Modern Linux Tools for Oracle Troubleshooting

Luca Canali, CERN
Zbigniew Baranowski, CERN

UKOUG TECH14, Liverpool, December 2014
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
About CERN

- CERN - European Laboratory for Particle Physics
- Founded in 1954 by 12 countries for fundamental physics research in a post-war Europe
- Today 21 member states + world-wide collaborations
  - About ~1000 MCHF yearly budget
  - 2’300 CERN personnel + 10’000 users from 110 countries
LHC is the world’s largest particle accelerator

- LHC = Large Hadron Collider
- 27km ring of superconducting magnets
- Currently undergoing upgrades, restart in 2015
From particle to article..

Higgs boson-like particle discovery claimed at LHC

By Paul Rincon
Science editor, BBC News website, Geneva

Cern scientists reporting from the Large Hadron Collider (LHC) have claimed the discovery of a new particle consistent with the Higgs boson.
This talk covers Linux tools for advanced tracing and Oracle investigations

- Modern tools opening new roads: Perf, DTrace, Systemtap, ...
- Probes on the OS to understand system calls
- Probes for the userspace to understand Oracle internals

- This is a short exploration rather than a lesson
- Focus on techniques that you can use today
Prerequisites

• Not covered here are many of the common and most used tools and utilities
  • top
  • strace
  • vmstat
  • iostat, sar, collectl, dstat
  • use of the /proc filesystem (ex cat /proc/meminfo)
  • ...
You can probably run userspace and dynamic tracing tools in Linux already

- Available with RHEL/OEL 6 or higher
- You gain:
  - Tools for advanced troubleshooting
  - Insights in the working of Oracle and OS
Do I need dynamic tracing when I have Oracle wait events?

• Data beyond wait events
  • Instrument the latency details of ‘on CPU’ time
  • Many Oracle wait events don’t provide good measurements of I/O latency

• Wait event data + user space tracing + OS tracing = new insights
About Zbigniew

• since 2009 at CERN
  • Developer
  • Researcher
  • DBA

• DBA with > 5 years of experience

• Already presented twice at UKOUG
  • about Streams and GoldenGate
Long Running Query -> on CPU?

- Enterprise Manager

- OS

- Execution plan
Snapper Can Help

Wait event interface not much useful in this case

---

-- Session Snapper v3.10 by Tanel Poder @ E2SN (http://tech.e2sn.com)

<table>
<thead>
<tr>
<th>SID, USERNAME</th>
<th>TYPE, STATISTIC</th>
<th>DELTA, HDELTA/SEC</th>
<th>%TIME, GRAPH</th>
</tr>
</thead>
<tbody>
<tr>
<td>2416, SYS</td>
<td>STAT, session logical reads</td>
<td>1130, 113</td>
<td></td>
</tr>
<tr>
<td>2416, SYS</td>
<td>STAT, consistent gets</td>
<td>1130, 113</td>
<td></td>
</tr>
<tr>
<td>2416, SYS</td>
<td>STAT, consistent gets from cache</td>
<td>1130, 113</td>
<td></td>
</tr>
<tr>
<td>2416, SYS</td>
<td>STAT, consistent gets from cache (fastpath)</td>
<td>1130, 113</td>
<td></td>
</tr>
<tr>
<td>2416, SYS</td>
<td>STAT, logical read bytes from cache</td>
<td>9256960, 925.7k</td>
<td></td>
</tr>
<tr>
<td>2416, SYS</td>
<td>STAT, no work - consistent read gets</td>
<td>1130, 113</td>
<td></td>
</tr>
<tr>
<td>2416, SYS</td>
<td>STAT, table scan rows gotten</td>
<td>172892, 17.29k</td>
<td></td>
</tr>
<tr>
<td>2416, SYS</td>
<td>STAT, table scan blocks gotten</td>
<td>1130, 113</td>
<td></td>
</tr>
<tr>
<td>2416, SYS</td>
<td>TIME, PL/SQL execution elapsed time</td>
<td>9422513, 942.25ms</td>
<td>94.2%, [9999</td>
</tr>
<tr>
<td>2416, SYS</td>
<td>TIME, DB CPU</td>
<td>9991481, 999.15ms</td>
<td>99.9%, [9999</td>
</tr>
<tr>
<td>2416, SYS</td>
<td>TIME, sql execute elapsed time</td>
<td>9999279, 999.93ms</td>
<td>100.0%, [9999</td>
</tr>
<tr>
<td>2416, SYS</td>
<td>TIME, DB time</td>
<td>9999279, 999.93ms</td>
<td>100.0%, [9999</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Active%</th>
<th>SQL_ID</th>
<th>EVENT</th>
<th>WAIT_CLASS</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td>dhtawwt478tg</td>
<td>ON CPU</td>
<td>ON CPU</td>
</tr>
</tbody>
</table>

---

PL/SQL?

