GOPHER CHINA 2020

中国 上海 / 2020-11.21-22

Go runtime related problems in TiDB production environment

About me

- Arthur Mao(毛康力), Senior Engineer@PingCAP
- TiDB core developer (top3 contributor)
- GitBook about golang internals (@tiancaiamao)
- Gopher since 2012 ...

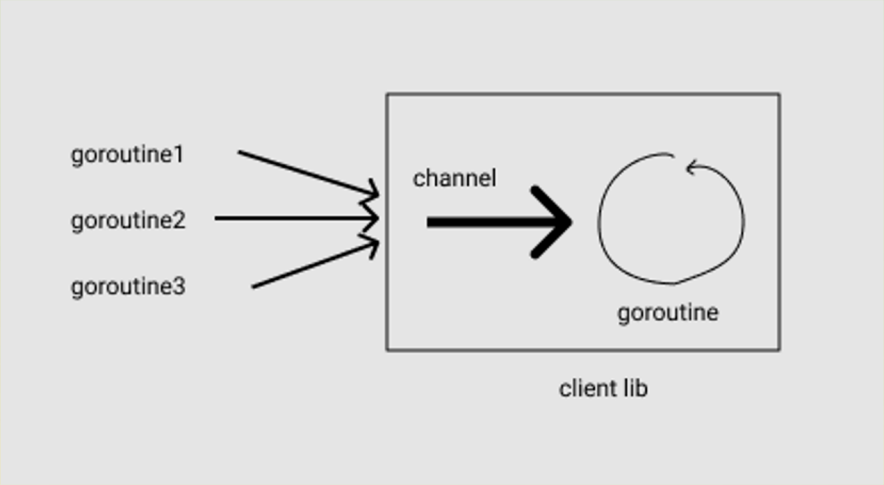
Agenda

- Latency in scheduler
  - case study: batching client requests
- Memory control
  - case study: transparent huge pages
- GC Related
  - case study: GC sweep caused latency jitter
  - case study: Lock and NUMA aware

Part I - Latency in scheduler

Background

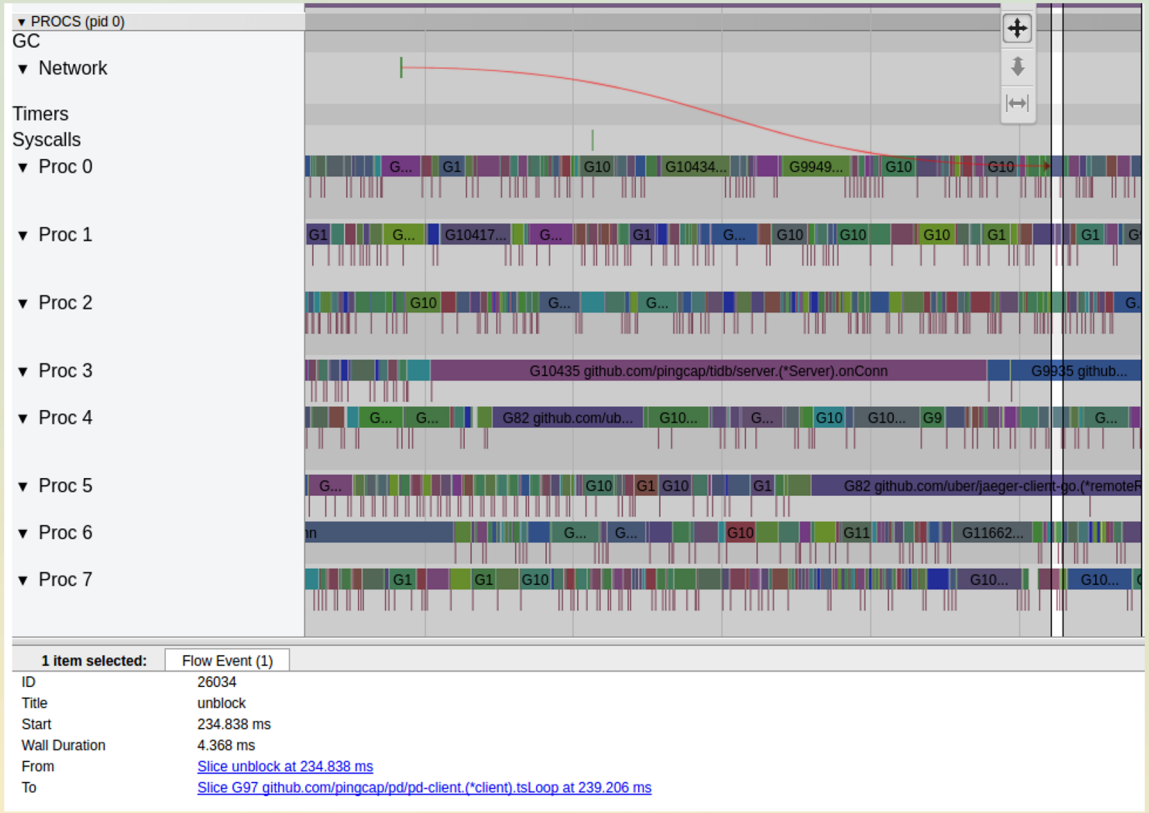
- The client consists of a goroutine and a channel
  - The channel batch the request
  - A goroutine run read-send-recv loop



Description

- When the machine load and CPU usage is high …
- The network round-trip is reasonable
- But the RPC metrics show a high latency

Investigation

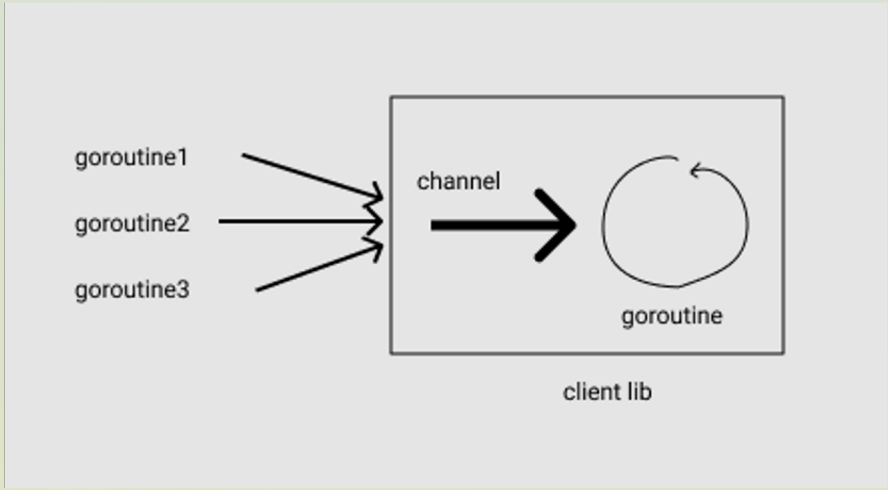


Analysis

- Network IO is ready => goroutine wake up == 4.3ms
  - Sometime even 10ms+ latency here!
  - The time spend on runtime schedule is not negligible
- When CPU is overload, which goroutine should be given priority?

