Stack Traces and Flame Graphs for Oracle Troubleshooting

Luca Canali, CERN

UKOUG, Birmingham, December 6th, 2015
About Luca

• Senior DBA and team lead at CERN IT
• Joined CERN in 2005
• Working with Oracle RDBMS since 2000
• Passionate to learn and share knowledge, how to get most value from database technology
• @LucaCanaliDB and http://cern.ch/canali
Large Hadron Collider – Run 2: Resumed Operations after Upgrades
In 2015 Proton-Proton Collisions at 13 TeV  (was 8 TeV in 2013)

http://build-your-own-particle-detector.org/models/lhc-micro-models
Agenda

- Basic techniques for stack backtraces
- Stack profiling
- Visualization with flame graphs
- Examples
Motivations and Context: Troubleshooting

- **How understand the workload?**
- **You need the right tools for the job**
  - What is the system doing?
  - Where is the time spent?
- **Oracle has mature instrumentation**
  - **Wait event model, ASH, AWR, SQL monitor and a large amount of info on V$ etc**
Why OS-Level Tools?

• When wait events are not available
  • Servers with large memory means queries operating mostly ‘on CPU’

• When wait events are not enough
  • Measure I/O latency, understand what is inside I/O waits

• Exploring Oracle internals
  • For those cases where we need to drill down deeper
OS Tools

- Standard tools: top, vmstat, pidstat, iostat, sar, /proc, ...

- Modern tools for dynamic tracing: DTrace, SystemTap, perf-events, ...

- Stack tracing
  - Tools: sqlplus with oradebug, gdb, perf-events, custom tools, /proc/<pid>/stack
Stack Trace Using SQL*Plus

- **Sqlplus**: `oradebug`
  - Many diagnostic functions, including stack tracing
  - Note: ignore overhead at the top of the stack

```sql
SQL> oradebug setospid <pid>
SQL> oradebug dump callstack 1
SQL> oradebug dump errorstack 1
SQL> oradebug short_stack

ksedsts() +244<--ksdxfstk()+58<--ksdxcb()+918<--sspuser()+224<--___sighandler()<-pread64()+19<--ksfd_skgfqio()+303<--ksfd_io()+823<--ksfdread()+182<--kfk_ufs_sync_io()+797<--kfk_submit_ufs_io()+88<--kfk_submit_io()+114<--kfk_iol()+989<--kfkRequest()+32<--kfk_transitIO()+1248<--kfioSubmitIO()+3085<--kfioRequestPriv()+218<--kfioRequest()+685..[EDITED]..<-__libc_start_main()+253
```
Stack Traces, SQL Trace and Wait Events

- **Advanced** syntax for **events** and tracing
- **Credits:** I first learned about this from Tanel Poder
- **Examples:**

```sql
SQL> alter session set events 'sql_trace callstack(1)'; --11g

SQL> alter session set events 'wait_event["<wait event name>"]
trace("shortstack: %s\n", shortstack())'; --12c

```
Stack Traces with OS tools

- Standard OS tools for stack tracing:
  - **gdb**, gnu debugger, use ‘bt’ command for stack backtrace
  - Wrappers on top of **gdb**: `pstack`, `gstack`
- Example:

```
$ pstack 24318
#0 0x0000000340ae0f0c3 in __pread_nocancel () from /lib64/libpthread.so.0
#1 0x000000000cda5bd7 in skgfqio ()
#2 0x000000000cc81bcf in ksfd_skgfqio ()
#3 0x000000000cc7ca57 in ksfd_io ()
#4 0x000000000cc7c156 in ksfdread ()
#5 0x00000000060ada9d in kfk_ufs_sync_io ()
#6 0x00000000060acba8 in kfk_submit_ufs_io ()
#7 0x00000000060ab612 in kfk_submit_io ()
#8 0x00000000060a848d in kfk_iol ()
#9 0x00000000060a7e90 in kfkRequest ()
... #55 0x0000000000b00000bd83c in main ()
```
More Linux Tools for Stack Tracing

- Lsstack64 using ptrace and libunwind: https://github.com/jarun/lsstack64
- Tbstack: more lightweight, async stack unwinding -&gt; https://github.com/tbricks/tbstack

