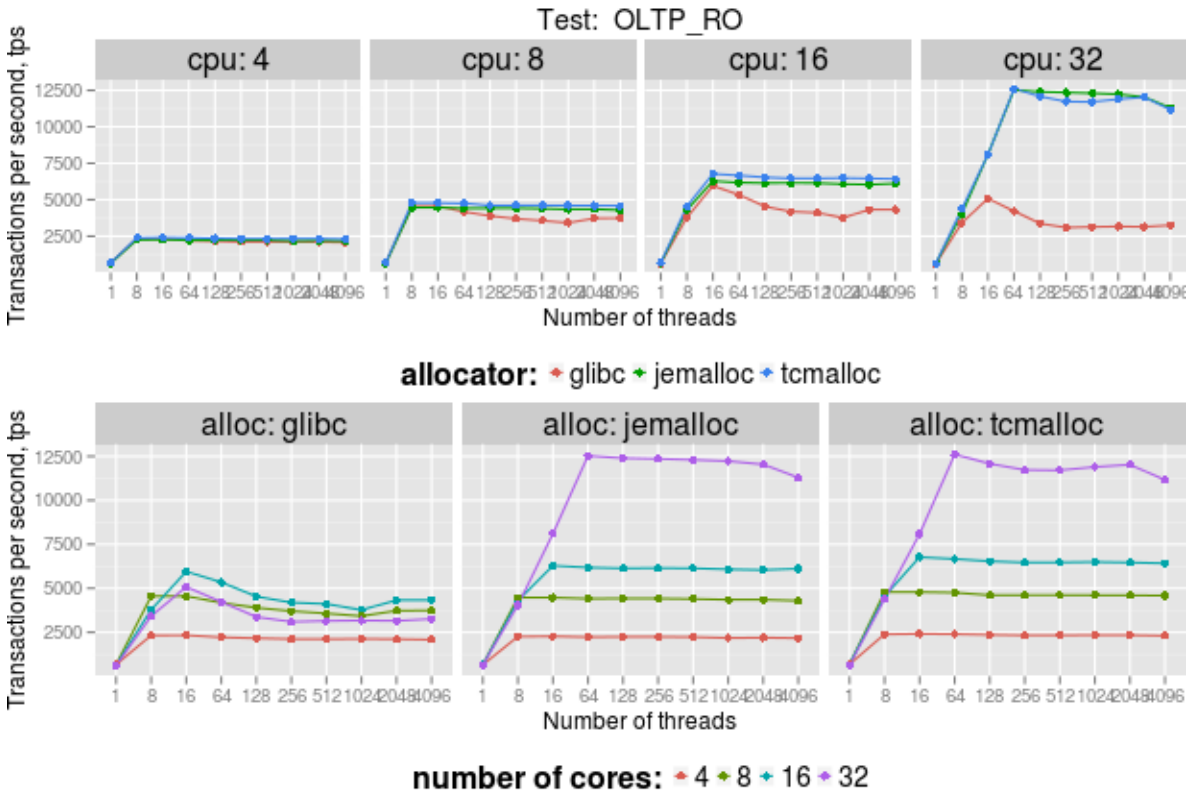
Let me share a conclusion first: **If you have a server with more than 8 cores you should use something different than the default glibc memory allocator.**
We recommend jemalloc or tcmalloc.

In my test I will use Dell R720 box(spec), Centos 6.3, upcoming Percona Server 5.5.30 and 3 allocators – stock glibc 2.13, jemalloc-3.1.0, the latest tcmalloc from svn repo. Regarding my selection of jemalloc version see my notes at the end of this post.

Test box has 2xIntel E5/2.2Ghz with 8 real cores per socket – 16 real cores + enabled hyper-threading gives us total – 32 vcpu. In my tests I didn't see any notable difference between allocators up to 4 vcpu, so on charts below I will highlight results from 4 to 32 vcpu.

As test workload I will use the same 2 sysbench tests – OLTP_RO and POINT_SELECT that I used before. Sysbench dataset – 16 tables, each 5M rows, uniform distribution.