Analysis

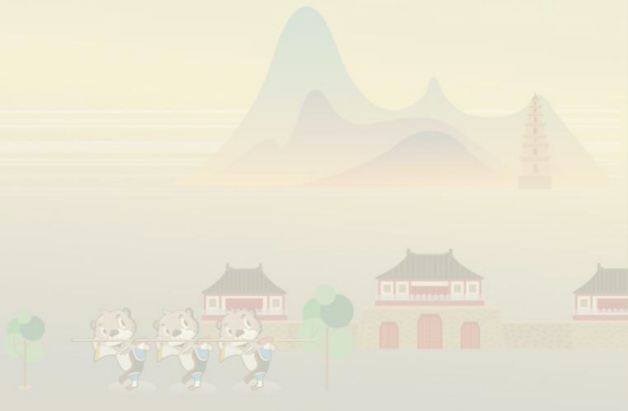
- The goroutine is special, it block all the callers
- The scheduler treat them equally



Conclusion

- Under heavy workload, goroutines get longer to be scheduled
- The runtime scheduling does not consider priority
- CPU dense workload could affect IO latency

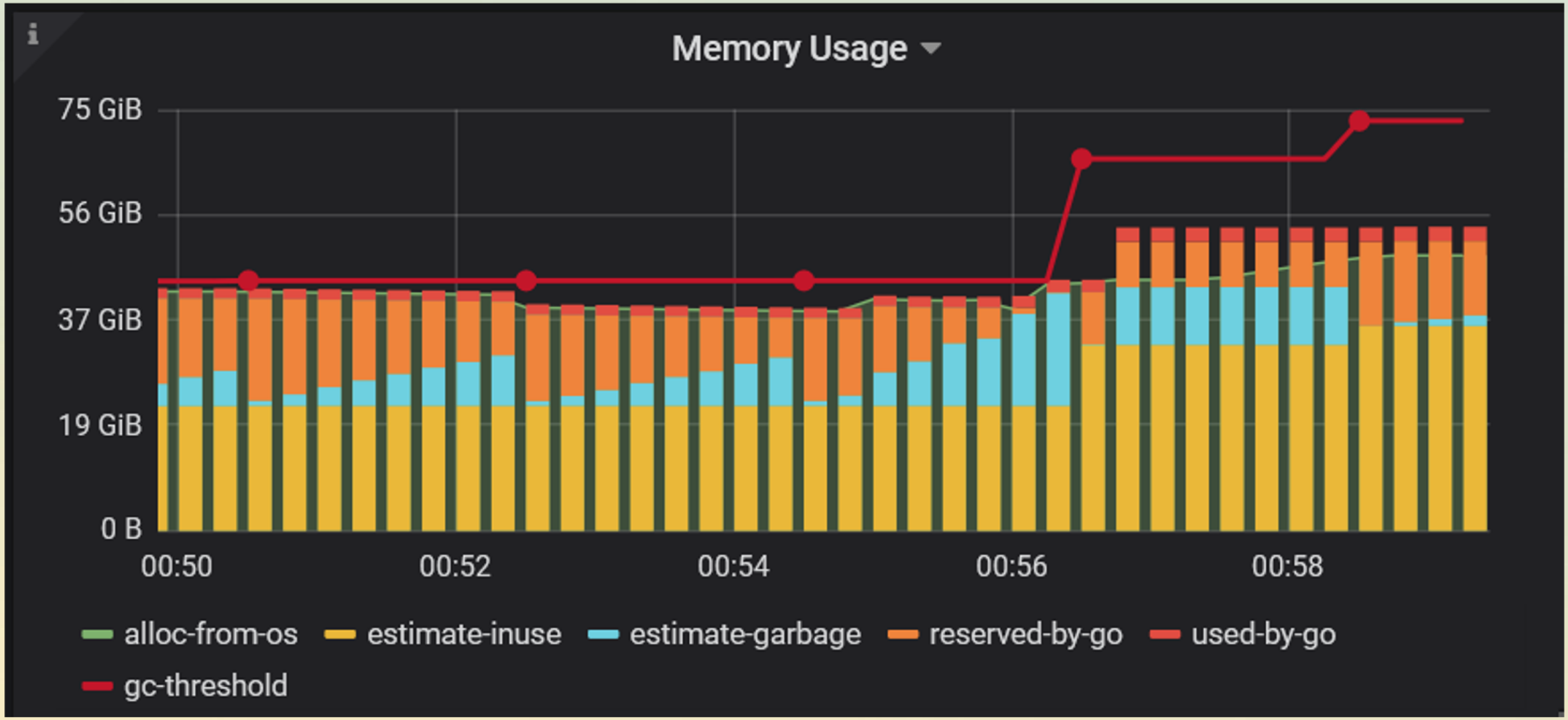
Part II - Memory control



Background: What are we talking about when we talk about Memory

- Go Runtime
  - Allocated from OS (mmaped)
  - Managed Memory
    - Should the memory be returned to the OS?
  - Inuse Memory
    - How much memory is *really* used?
  - GC
    - After each GC, the unused objects are released
  - Idle Memory
    - the unused objects are returned to Managed Memory, not OS

Background: What are we talking about when we talk about Memory



Background: What are we talking about when we talk about Memory

- OS
  - Virtual memory
    - each process has its virtual address space
  - Physical memory
    - OS divides physical memory into pages
  - memory mapping
    - page fault, built the virtual-physical mapping
  - RSS (resident set size)
    - Held in the RAM

Background: THP (transparent huge pages)

- TLB (translation lookaside buffer)
  - A CPU cache for virtual memory to physical addresses mapping
  - Size limited, when cache miss the memory access become slower
- Default page size 4K
- THP, increase page size to 2M or more
  - Page size larger, so the page entries for a process is less
  - Less page entries means better cache hit rate
  - Result in better memory access performance

Case study: THP (transparent huge pages)

OnCall / ONCAL[redacted]

[redacted]3.0.14 - tidb 内存使用异常，并且 heap memory usage 显示不正确

Edit | Comment | Assign | More | Under investigation | Problem located | Workflow

Details

Type:	Incident	Status:	CAN'T REPRODUCE (View Workflow)
Priority:	P2	Resolution:	Unresolved
Affects Version/s:	v3.0.14	Fix Version/s:	None
Component/s:	TiDB		
Labels:	None		
[redacted]			
Handler:	Reporter		