$ ./tbstack 24970
------------------------------- thread 1 (24970) -----------------------------
000000340ae0f0c3 pread64 + 0x13
000000000cda5bd7 skgfqio + 0x227
000000000cc81bcf ksfd_skgfqio + 0x12f
000000000cc7ca57 ksfd_io + 0x337
000000000cc7c156 ksfdread + 0xb6
00000000060ada9d kfk_ufs_sync_io + 0x31d
00000000060acba8 kfk_submit_ufs_io + 0x58
...
000000340aa1ed5d __libc_start_main + 0x1fd
000000000b0d679 _start + 0x29
------------------------------- summary -----------------------------
time the process was frozen: 0ms 808us
Kernel Stack Backtrace From /proc

- /proc/<pid>/stack
- Requires kernel built with CONFIG_STACKTRACE
  - Notably, available in RHEL6 / OL6
  - Check: grep CONFIG_STACKTRACE /boot/config-`uname -r`

```
$ cat /proc/25894/stack
[fffffffffffffff811d260d] __blockdev_direct_IO_newtrunc+0xb7d/0x1270
[fffffffffffffff811d2d77] __blockdev_direct_IO+0x77/0xe0
[fffffffffffffff811ceff7] blkdev_direct_IO+0x57/0x60
[fffffffffffffff811293db] generic_file_aio_read+0x6bb/0x700
[fffffffffffffff811ce4e1] blkdev_aio_read+0x51/0x80
[fffffffffffffff811918da] do_sync_read+0xfa/0x140
[fffffffffffffff8119121d5] vfs_read+0xb5/0x1a0
[fffffffffffffff8119233a] sys_pread64+0x7a/0x90
[fffffffffffffff8100b0d2] system_call_fastpath+0x16/0x1b
[ffffffffffffffffffffffff] 0xffffffffffffffff
```
Stack Profiling and Sampling

• Evolves from typical usage of stack info
  • Repeated samples of stack backtraces

• Answer questions
  • Is anything changing between samples?
  • What are the most common code paths?

• Analogy: the flip book
Basic example: Workload with a Network Bottleneck

- **SQL>** SELECT * FROM LARGETABLE;
- Note also SQL> SET AUTOTRACE TRACEONLY
- Data transfer over *slow network*
- port forwarding into a VM: putty -P <forwarded_port> -L 1111:DB_server:1521 oracle@VM_host
- Oracle process cannot “pump data”: mostly idle

<table>
<thead>
<tr>
<th>Time</th>
<th>PID</th>
<th>%usr</th>
<th>%system</th>
<th>%guest</th>
<th>%CPU</th>
<th>CPU</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:10:28 PM</td>
<td>9124</td>
<td>0.00</td>
<td>4.00</td>
<td>0.00</td>
<td>4.00</td>
<td>0</td>
<td>oracle_9124_orc</td>
</tr>
<tr>
<td>10:10:29 PM</td>
<td>9124</td>
<td>4.85</td>
<td>4.85</td>
<td>0.00</td>
<td>9.71</td>
<td>2</td>
<td>oracle_9124_orc</td>
</tr>
<tr>
<td>10:10:30 PM</td>
<td>9124</td>
<td>1.00</td>
<td>2.00</td>
<td>0.00</td>
<td>3.00</td>
<td>2</td>
<td>oracle_9124_orc</td>
</tr>
</tbody>
</table>
KStackSampler

- A basic sampler of the kernel stack and process state, written in shell script

```bash
for x in $(seq 1 $iterations); do
cat /proc/$pid/stack
  # get kernel stack trace from /proc
  # this is an approximation as the process state may have changed to runnable in the meantime
grep -m 1 State /proc/$pid/status
echo "1"
  # this makes the output ingestible by FlameGraph/stackcollapse-stap.pl

echo ""
sleep $interval
done
```
Kernel Stack Sampling