OLTP_RO test consists of 5 select queries – select_ranges, select_order_ranges, select_distinct_ranges, select_sum_ranges, point_select. Processing these queries will involve notable amount of malloc()/free() operations, so allocator efficiency is the key factor to achieve high throughput in this test.

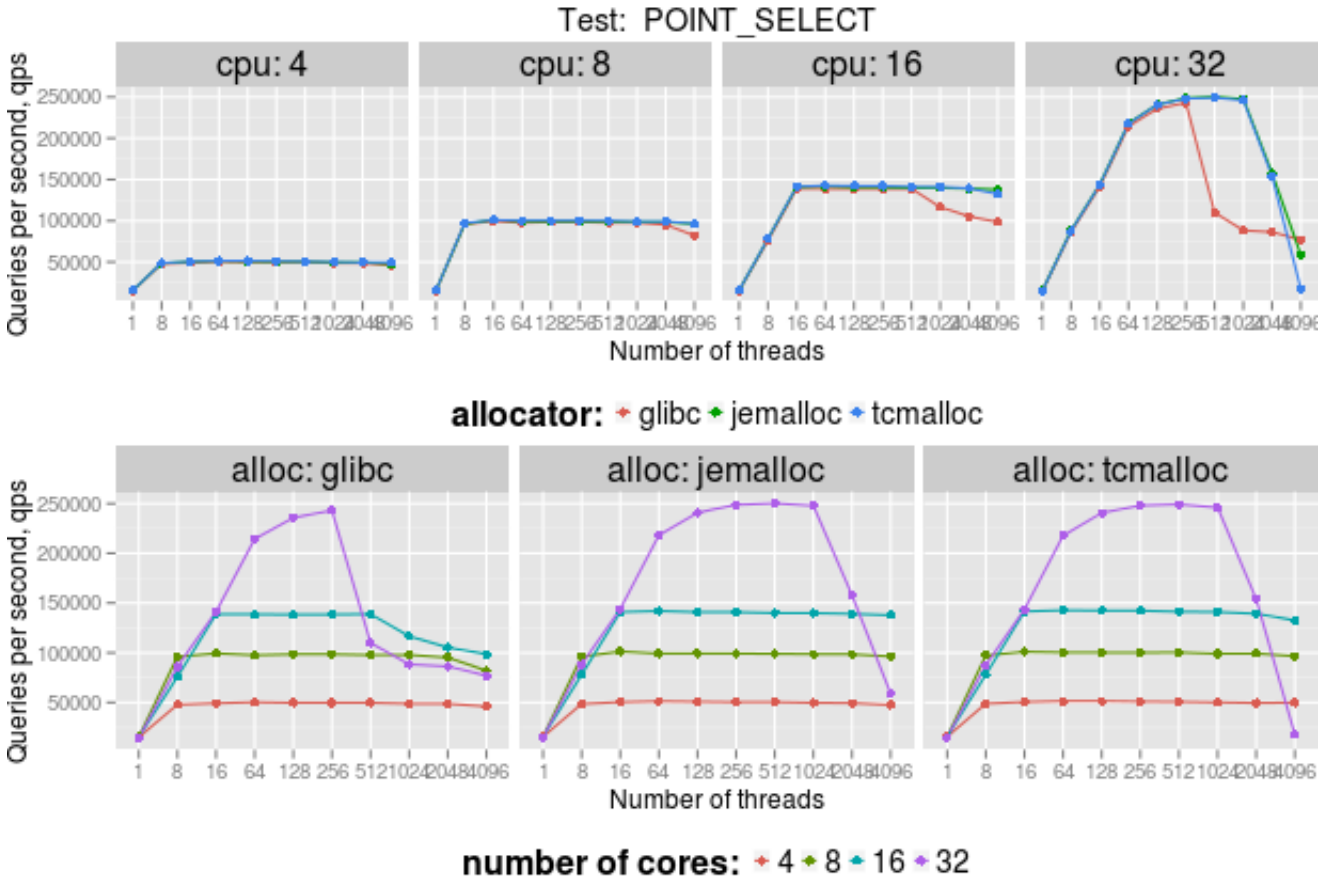


Observations:

- 4 vcpu – results are almost identical for all allocators (~2500tps)
- 8 vcpu – **results doubled (~5000tps) for jemalloc and tcmalloc**, but with glibc malloc we have a drop at 64/128 threads to ~3500tps
- 16vcpu – **increase in throughput and quite stable results for jemalloc and tcmalloc** up to 4096 threads (~6300tps) and again drop after 16 threads for glibc to ~4000tps
- 32vcpu – throughput for jemalloc and tcmalloc jumped to ~12500tps, results stay at this level up to 1024 threads and then tps slightly decreased but still looks ok. For glibc tps drops below results we have observed for 8/16 vcpu(~3100tps).

So the difference in OLTP_RO test between glibc and jemalloc/tcmalloc in case of 32vcpu is ~4x.

POINT_SELECT – very simple query – SELECT c FROM sbtest WHERE id=N. Test workload with this query
allows to generate significant load and check server behavior under very high pressure



Observations:

- 4 vcpu – again **no difference** between allocators (~50,000qps)
- 8 vcpu – with all allocators we got ~100,000qps. Results for jemalloc/tcmalloc are stable up to 4096 threads, for glibc malloc there is decrease in qps for 2048/4096 threads to ~80.000qps.
- 16vcpu – with all allocators we got ~140,000qps. For jemalloc/tcmalloc up to 4096 threads, for glibc up to 512 threads, then decrease in throughput to 100,000qps.
- 32vcpu – with all allocators we got up to ~240,000qps. Then for every allocator we have drop in throughput but at different point and to different level.
  - for glibc malloc drop happened after 256 threads, qps is below the level for 8/16 vcpu. (~80,000qps).
  - for tcmalloc drop happened after 1024 threads, at 2048 thread qps is very close to results for 16vcpu and at 4096 threads qps is ~17,000.
  - for jemalloc drop happened after 1024 threads as well, at 2048 thread qps is very close to results for 16vcpu and at 4096 threads – qps is slightly better than results for 4vcpu (~60,000qps).As you can see in the case of the very high concurrency and notable amount of the small/medium allocations, we have quite poor results for jemalloc/tcmalloc. Even worse than for glibc. This is the very specific case when overhead from the advanced techniques used in these allocators that should help to speed up allocation,purging of the dirty pages, minimize impact of the memory fragmentation is so significant that becomes bottleneck for the query processing. I believe that both allocators can be tuned to handle such cases better – for instance allocate more arenas but that may notably increase memory footprint.

Conclusion:
- if your box has 8 cores or less – there is almost no difference between glibc malloc and alternative allocators
- if your box has more than 8 cores – **you should try/evaluate alternative allocators; it can notably boost your MySQL server at no cost**. Also, an alternative allocator must be used if you run benchmarks in this configuration, otherwise the performance will be limited by glibc/malloc and not by MySQL.

Notes regarding jemalloc version I've used in my tests: I've noted notable impact on MySQL performance after version 3.2.0 (see raw results below) so I used jemalloc-3.1.0 in my tests. I suppose that some changes in 3.2.0 like for instance changes re: page run allocation and dirty page purging may have some correlation with decreasing performance in workloads with MySQL.

```
# Test: POINT_SELECT:throughput, QPS
#
# Set 1 - 5.5.30pre-jemalloc-3.0.0
# Set 2 - 5.5.30pre-jemalloc-3.1.0
# Set 3 - 5.5.30pre-jemalloc-3.2.0
# Set 4 - 5.5.30pre-jemalloc-3.3.0
#
# Threads         Set 1     Set 2     Set 3     Set 4
        1024  236575.74 236862.59 211203.42 215098.20
        2048  154829.26 154348.16 135607.69 137162.29
```

Profiling MySQL memory usage with Valgrind Massif

By Roel Van de Paar

There are times where you need to know **exactly** how much memory the mysqld server (or any other program) is using, where (i.e. for what function) it was allocated, how it got there (a backtrace, please!), and at what point in time the allocation happened.