Description

profile debug.zip

Description

- The TiDB server memory footprint is abnormal

Description

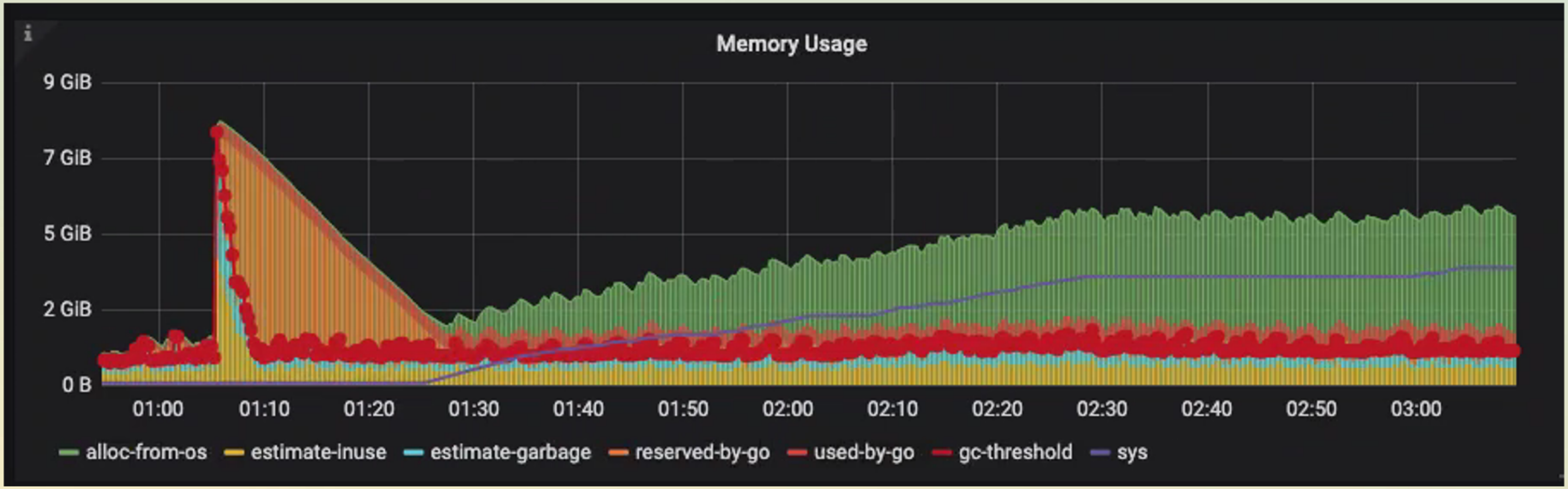
- The memory available on this node is not too much

```
top - 10:47:33 up 227 days, 19:26,  1 user,  load average: 18.24, 19.49, 20.16
Tasks: 474 total,   2 running, 472 sleeping,   0 stopped,   0 zombie
%Cpu(s):  2.1 us,  0.8 sy,  0.0 ni, 97.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 26335673+total, 38975960 free, 21687872+used,  7502052 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 43879948 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
352423 tidb      20   0  0.241t 0.200t  24948 R 101.3 81.3 175586:59 tidb-server
 17287 node_ex+  20   0  123788  35708   5244 S  13.2  0.0  28880:50 node_exporter
307144 tidb      20   0   39220  23708   4728 S   4.0  0.0   7395:08 blackbox_export
    10 root      20   0       0      0      0 S   0.3  0.0 344:15.44 rcu_sched
 20850 zabbix    20   0   83084   4116   3308 S   0.3  0.0   1083:45 zabbix_agentd
323996 root      20   0  344316  21920   3772 S   0.3  0.0  48:23.99 python3
     1 root      20   0   55240   7408   2584 S   0.0  0.0 154:47.06 systemd
     2 root      20   0       0      0      0 S   0.0  0.0   0:02.63 kthreadd
```

Investigate

- The Go Runtime thinks it does not use much memory
- The OS does not release the memory (RSS is high)



Investigate

Check /proc/{PID}/smaps to see the memory mapping of the TiDB server processor

AnonHugePages 1980416 kB

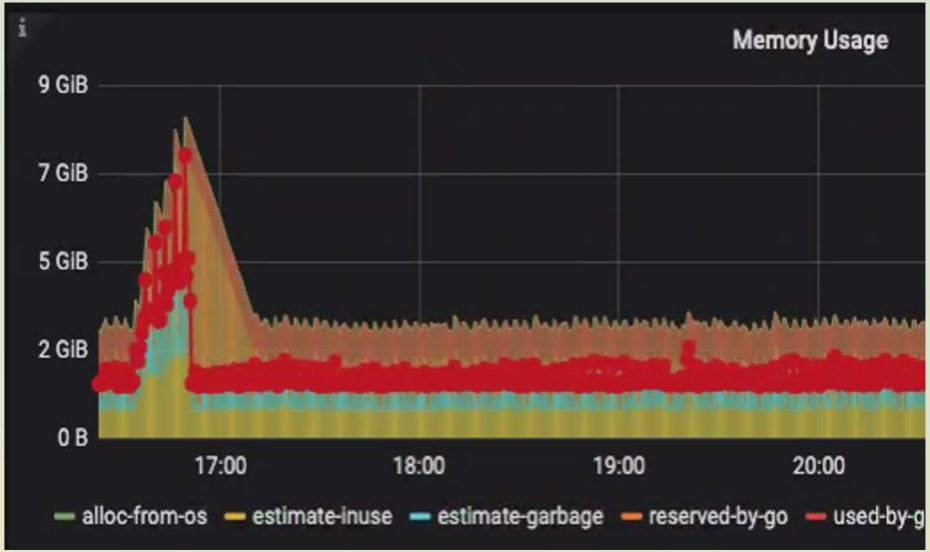
The memory is occupied by THP

```
VmFlags: rd wr mr mp me ac sd
c000000000-c3ac000000 rw-p 00000000 00:00 0
Size:           15400960 kB
Rss:             4767200 kB
Pss:             4767200 kB
Shared_Clean:          0 kB
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:   4767200 kB
Referenced:      4767200 kB
Anonymous:       4767200 kB
AnonHugePages:   1980416 kB
Swap:                  0 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Locked:                0 kB
ProtectionKey:         0
```

Investigate

Compare with other nodes in the cluster with THP disabled

```
c000000000-c3cc000000 rw-p 00000000 00:00 0
Size:           15925248 kB
Rss:             2395580 kB
Pss:             2395580 kB
Shared_Clean:          0 kB
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:   2395580 kB
Referenced:      2395580 kB
Anonymous:       2395580 kB
AnonHugePages:         0 kB
Swap:                  0 kB
KernelPageSize:        4 kB
```

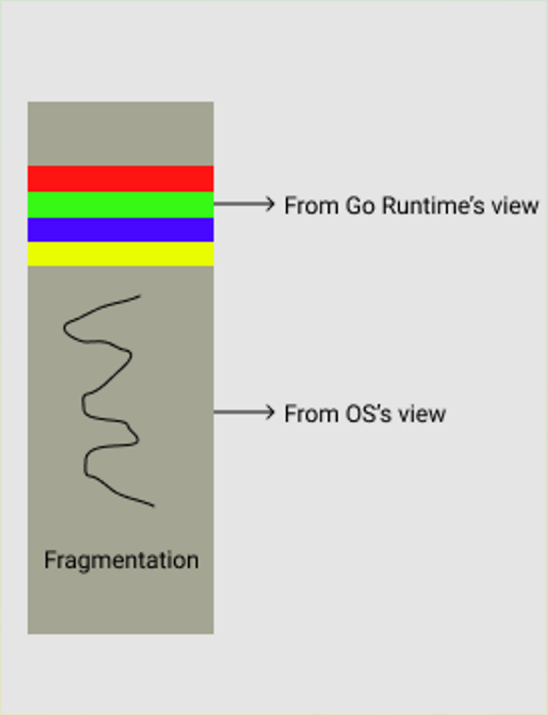


Analysis

- So, the root cause must be related to THP (transparent huge pages)
- But … why?