- **Step1**: Kernel stack and OS state **sampling**
  - Using https://github.com/LucaCanali/Stack_Profiling

  ```
  $ ./kstacksampler.sh -p 9124 -n 100 -i .05 | grep -v 0xffffffffffffffff | sed 's/State:\t//g' | sed 's/\[[<.*>]/ //g' >stack_sample_example1.txt
  ```

- **Step2**: Flame Graph
  - Using https://github.com/brendangregg/FlameGraph

  ```
  $ cat stack_sample_example1.txt | ../FlameGraph/stackcollapse-stap.pl | ../FlameGraph/flamegraph.pl --title "Kernel stack profiling - Example 1, network-related bottleneck">
  kstacksampler_network_bottleneck_Fig1.svg
  ```
Flame Graph for Analysis

• Stack traces and OS status are collapsed and sorted, colours added to provide contrast
• Line length proportional to the number of samples (most popular functions have longer lines)

Finding: Oracle process is waiting for network messages
• Confirmed with Oracle wait interface: session mostly waiting for “SQL*Net message from client”
Actionable Information

- From previous graph we see a network bottleneck
- How to make improve the throughput?
  - Action: reduce impact of round trips
  - Sqlplus: "SET ARRAYSIZE 5000"

Results:
- The flame graph shows more time is spent running in userspace mode (more work being done)
- DB metrics confirm data throughput increase
Comments on Flame Graphs

- Help to identify most common code paths
  - where most time is spent

- The current function is at the top of the stack
  - Parent functions in lower layers add context

- Gotcha: the horizontal axis is not time!
  - Stacks are collapsed and sorted

- Gotcha: sampling frequency is important
  - Trade off accuracy vs. overhead

- Gotcha: need to know if you are capturing all workload or only parts
  - Perf for example only captures processes on CPU
Example: Wait Event Investigation

- AWR report shows: %CPU + %WAIT > 100%
- Will use **extended stack profiling** for investigating
  - Wait events and CPU usage
- Extract from the AWR report:

  ![Top 10 Foreground Events by Total Wait Time](image)

  - Case first discussed by John Beresniewicz (JB) at Oak Table World 2015
The Test System

- Workload: Oracle process random read
- Tool: http://kevinclosson.net/slob/

- Test system is in a VM
  - Oracle 12.1.0.2 with ASM (block device storage)
  - Fast I/O, DB files are cached in host RAM (128 GB)
  - Only one active process
  - No significant additional load on the system
## Tanel’s Snapper Data