For example; you may have noticed a sharp memory increase after executing a particular query. Or, maybe mysqld is seemingly using too much memory overall. Or again, maybe you noticed mysqld's memory profile slowly growing overtime, indicating a possible memory bug.

Whatever the reason, there is a simple but powerful way to profile MySQL memory usage; the Massif tool from Valgrind. An excerpt from the Massif manual page (Heap memory being simply the allotted pool of memory for use by programs):

Massif tells you not only how much heap memory your program is using, it also gives very detailed information that indicates which parts of your program are responsible for allocating the heap memory.

Firstly, we need to get the Valgrind program. Though you could use the latest version which comes with your OS (think yum or apt-get install Valgrind), I prefer to obtain & compile the latest release (3.8.1 at the moment):

```
sudo yum remove valgrind*      # or apt-get etc.
sudo yum install wget make gcc gcc-c++ libtool libaio-devel bzip2 glibc*
wget http://valgrind.org/downloads/valgrind-3.8.1.tar.bz2      # Or newer
tar -xf valgrind-3.8.1.tar.bz2
cd valgrind-3.8.1
./configure
make
sudo make install
valgrind --version             # check version to be same as what was downloaded (3.8.1 here)
```

There are several advantages to self-compiling:

1. When using the latest version of Valgrind, even compiled 'out of the box' (i.e. with no changes), you will likely see less issues then with earlier versions. For example, earlier versions may have too-small Valgrind-internal memory tracking allocations hardcoded. In other words; you may not be able to run your huge-buffer-pool under Valgrind without it complaining quickly.
2. If you self compile, and those Valgrind-internal limits are still too small, you can easily change them before compiling. An often bumped up setting is VG_N_SEGMENTS in coregrind/m_aspacemgr/aspacemgr-linux.c (when you see 'Valgrind: FATAL: VG_N_SEGMENTS is too low')
3. Newer releases [better] support newer hardware and software.

Once 'valgrind –version' returns the correct installed version, you're ready to go. In this example, we'll write the output to /tmp/massif.out. If you prefer to use another location (and are therefore bound to set proper file rights etc.) use:

```
$ touch /your_location/massif.out
$ chown user:group /your_location/massif.out    # Use the user mysqld will now run under
```

Now, before you run mysqld under Valgrind, make sure debug symbols are present. Debug symbols are present when the binary is not stripped of them (downloaded 'GA' [generally available] packages may contain optimized or stripped binaries, which are optimized for speed rather than debugging). If the binaries you have are stripped, you have a few options to get a debug build of mysqld to use for memory profiling purposes:

- Download the appropriate debuginfo packages (these may not be available for all releases).
- Download debug binaries of the same server version as you are currently using, and simply use the debug mysqld as a drop-in replacement for your current mysqld (i.e. shutdown, mv mysqld mysqld.old, cp /debug_bin_path/mysqld ./mysqld, startup).
- If you have (through download or from past storage) the source code available (of the same server version as you are currently using) then simply debug-compile the source and use the mysqld binary as a drop-in replacement as shown in the last point. (For example, Percona Server 5.5 source can be debug-compiled by using './build/build-binary –debug ..').

Valgrind Massif needs the debug symbol information to be present, so that it can print stack traces that show where memory is consumed. Without debug symbols available, you would not be able to see the actual function call responsible for memory usage. If you're not sure if you have stripped binaries, simply test the procedure below and see what output you get.