Analysis

- Go Runtime manage memory at 8K size granularity
- Go Runtime give hint to OS about the use of the memory block
- With THP enabled, the memory block address is round up to huge page boundary
- Fragmention!!!
- Fragmention make the memory difficult to be reclamed by the OS



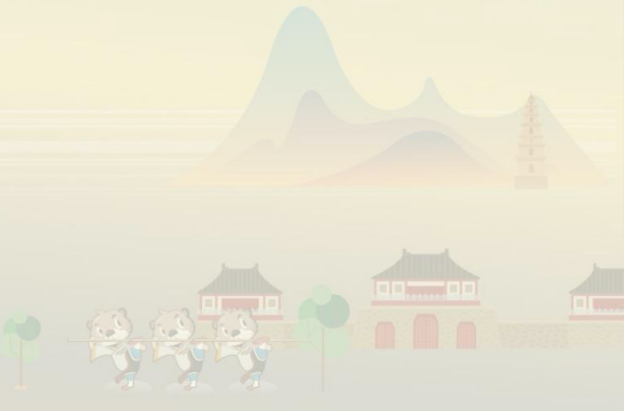
Analysis

- And more confusing behavior by the OS, merge pages into huge pages
  - The user program return 4K pages to OS
  - khugepaged turn a single 4K page to 2M
  - bloating the process's RSS by as much as 512x
- Some clues from the Go source code
  - https://github.com/golang/go/blob/release-branch.go1.11/src/runtime/mem_linux.go#L38

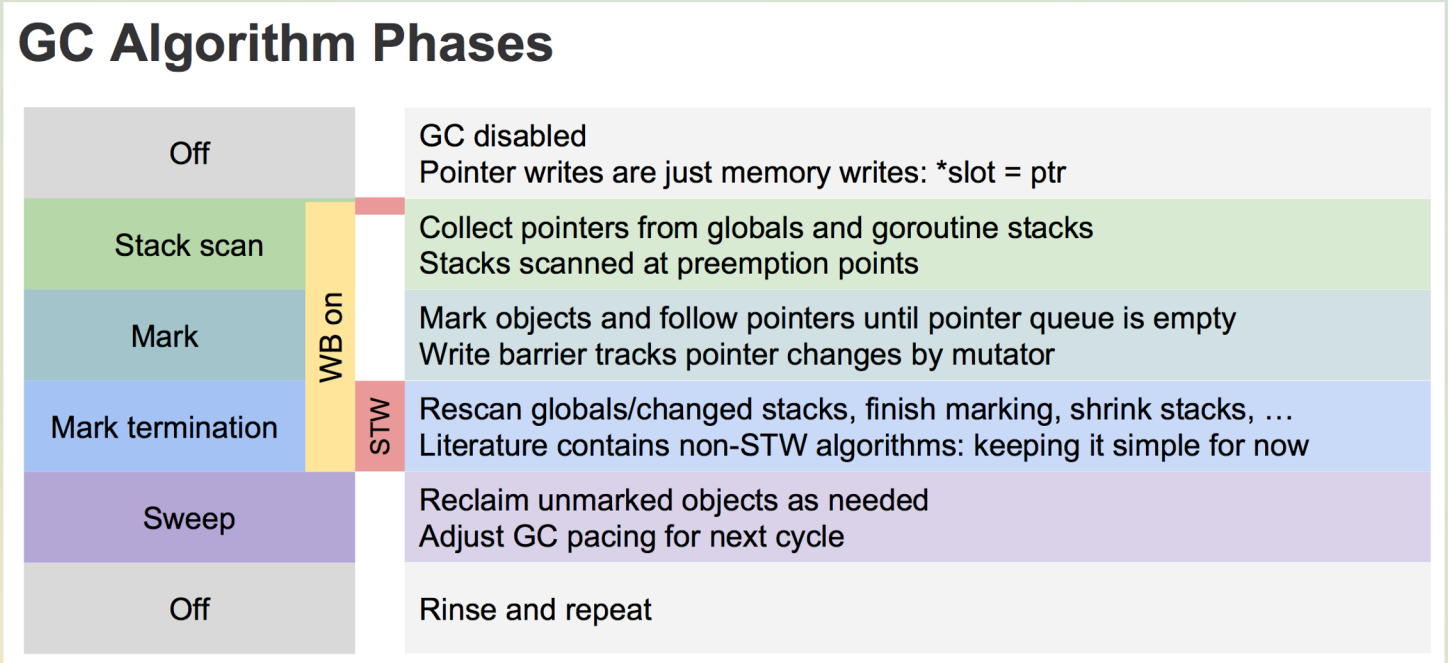
Follow up

- There are some more cases reported by the TiDB user, caused by THP
- TiDB has now disabled the use of THP

Part III - GC Related

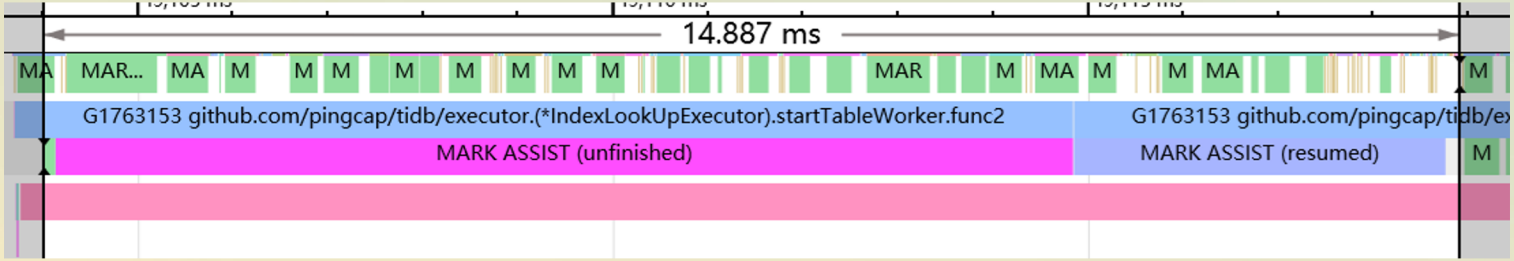


A brief introduction about GC



A brief introduction about GC

- What is STW(stop the world) exactly?
- What happen when collector can not catch up allocation rate?
- How the garbage recycle interrupt the user program?
  - STW takes too long
  - 25% CPU is dedicated to GC
  - MARK ASSIST
  - ...



Case study: GC Sweep caused latency jitter

OnCall / ONCALL[redacted]

[redacted] - v3.0.13 - 基于主键 Update 更新响应耗时浮动异常

Edit | Comment | Assign | More | Under investigation | Problem located | Workflow

Details

Type:	Incident	Status:	RESOLVED (View Workflow)
Priority:	P2	Resolution:	Done
Affects Version/s:	v3.0.13	Fix Version/s:	None
Component/s:	TiDB		
Labels:	None		
KeyUser:	是		

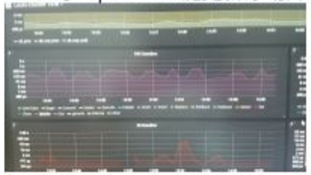
[redacted]

Root cause: 客户非常多库和表的统计信息导致 GO GC Sweep 慢问题 https://docs.google.com/document/d/1dKoJ9vjwL7K1z…

Handler: Reporter

Description

1、基于 update 主键更新浮动异常，快的30ms，慢的800ms



2、从慢日志看更新时间，parse、compile 等时间加起来都不足 10ms，IO CPU 网络都没瓶颈问题

Description

- The query duration is unstable, varies in a wide range from 30ms to 800ms
- No problem with the IO, CPU, network