### Session Snapper v3.64 by Tanel Poder

<table>
<thead>
<tr>
<th>User</th>
<th>Stat</th>
<th>Delta</th>
<th>Time (sec)</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER1</td>
<td>Opened Cursors Cumulative</td>
<td>1697.0</td>
<td>169.7</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Recursive CPU Usage</td>
<td>690.0</td>
<td>69.0</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Session Logical Reads</td>
<td>110510.0</td>
<td>11.0k</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>CPU Used by this Session</td>
<td>8991.0</td>
<td>89.9</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>User I/O Wait Time</td>
<td>879.0</td>
<td>87.9</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Non-IDle Wait Time</td>
<td>879.0</td>
<td>87.9</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Physical Read Total IO</td>
<td>1029988.0</td>
<td>10.3K</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Physical Read Total Bytes</td>
<td>843677696.0</td>
<td>84.37M</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Consistent Gets</td>
<td>110573.0</td>
<td>11.0k</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Consistent Gets from Cache</td>
<td>110573.0</td>
<td>11.0k</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Consistent Gets Pin (FastPath)</td>
<td>108875.0</td>
<td>10.8k</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Consistent Gets Examination</td>
<td>108875.0</td>
<td>10.8k</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Logical Read Bytes from Cache</td>
<td>905584016.0</td>
<td>90.5M</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Physical Reads</td>
<td>103052.0</td>
<td>10.31K</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Physical Reads Cache</td>
<td>103052.0</td>
<td>10.31K</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Free Buffer Requested</td>
<td>103052.0</td>
<td>10.31K</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Pinned Buffers Inspected</td>
<td>15.0</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Free Buffer Inspected</td>
<td>103063.0</td>
<td>10.31K</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Physical Reads Cache Prefetch</td>
<td>98337.0</td>
<td>9.83K</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Shared Hash Latch Upgrades - No Wait</td>
<td>37203.0</td>
<td>3.72K</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Calls to Get Snapshot Scn: Kcmgss</td>
<td>1700.0</td>
<td>170.0</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>File Io Service Time</td>
<td>4274931.0</td>
<td>42.74K</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>File Io Wait Time</td>
<td>4982896.0</td>
<td>49.82K</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Batched IO Vector Read Count</td>
<td>1693.0</td>
<td>169.3</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Batched IO Block Count</td>
<td>98735.0</td>
<td>9.87K</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Batched IO Single Block Count</td>
<td>3.0</td>
<td>0.3</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Batched IO Block Miss Count</td>
<td>98819.0</td>
<td>9.88K</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Batched IO Double Miss Count</td>
<td>16979.0</td>
<td>16.98K</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>No Work - Consistent Read Gets</td>
<td>109385.0</td>
<td>10.94K</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Table Fetch By Rowid</td>
<td>270798.0</td>
<td>27.08K</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Index Scans Kofiks</td>
<td>1721.0</td>
<td>17.21</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Session Cursor Hits</td>
<td>1740.0</td>
<td>17.40</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Buffer Is Pinned Count</td>
<td>107949.0</td>
<td>10.79K</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Buffer Is Not Pinned Count</td>
<td>314929.0</td>
<td>31.49K</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Execute Count</td>
<td>61740.0</td>
<td>61.74K</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Db Time</td>
<td>57815.0</td>
<td>5.78ms</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Db Time + Execution Elapsed Time</td>
<td>71693.0</td>
<td>71.69ms</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Db Time + Wait Elapsed Time</td>
<td>10057609.0</td>
<td>100.57609s</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Db File Sequential Read</td>
<td>914792.0</td>
<td>91.48ms</td>
<td></td>
</tr>
<tr>
<td>USER1</td>
<td>Db File Parallel Read</td>
<td>7866335.0</td>
<td>78.66ms</td>
<td></td>
</tr>
</tbody>
</table>

--- End of Stats snap 1, seconds=10

**ASH is OK: %CPU + %WAIT = 100%**

**Anomaly:** DB CPU + WAIT > DB Time!
Extended Stack Profiling

• ORA_KStackProfiler a tool for:
  • sampling /proc/<pid>/stack to sample the kernel stack trace if any
  • sampling of /proc/<pid>/stat to measure the process state (Running, Sleeping, Disk, etc),
  • sampling the status of the Oracle session (on CPU of waiting for a wait event).
  • The sampling is done directly from SGA (X$KSUSE)
ORA_KStackProfiler - Example

The script finds the OS pid and address of X$KSUSE.KSUSEOPC and X$KSUSE.KSUSETIM for a given Oracle session. To be used together with ora_kstackprofiler. Run as user SYS.

Enter Oracle SID to be investigated: 111
OS pid = 1234, ksuseopc = 5796061762, ksusetim = 5796061792

3. Stack traces sampling extended with Oracle wait event data as in the following example:

```
/ora_kstackprofiler --pid 1234 --delay 100 --count 1000 --ksuseopc 5796061762 --ksusetim 5796061792 > ora_stack_slob_fastio.txt
```

4. Process the output with additional filtering and with flame graphs for visualization using https://github.com/brendangregg/FlameGraph

```
cat ora_stack_slob_fastio.txt | grep -v 0xffffffffffffffffffffff | sed -f oracle_scripts/eventsname.sed | ../FlameGraph/stackcollapse-stap.pl | ../FlameGraph/flamegraph.pl --title "Flame Graph of Oracle random I/O (SLOB)" > ora_stack_slob_fastio.svg
```
Flame Graph and Analysis

- Kernel stack, with OS state and wait events
- **Finding**: many CPU cycles inside I/O wait events
What Is the Added Value of The Flame Graph Analysis

- Extended stack profiling and flame graph
  - Provide direct evidence that CPU is used inside the wait event
- Why CPU Time + Wait Time > DB Time?
  - Oracle measures wait events by timing the begin and end of the related operations (in this case system calls for I/O)
  - CPU time is measured by OS, independently of Oracle
  - The two measurements overlap (double counting of CPU which can be important, as in the previous example)
Stack Profiling with Userspace

Ptrace_Profiler:
- /proc/pid/stack
- Userspace trace
- /proc/pid/stat
- Wait Event from X$KSUSE/SGA
- Adds context to understand workload
Notes on Ptrace_Profiler

- **On-off CPU sampling**
  - Written in C, profiles stack, OS, wait events from SGA
  - Gathers data both if the session is on or off CPU
  - Not only for Oracle, use mainly for non CPU-bound.