Once you're all set with debug symbols, shutdown your mysqld server using your standard shutdown procedure, and then re-start it manually under Valgrind using the Massif tool:

```
$ valgrind --tool=massif --massif-out-file=/tmp/massif.out /path/to/mysqld {mysqld options...}
```

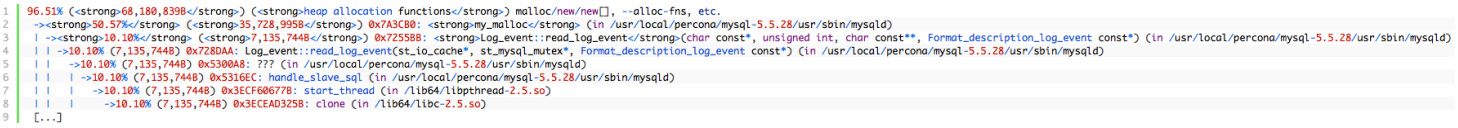
Note that '{mysqld options}' could for instance include –default-file=/etc/my.cnf (if this is where your my.cnf file is located) in order to point mysqld to your settings file etc. After mysqld is properly started (check if you can login with your mysql client), you would execute whatever steps you think are necessary to increase memory usage/trigger the memory problem. You could also just leave the server running for some time (for example, if you have experienced memory increase over time).

Once you've done that, shutdown mysqld (again using your normal shutdown procedure), and then use the ms_print tool on the masif.out file to output a textual graph of memory usage:

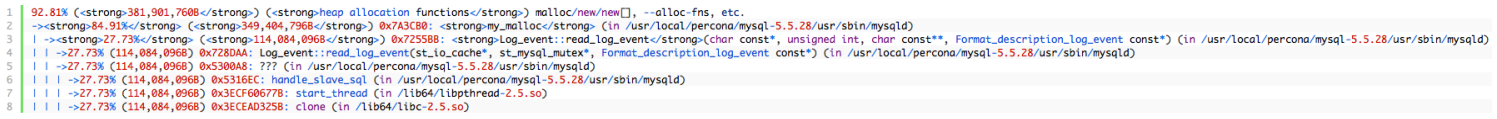
```
ms_print /tmp/massif.out
```

An partial example output from a recent customer problem we worked on:

Click images for enlarged view



And, a few snapshots later:



As you can see, a fair amount of (and in this case 'too much') memory is being allocated to the Log_event::read_log_event function. You can also see the memory allocated to the function grow significantly accross the snapshots. This example helped to pin down a memory leak bug on a filtered slave (read more in the actual bug report).

Besides running Valgrind Massif in the way above, you can also change Massif's snapshot options and other cmd line options to match the snapshot frequency etc. to your specific requirements. However, you'll likely find that the default options will perform well in most scenario's.

For the technically advanced, you can take things one step further: use Valgrind's gdbserver to obtain Massif snapshots on demand (i.e. you can command-line initiate Massif snapshots just before, during and after executing any commands which may alter memory usage significantly).

Conclusion: using Valgrind Massif, and potentially Valgrind's gdbserver (which was not used in the resolution of the example bug discussed), will help you to analyze the ins and outs of mysqld's (or any other programs) memory usage.

Credits: Staff @ a Percona customer, Ovais, Laurynas, Sergei, George, Vladislav, Raghavendra, Ignacio, myself & others at Percona all combined efforts leading to the information you can read above.

PERCONA

About the authors

Nilnandan Joshi officially started with Percona as a Support Engineer. Before joining Percona, he has worked as a MySQL Database administrator with different types of service based companies managing high-traffic websites and web applications. Nilnandan has extensive experience in database design and development, database management, client management, security/documentations/training, implementing DRM solutions, automating backups and high availability.

Alexey Stroganov is a Performance Engineer at Percona, where he works on improvements and features that makes Percona Server even more flexible, faster and scalable. Before joining Percona he worked on the performance testings/analysis of MySQL server and it components at MySQL AB/Sun/ Oracle for more than ten years. During this time he was focused on performance evaluations, benchmarks, analysis, profiling, various optimizations and tunings.

Roel Van de Paar leads Percona's Development Services group, which includes Percona's QA, documentation, software packaging, build/release management and performance testing teams. Roel has an extensive and varied background in IT, backed up by many industry leading certifications. Previously Roel was a Senior QA Engineer at Oracle, contributing significantly to the MySQL QA infrastructure. Prior to this he was a Senior Support Engineer at Oracle/Sun, supporting enterprise scale MySQL deployments across a variety of platforms.

Copyright © 2006-2014 Percona LLC