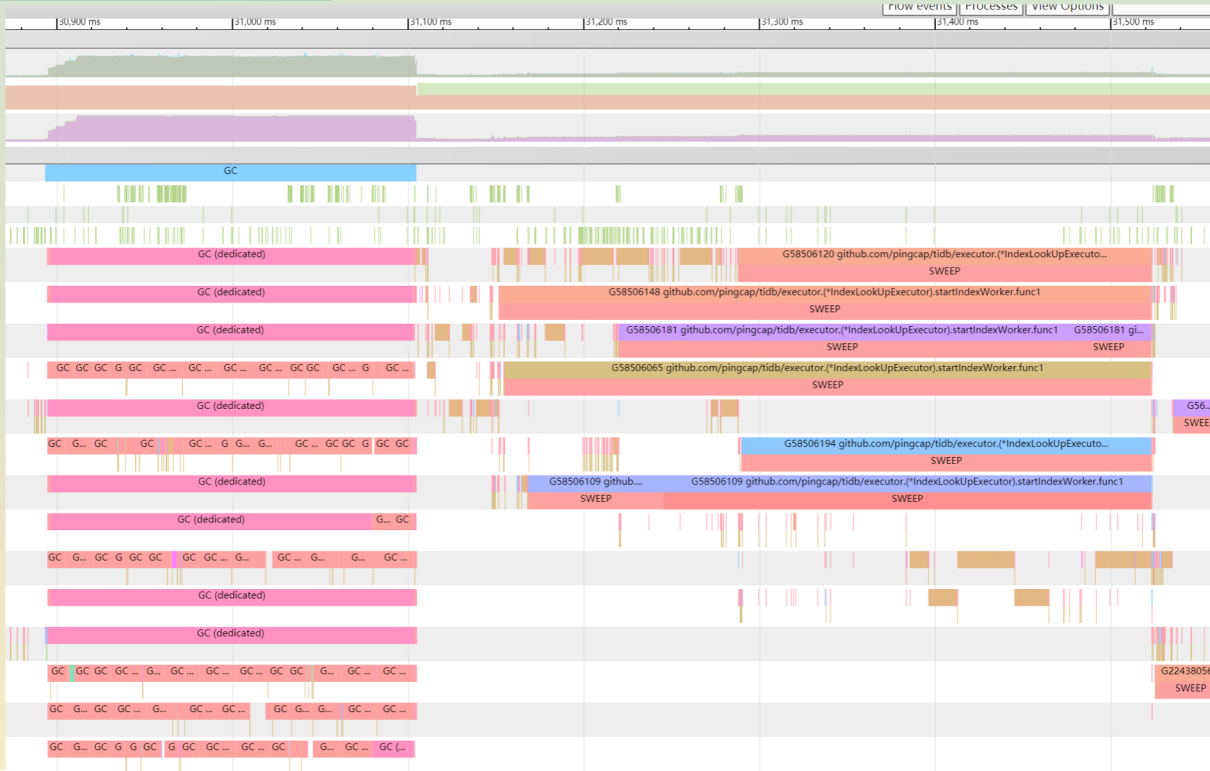
So, where does the latency come from?

Investigation

curl http://0.0.0.0:10080/debug/pprof/trace?seconds=60

Get the trace information

Observation：
A query is blocked by GC sweep



Investigation

Wall Duration	371,501,833 ns
Start Stack Trace	

Title
runtime.traceGCSweepSpan:1020
runtime.(*mspan).sweep:220
runtime.(*mcentral).cacheSpan:80
runtime.(*mcache).refill:138
runtime.(*mcache).nextFree:854
runtime.mallocgc:1022
runtime.makeslice:49
github.com/pingcap/tidb/util/chunk.newFixedLenColumn:133
github.com/pingcap/tidb/util/chunk.New:66
github.com/pingcap/tidb/util/chunk.NewChunkWithCapacity:51
github.com/pingcap/tidb/executor. (*indexWorker).fetchHandles:732
github.com/pingcap/tidb/executor. (*IndexLookUpExecutor).startIndexWorker.func1:526

▼ Args

Swept bytes	1118142464
Reclaimed bytes	8192

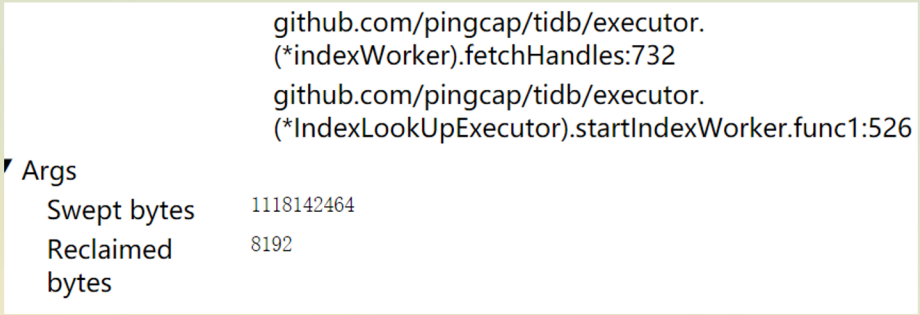
Analysis

- The garbage collector can stop individual goroutines, even without STW
  - eg. A goroutine that needs to allocate memory right after a GC cycle is required to help with the sweep work

- It's… er … so long?
  - Let's dig the rabbit hole

Analysis

- Scan 1G memory
- So, that's why it takes so long!
- But release only 8K?



Analysis

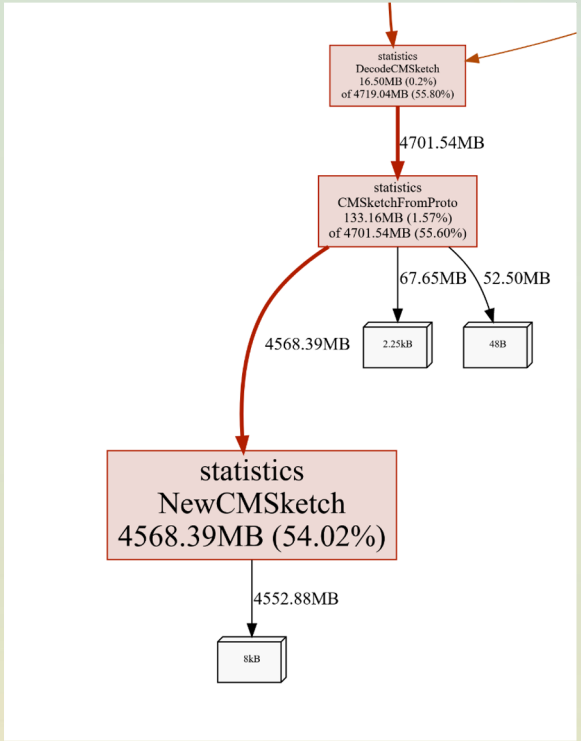
And we find a similar issue [latency in sweep assists when there's no garbage](), and also the [fix]()

```
                                                                    131 +
115     // Now try partial unswept spans.                           132     // Now try partial unswept spans.
116 -   for {                                                        133 +   for ; spanBudget >= 0; spanBudget-- {
117         s = c.partialUnswept(sg).pop()                           134         s = c.partialUnswept(sg).pop()
118         if s == nil {                                            135         if s == nil {
119             break                                                136             break
    @@ -132,7 +149,7 @@ func (c *mcentral) cacheSpan() *mspan {
132     }                                                            149     }
133     // Now try full unswept spans, sweeping them and putting them into the  150     // Now try full unswept spans, sweeping them and putting them into the
134     // right list if we fail to get a span.                      151     // right list if we fail to get a span.
135 -   for {                                                        152 +   for ; spanBudget >= 0; spanBudget-- {
136         s = c.fullUnswept(sg).pop()                              153         s = c.fullUnswept(sg).pop()
137         if s == nil {                                            154         if s == nil {
138             break                                                155             break
```