- **Important limitations**
  - Inaccurate as collection from different data sources is not atomic
  - Important overhead from userspace stack collection and unwinding, currently using libunwind-ptrace

- **Link:**
  - https://github.com/LucaCanali/Stack_Profiling
Additional Investigation on I/O

- High CPU usage in kernel mode because of high I/O rate and low latency
- Direct measurement of I/O latency
  - Use tracepoints for block I/O.
  - Example with Ftrace (based on B. Gregg’s work)

```
# ./iolatency_micro 10

Tracing block I/O. Output every DeltaT = 10 seconds. Ctrl-C to end.
>=(msec) .. <(msec) : IOPS  IO_latency/DeltaT  |IOPS Distribution |
 16  ->  32 :  0          0                   |                 |
 32  ->  64 :  623        29947               |####|
 64  -> 128 :  8392      805049              |###################################|
 128 -> 256 :  3482      668659              |#####|
 256 -> 512 :  664       255206              |#####|
 512 -> 1024 :  47       36249               |#|
1024 -> 2048 :  3        5529                |#|
2048 -> 4096 :  4        13209               |#|
```

https://github.com/LucaCanali/Linux_tracing_scripts
**Slow I/O - Little CPU Usage**

- Slow down the storage on the VM by flushing cache on host
  
  ```
  # free && sync && echo 3 > /proc/sys/vm/drop_caches && free
  ```

- Run SLOB workload again
  - Storage serves I/O at high latency and delivers only 300 IOPS

**Top 10 Foreground Events by Total Wait Time**

<table>
<thead>
<tr>
<th>Event</th>
<th>Waits</th>
<th>Total Wait Time (sec)</th>
<th>Wait Avg (ms)</th>
<th>% DB time</th>
<th>Wait Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>db file parallel read</td>
<td>5,455</td>
<td>900</td>
<td>164.95</td>
<td>89.6 User I/O</td>
<td></td>
</tr>
<tr>
<td>db file sequential read</td>
<td>17,206</td>
<td>94.1</td>
<td>5.47</td>
<td>9.4 User I/O</td>
<td></td>
</tr>
<tr>
<td>DB CPU</td>
<td>32.9</td>
<td>32.9</td>
<td>3.3</td>
<td>3.3 System I/O</td>
<td></td>
</tr>
<tr>
<td>control file sequential read</td>
<td>164</td>
<td>.7</td>
<td>4.44</td>
<td>1.1 System I/O</td>
<td></td>
</tr>
<tr>
<td>Disk file operations I/O</td>
<td>31</td>
<td>.2</td>
<td>7.60</td>
<td>.0 User I/O</td>
<td></td>
</tr>
<tr>
<td>SQL*Net break/reset to client</td>
<td>2</td>
<td>.1</td>
<td>70.74</td>
<td>.0 Application</td>
<td></td>
</tr>
<tr>
<td>log file sync</td>
<td>4</td>
<td>0</td>
<td>7.63</td>
<td>.0 Commit</td>
<td></td>
</tr>
<tr>
<td>CSS initialization</td>
<td>1</td>
<td>0</td>
<td>30.45</td>
<td>.0 Other</td>
<td></td>
</tr>
<tr>
<td>SQL*Net more data from client</td>
<td>26</td>
<td>0</td>
<td>0.33</td>
<td>.0 Network</td>
<td></td>
</tr>
<tr>
<td>direct path write</td>
<td>1</td>
<td>0</td>
<td>8.46</td>
<td>.0 User I/O</td>
<td></td>
</tr>
</tbody>
</table>
Flame Graph and Analysis

- The case of slow I/O

- Most of the time spent in sleep mode
  - Waiting to reap asynchronous I/O: io_getevents

- Compare:
  - In the case of low-latency/high IOPS most time was spent on io_submit, as io_getevents was very fast
Something about Async I/O

- Asynchronous I/O io_submit and io_getevents:
- Flame graphs show where most time is spent
- Tracing shows all actions. Example with SystemTap:

```
root@01.29.6:/#systemtap

stap -v trace_oracle_logicalio_wait_events_physicalio_12102.stp -x 17033|sed -f eventsname.sed

DB LOGICAL IO Consistent Read (kcbgtrc) for block: tbs##7, rfile##0, block##29966, obj##32173

DB LOGICAL IO Consistent Read (kcbgtrc) for block: tbs##7, rfile##0, block##91959, obj##32173

--

DB WAIT EVENT BEGIN: timestamp_ora=5928720459249, pid=18931, sid=7, event-db file parallel read

DB WAIT EVENT BEGIN: timestamp_ora=5928720459249, pid=18931, sid=7, event-db file parallel read

--

DB WAIT EVENT BEGIN: timestamp_ora=5928720459249, pid=18931, sid=7, event-db file parallel read

```

```bash
stap -v trace_oracle_logicalio_wait_events_physicalio_12102.stp -x 17033|sed -f eventsname.sed

DB LOGICAL IO Consistent Read (kcbgtrc) for block: tbs##7, rfile##0, block##29966, obj##32173

DB LOGICAL IO Consistent Read (kcbgtrc) for block: tbs##7, rfile##0, block##91959, obj##32173

--

DB WAIT EVENT BEGIN: timestamp_ora=5928720459249, pid=18931, sid=7, event-db file parallel read

```
Perf

- Perf_events is a Linux profiler tool for
  - performance counters (PCL)
  - events observer (LPE)

- Integrated into the kernel
  - Available for kernel versions $\geq 2.6.31$
  - E.g. available on RHEL/OL 6 and 7.

- Safe to use on production systems
On-CPU Flame Graph and Stack Profiling with Perf

1. Collect on-CPU stack of the Oracle process under investigation

   $ perf record -a -g -F 99 -p <pid of process> sleep 10

   99 Hz is a starting point for the sampling frequency
   Default event: cpu-cycles on physical machines and cpu-clock on VMs

2. Dump stack traces in a text file

   $ perf script > myperf_script.txt

3. Get scripts: https://github.com/brendangregg/FlameGraph

4. Create a flame graph

   cat myperf_script.txt | 
   ../FlameGraph/stackcollapse-perf.pl | 
   ../FlameGraph/flamegraph.pl --title "My Flame Graph"
Example 1/3: parsing with rule based optimization

Data collected with perf
Sampling rate 100 KHZ: option -F 100000
Navigating Oracle Functions Names

- Not documented, but some hints at:
  - “ORA-600 Lookup Error Categories” (formerly 175982.1, now at Doc ID 1321720.1)
  - Search on MOS or Google for it
  - Understand from context, educated guess
- A few Oracle kernel modules exposed with oradebug
  - SQL> oradebug doc COMPONENT
- Examples:
  - kks prefix -> support for managing shared cursors/ shared sql
  - kksLoadChild -> related to hard parse
  - kksParseChildCursor -> soft parse
  - 'apa' (Access Path Analysis?), 'kko' (Kernel Kompile Optimizer?), 'kkq' (Kernel Kompile Query?)
Example 2/3: cost based optimization

Data collected with perf

Sampling rate: -F 100000
Example 3/3:

12c with adaptive_optimizer_features=TRUE

Data collected with perf

High rate sampling: -F 100000
Flame Graph: Internals Investigations

- New function in 12.1.0.2: `approx_count_distinct`
- **Scalable** (no need for large sort) cardinality estimation
- Implementation of the Hyper Log Log algorithm?
Execution Plans and QER Functions