Analysis

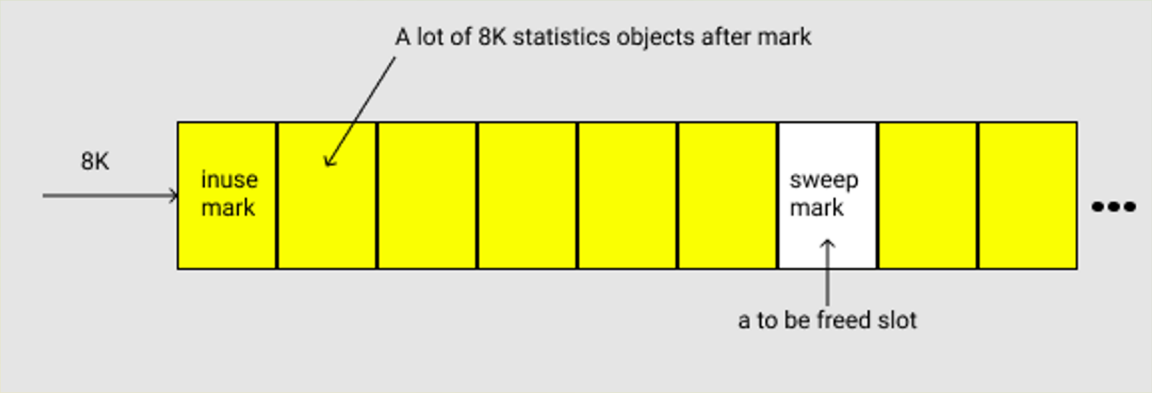
- Verified with the fix, the latency jitter gone
  - yeah, that's the problem

- One more thing … how TiDB trigger that bug?
  - statistics code use a lot of 8K objects
  - statistics objects are all alive after GC



Analysis

- Sweep, until it finds a free slot ...
- … or there are no more spans to sweep
- a lot of inuse objects after the mark phase



Conclusion

- Although the STW duration is low enough for Go GC
- Unexpected latency jitter caused by GC may still exist

Case study: Lock and NUMA aware

TiDB / [redacted] - v2.1.10 - QPS 有瓶颈问题排查

Edit | Comment | Assign | More | WON'T FIX | DUPLICATED | Workflow

Details

Type:	Bug	Status:	JOB CLOSED (View Workflow)
Priority:	High	Resolution:	Done
Affects Version/s:	2.1.10	Fix Version/s:	DO NOT APPLY
Component/s:	executor		
Labels:	OnCall		
RCA:	单机多实例		
Minimal Reproduce Step:	null		
Solution:	单机多实例		
Severity:	Moderate		
Github PRs:	null		
Sprint:	TiDB 201907, TiDB 201908		
TiBug Status:	Puzzled		

Description

- TiDB server CPU usage is around 60%
- Networking and IO are both OK
- Increase the benchmark client concurrency, throughput does not increase
- It seems the resources are not exhausted, but where is the bottleneck?

Reproduce

- The table is wide in the customer environment
  - CPU usage is higher with less columns
- The deployment does not consider NUMA
  - CPU usage is higher when binding CPUs to NUMA nodes

指标	178 Column 的 Table（RAW）	178 Column 的 Table（优化）	178 Column 的 Table（绑核）	8 Column 的 Table（RAW）
QPS	4.3K	5.2K	7.59K	29.4K
Duration	P99：8s P999：8s	P99：1.92s P999：3.06s	P99：2s P999：2s	P99：466ms P999：916ms
CPU Usage	58.23%	64%	88.49%	83.33%

Reproduce

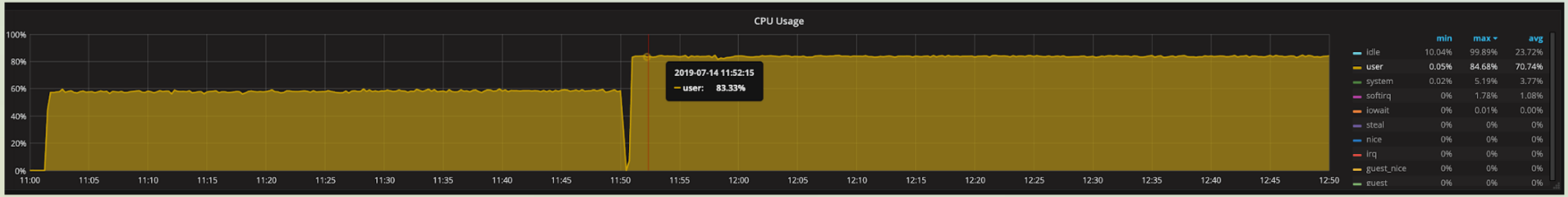
- Context Switch, Minor Page Fault, and NUMA Hit Miss are related

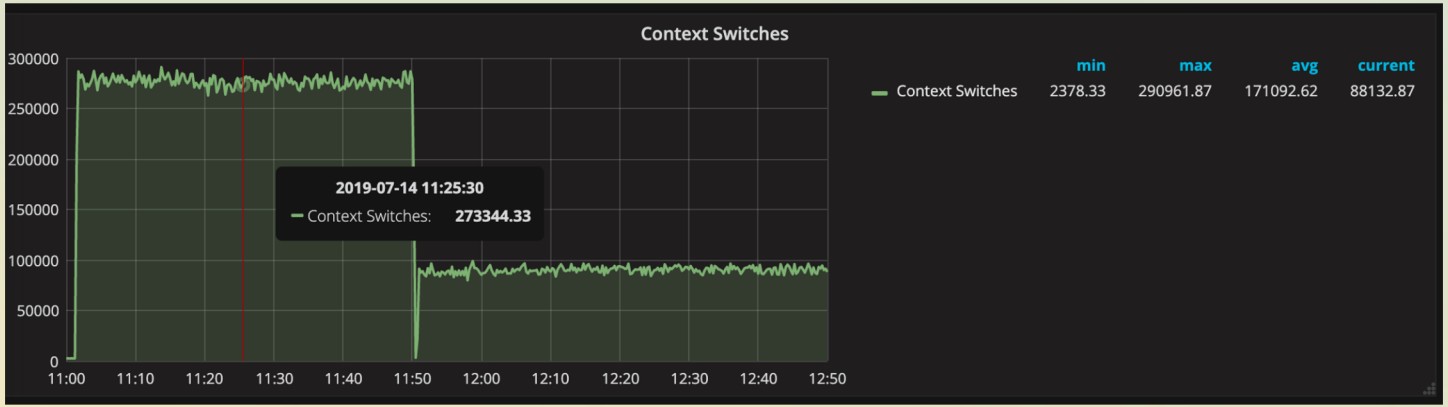
Mem Usage	3.4GB	2.979GB	3.2GB	870MB
Context Switch	273.344K	248.12K	193.847K	84.078K
Minor Page Fault	179.3K	229.8K	8.68K	13.6K
NUMA Statistics **HIT** **Miss** **Page Migration**	2.97K 0 1.25K	3.59K 0 1.30K	3.34K 0 125	674 0 198