- Functions with “qer” prefix
  - Query Execution Rowsource
  - Expose the execution plan steps
  - Example from os_explain by Tanel Poder

```
s/qerim/INDEX MAINTENANCE: /g;
s/qerix/INDEX: /g;
s/qerjot/NESTED LOOP JOIN: /g;
s/qerjo/NESTED LOOP OUTER: /g;
s/qerle/LINEAR EXECUTION IMPLEMENTATION: /g;
s/qerli/PARALLEL CREATE INDEX: /g;
s/qerlt/LOAD TABLE: /g;
s/qerns/GROUP BY NO SORT: /g;
...```
Additional Steps for Oracle Flame Graphs

4a) Extract `sed` commands from `os_explain` script (by Tanel Poder)

```
wget http://blog.tanelpoder.com/files/scripts/tools/unix/os_explain
grep "s\\/q" os_explain > os_explain.sed
```

4b) Create the flame graph using `os_explain` mapping for rowsource functions

```
cat myperf_script.txt| sed -f os_explain.sed| ../FlameGraph/stackcollapse-perf.pl | ../FlameGraph/flamegraph.pl --title "My Flame Grapg" >Figure1.svg
```
Filtered Flame Graphs

- Create flame graph for query execution operations and functions that start with ‘k’:

```bash
grep -i -e qer -e opifch -e 'k' -e ^$ myperf_script.txt| sed -f os_explain.sed| ../FlameGraph/stackcollapse-perf.pl| ../FlameGraph/flamegraph.pl >Figure2.svg
```

Flame Graph: SLOB logical IO - filtered

kcbgtrc = Kernel Cache Buffers
Get Consistent Read

```
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>E-Rows</th>
<th>E-Bytes</th>
<th>Cost (%CPU)</th>
<th>E-Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* 2</td>
<td>FILTER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>CF1</td>
<td>256</td>
<td>34304</td>
<td>260 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>* 4</td>
<td>INDEX RANGE SCAN</td>
<td>L_CF1</td>
<td>256</td>
<td>3</td>
<td>00:00:01</td>
<td></td>
</tr>
</tbody>
</table>
```
Flame Graphs of Execution Plans

Step 11: Index Fast Full Scan of I_SOURCE1

SQL:
```
select count(*) from dba_source
```
 Queries on Cached Data

- Trend -> servers with large amounts of memory
  - Logical I/O becomes critical: “RAM is the new disk”
  - .. but without wait events
- Example query and data:
  - select cust_city, avg(cust_year_of_birth) from Customers group by cust_city;
  - Customers table: 911 M rows, 23 columns, 21M 8K blocks, cached in buffer cache
Query From the Buffer Cache

- Time: **287** sec, ~50% on “group by”. 21M blocks

Example query from buffer cache:
```
SELECT *+ NO_INMEMORY */ CUST_CITY, AVG(CUST_YEAR_OF_BIRTH) FROM CUSTOMERS GROUP BY CUST_CITY
```

<table>
<thead>
<tr>
<th>Rows</th>
<th>Row Source Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>502</td>
<td>HASH GROUP BY (cr=21024838 pr=0 time=287 sec ..)</td>
</tr>
<tr>
<td>910595840</td>
<td>TABLE ACCESS FULL CUSTOMERS (cr=21024838 pr=0 time=161 sec)</td>
</tr>
</tbody>
</table>
Perf Top vs. Stack Profiling

- Perf top – fast and easy
  - Explore real-time the surface of the flame graph
  - Drill down with stack if needed