Reproduce

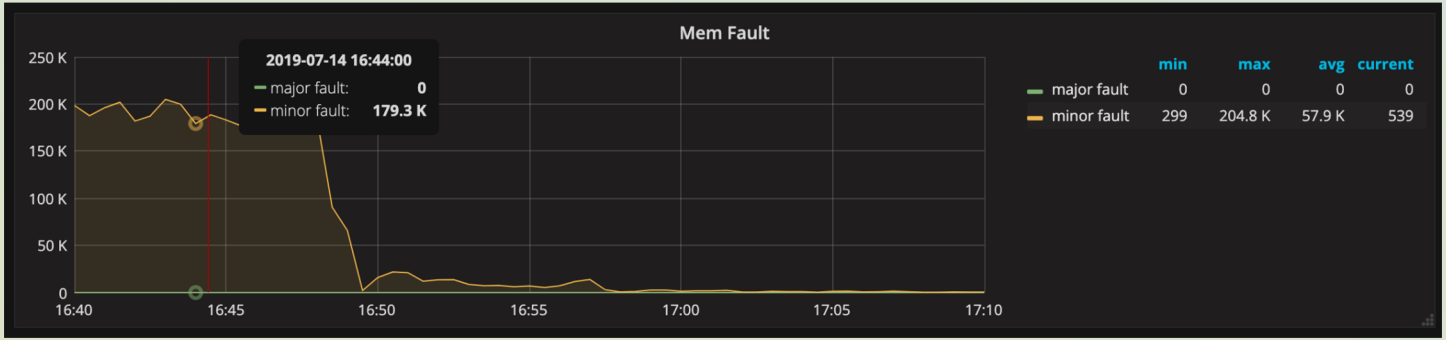

QPS
35 K
30 K
25 K
20 K
15 K
10 K
5 K
0
11:00
11:10
11:20
11:30
11:40
11:50
12:00
12:10
12:20
12:30
12:40
12:50
query OK Max: 29.56 K Current: 29.39 K
Duration
32 s
8 s
2 s
500 ms
125 ms
31 ms
8 ms
2 ms
11:00
11:10
11:20
11:30
11:40
11:50
12:00
12:10
12:20
12:30
12:40
12:50
999
99
95
80

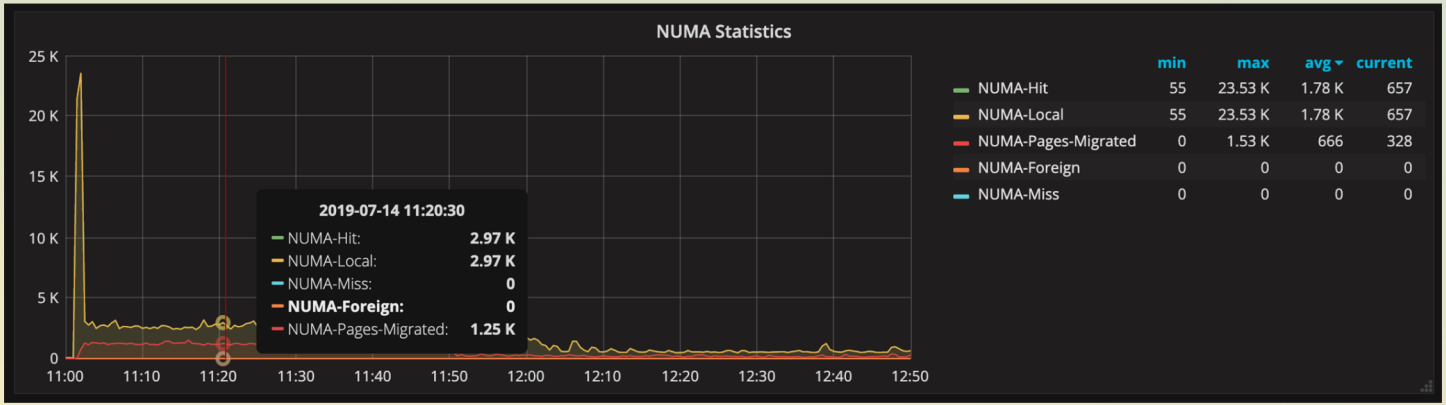
Reproduce


CPU Usage
100%
80%
60%
40%
20%
0%
11:00 11:05 11:10 11:15 11:20 11:25 11:30 11:35 11:40 11:45 11:50 11:55 12:00 12:05 12:10 12:15 12:20 12:25 12:30 12:35 12:40 12:45 12:50
2019-07-14 11:52:15
user: 83.33%
min max avg
idle 10.04% 99.89% 23.72%
user 0.05% 84.68% 70.74%
system 0.02% 5.19% 3.77%
softirq 0% 1.78% 1.08%
iowait 0% 0.01% 0.00%
steal 0% 0% 0%
nice 0% 0% 0%
irq 0% 0% 0%
guest_nice 0% 0% 0%
guest 0% 0% 0%


Context Switches
300000
250000
200000
150000
100000
50000
0
11:00 11:10 11:20 11:30 11:40 11:50 12:00 12:10 12:20 12:30 12:40 12:50
2019-07-14 11:25:30
Context Switches: 273344.33
min max avg current
Context Switches 2378.33 290961.87 171092.62 88132.87

Reproduce


Mem Fault
250 K
200 K
150 K
100 K
50 K
0
16:40
16:45
16:50
16:55
17:00
17:05
17:10
2019-07-14 16:44:00
major fault: 0
minor fault: 179.3 K
min max avg current
major fault 0 0 0 0
minor fault 299 204.8 K 57.9 K 539


NUMA Statistics
25 K
20 K
15 K
10 K
5 K
0
11:00
11:10
11:20
11:30
11:40
11:50
12:00
12:10
12:20
12:30
12:40
12:50
2019-07-14 11:20:30
NUMA-Hit: 2.97 K
NUMA-Local: 2.97 K
NUMA-Miss: 0
NUMA-Foreign: 0
NUMA-Pages-Migrated: 1.25 K
min max avg current
NUMA-Hit 55 23.53 K 1.78 K 657
NUMA-Local 55 23.53 K 1.78 K 657
NUMA-Pages-Migrated 0 1.53 K 666 328
NUMA-Foreign 0 0 0 0
NUMA-Miss 0 0 0 0

Hypothesis

- Something make the CPU poorly used
- So the bottleneck should be one of them?
  - Mutex
  - GC
  - NUMA

Is it caused by mutex?

curl -G "172.16.5.2:10080/debug/pprof/mutex" > mutex.profile

go tool pprof -http 0.0.0.0:4002 mutex.profile

- GRPC
- log slow query
- log slow cop task
- query feedback

After optimization:

QPS 4.3K => 5.2K, CPU 58.2% => 64%

Is it caused by NUMA?

Bind CPUs to NUMA nodes

- QPS 5.2K => 7.59K
- P999 3.06s => 2s
- CPU usage 64% => 88.49%
- Minor page fault 229.8K => 8.68K

This single change get the biggest improvement!

Is it caused by GC?

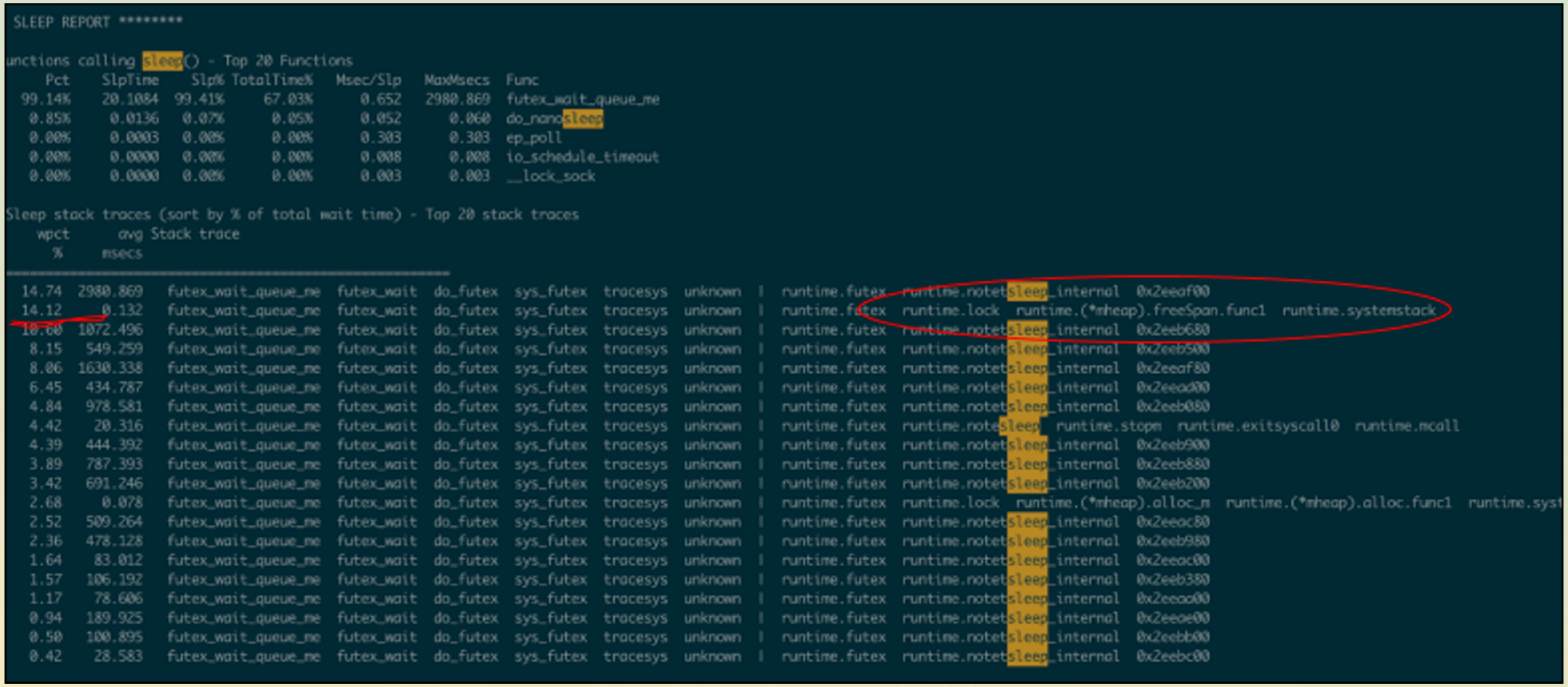
- Off CPU analysis with linuxki
  - From the scheduler activity report, 67% of time is spend on sleeping

```
********* SCHEDULER ACTIVITY REPORT ********
RunTime     :  9.363532  SysTime    :  0.390197   UserTime    :  8.910223
SleepTime   : 20.228043  Sleep Cnt :     31129   Wakeup Cnt  :     31316
RunQTime    :  0.408500  Switch Cnt:     32598   PreemptCnt  :      1469
HardIRQ     :  0.007473  HardIRQ-S :  0.000332    HardIRQ-U :  0.007141
SoftIRQ     :  0.055638  SoftIRQ-S :  0.002849    SoftIRQ-U :  0.052789
Last CPU    :         4  CPU Migrs :      1792   NODE Migrs  :        28
Policy      : SCHED_NORMAL     vss :   1890638          rss :    843396

busy     :     31.21%
  sys    :      1.30%
  user   :     29.70%
  irq    :      0.21%
runQ     :      1.36%
sleep    :     67.43%
```

Is it caused by GC?

runtime.(*mheap).freeSpan!!! it’s about 14%



Analysis

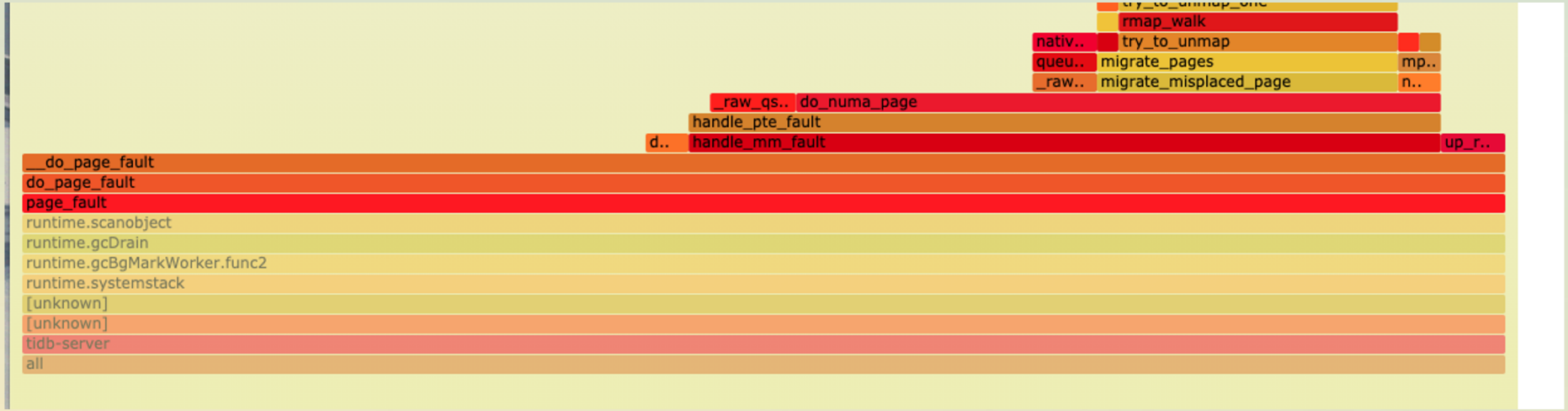
- There is a lock on the page allocator, see this issue make the page allocator scale
- A high allocation rate combined with a high degree of parallelism leads to significant contention
- The performance is not scalable on a machine with lots of CPUs
    - PS: The Go folks have already done a lot of work on it

Background: a bit about NUMA

- NUMA (non-uniform memory access)
  - sockets
  - CPUs
  - local memory access
- OS allocation strategy
  - balanced
  - local
- Minor Page Faults might be related to the page migration caused by NUMA balancing

Analysis: NUMA aware

- This is the same issue reported by other user
- GC is not NUMA aware



Conclusion

- Those factors make Go runtime less scalable on large number of cores
    - Lock contention in the allocation path
    - GC is not NUMA aware
    - ...

Recap

- CASE1: Batching client requests
  - latency in scheduler
- CASE2: THP(transparent huge pages)
  - memory control
- CASE3: GC sweep caused latency jitter
- CASE4: Lock and NUMA aware
  - scalability on multiple cores

GOPHER CHINA 2020

中国 上海 / 2020-11.21-22