```bash
$ perf top [-p <pid of process> ]
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Percentage</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>oracle</td>
<td>24.93%</td>
<td>kdstf00001010000000km</td>
</tr>
<tr>
<td>oracle</td>
<td>8.95%</td>
<td>qeshrGetUHash_Fast</td>
</tr>
<tr>
<td>oracle</td>
<td>7.46%</td>
<td>kaf4reasrp0km</td>
</tr>
<tr>
<td>oracle</td>
<td>6.56%</td>
<td>qesaOSBuildWA</td>
</tr>
<tr>
<td>oracle</td>
<td>6.44%</td>
<td>kcbgtcr</td>
</tr>
<tr>
<td>oracle</td>
<td>5.78%</td>
<td>qeshLoadRowForGBY</td>
</tr>
<tr>
<td>oracle</td>
<td>4.64%</td>
<td>qeshrPUCmpare</td>
</tr>
<tr>
<td>oracle</td>
<td>4.61%</td>
<td>sslnxsom</td>
</tr>
<tr>
<td>oracle</td>
<td>4.57%</td>
<td>qeaAvg</td>
</tr>
<tr>
<td>oracle</td>
<td>3.79%</td>
<td>qerghAggregateRecords</td>
</tr>
<tr>
<td>oracle</td>
<td>3.32%</td>
<td>qerghRowP</td>
</tr>
<tr>
<td>oracle</td>
<td>3.20%</td>
<td>qeshIHPProbeURow_Simple</td>
</tr>
<tr>
<td>oracle</td>
<td>2.67%</td>
<td>qesaMergeNonDist_Impl</td>
</tr>
<tr>
<td>oracle</td>
<td>2.59%</td>
<td>qesaSimpleAggNonDist</td>
</tr>
<tr>
<td>oracle</td>
<td>1.80%</td>
<td>qeshrGotoWorkArea</td>
</tr>
</tbody>
</table>
In-Memory Column Store

- Query: 193 sec (46 GB). IM more efficient for scan, query time dominated by “hash group by”

Example query from inmemory store:

```sql
SELECT /*+ inmemory */ cust_city, AVG(cust_year_of_birth) FROM CUSTOMERS GROUP BY cust_city;
```
Faster: Min and In-Memory

- Query: 10 sec, hash group for min is optimized

Example query from in-memory store: 
```
SELECT /*+ inmemory */ cust_city, MIN(cust_year_of_birth) FROM CUSTOMERS GROUP BY cust_city
```
PL/SQL Flame Graphs

- Data source: PL/SQL Hierarchical Profiler (DBMS_HPROF, 11g and higher)

- Martin Buechi: *OOW 2015, CON2082: Best Practices for Interpreting PL/SQL Hierarchical Profiles for Effective Tuning*

- Follow the link to download ora_hprof# package (implemented in Java). Example of the output:
Example: PL/SQL Flame Graph from DBMS_HPROF Profiles

- Gather profiler data:
  ```sql
  CREATE OR REPLACE DIRECTORY profiler_dir AS '<path>';
  exec DBMS_HPROF.start_profiling (location => 'PROFILER_DIR', filename => 'profiler.txt')
  -- Run here the PL/SQL workload to trace
  exec DBMS_HPROF.stop_profiling
  ```

- Generate stack collapsed data
- This step can be done on separate (test) machine
  ```sql
  exec ora_hprof#.FLATTEN('PROFILER_DIR', 'profiler.txt', 'plsql_hprofstack.txt')
  ```

```
$ FlameGraph/flamegraph.pl /<path>/plsql_hprofstack.txt --title 'PL/SQL Flame Graph' > plsql_hprofstack.svg
```
Some Ideas for Improvements of the Techniques and Tools

- On-off CPU stack profiler
  - Improve on Ptrace_Profile
  - more lightweight on userspace stack
- Generic PL/SQL stack tracing (beyond hprof)
- More extended stack profilers tools
  - Generic fields from X$KSUSE?
  - More Oracle-related info from SGA
  - More OS info from /proc
Acknowledgements and References

- Brendan Gregg -> Flame Graphs and more
- Tanel Poder -> Oracle internals, including previous original work on stack profiling
- Frits Hoogland -> Oracle userspace investigations, perf, gdb
- Kevin Clossen -> SLOB, Linux perf for Oracle
- Stefan Koehler, paper and talk at DOAG 2015
- Additional references:
  - http://db-blog.web.cern.ch
  - http://externaltable.blogspot.com
  - https://github.com/LucaCanali
Conclusions

• Effective troubleshooting starts with understanding
  • What the system is doing, where are the bottlenecks
• Oracle instrumentation helps in most cases
  • OS tools complement and extend Oracle data
  • Notably for CPU-bound workload
• Add Stack Profiling and Flame Graph to your toolbox and.. share your findings!