Reading from db cache
SQL Monitor

- Buffer gets

- PL/SQL almost all the time

- 100% activity on the hash join

Can we get more? Lets try with perf
Perf

• Linux profiler tool for
  • performance counters (PCL)
  • events observer (LPE)

• Integrated into the kernel
  • Available for kernel versions >= 2.6.31 (RHEL6)

• Safe to use on production systems
## Live view of top active functions

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

<table>
<thead>
<tr>
<th>%</th>
<th>Function</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>38.56%</td>
<td>oracle</td>
<td>[] lnxdiv</td>
</tr>
<tr>
<td>12.06%</td>
<td>oracle</td>
<td>[] lnxadd</td>
</tr>
<tr>
<td>9.25%</td>
<td>oracle</td>
<td>[] lnxmul</td>
</tr>
<tr>
<td>7.07%</td>
<td>oracle</td>
<td>[] lnxsub</td>
</tr>
<tr>
<td>3.70%</td>
<td>oracle</td>
<td>[] lnxmin</td>
</tr>
<tr>
<td>3.51%</td>
<td>oracle</td>
<td>[] pevm_icd_call_common</td>
</tr>
<tr>
<td>2.29%</td>
<td>libc-2.12.so</td>
<td>memmove</td>
</tr>
<tr>
<td>2.24%</td>
<td>oracle</td>
<td>[] pfrinstr_BCTR</td>
</tr>
<tr>
<td>1.65%</td>
<td>oracle</td>
<td>[] pfrinstr_ADDN</td>
</tr>
<tr>
<td>1.64%</td>
<td>oracle</td>
<td>[] pfrinstr_CVTIN</td>
</tr>
<tr>
<td>1.54%</td>
<td>oracle</td>
<td>[] pfrrun_no_tool</td>
</tr>
<tr>
<td>1.53%</td>
<td>oracle</td>
<td>[] pfrinstr_MULN</td>
</tr>
<tr>
<td>1.48%</td>
<td>oracle</td>
<td>[] pfrinstr_DIVN</td>
</tr>
<tr>
<td>1.21%</td>
<td>oracle</td>
<td>[] pesmod</td>
</tr>
<tr>
<td>1.03%</td>
<td>oracle</td>
<td>[] lnxtru</td>
</tr>
<tr>
<td>1.01%</td>
<td>oracle</td>
<td>[] pisonu</td>
</tr>
<tr>
<td>0.97%</td>
<td>oracle</td>
<td>[] lnxmod</td>
</tr>
<tr>
<td>0.72%</td>
<td>libc-2.12.so</td>
<td>__sigsetjmp</td>
</tr>
<tr>
<td>0.71%</td>
<td>oracle</td>
<td>[] pfrinstr_MOVAN</td>
</tr>
<tr>
<td>0.69%</td>
<td>oracle</td>
<td>[] peginu</td>
</tr>
<tr>
<td>0.67%</td>
<td>oracle</td>
<td>[] pfrinstr_BCAL</td>
</tr>
<tr>
<td>0.65%</td>
<td>oracle</td>
<td>[] __intel_new_memcpy</td>
</tr>
<tr>
<td>0.56%</td>
<td>oracle</td>
<td>[] lnxcopy</td>
</tr>
<tr>
<td>0.56%</td>
<td>oracle</td>
<td>[] __intel_fast_memcpy</td>
</tr>
</tbody>
</table>

Samples: 169K of event 'cycles', Event count (approx.): 36226334492
What are those Oracle functions?

- Complete description of the functions called by Oracle with is not officially published, but…

- Google it or just guess ;)

- Backups of some MOS notes can be handy
  - "ORA-600 Lookup Error Categories" (formerly 175982.1)

- For actions which are part of query execution
What have we learned so far?

- Our sql is running some arithmetic operations:
  - function \texttt{lnxdiv} (38\%) => division
  - function \texttt{lnxadd} (10\%) => addition
  - function \texttt{lnxmul} (9\%) => multiplication

- Is it all the time like that?
- Why (by whom) they are called?
Recording Samples with Perf

- **Function** currently being executed sampling
  
  ```
  perf record [-p <pid of process>] [-F <frequency> ]
  ```

- **Full stack** sampling
  
  ```
  perf record -g -p <pid of process> [-F <frequency> ]
  ```

- Be careful with the sampling **frequency**
  
  - 99Hz is reasonable

- Samples are recorded to a **binary file** `perf.data`
Displaying Recoded Data

• In human readable format (same as top)

<table>
<thead>
<tr>
<th>Function</th>
<th>Time Percentage</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>inxdiv</td>
<td>40.92%</td>
<td>oracle</td>
</tr>
<tr>
<td>lnxadd</td>
<td>10.77%</td>
<td>oracle</td>
</tr>
<tr>
<td>lnxmul</td>
<td>9.67%</td>
<td>oracle</td>
</tr>
<tr>
<td>lnxsub</td>
<td>5.45%</td>
<td>oracle</td>
</tr>
<tr>
<td>lnxmin</td>
<td>3.82%</td>
<td>oracle</td>
</tr>
<tr>
<td>memmove</td>
<td>3.25%</td>
<td>oracle</td>
</tr>
<tr>
<td>pfrinstr_BCTR</td>
<td>2.45%</td>
<td>libc-2.12.so</td>
</tr>
<tr>
<td>pfrinstr_NO_TOOL</td>
<td>1.59%</td>
<td>oracle</td>
</tr>
<tr>
<td>pfrinstr_MULN</td>
<td>1.58%</td>
<td>oracle</td>
</tr>
<tr>
<td>pfrinstr_CVTIN</td>
<td>1.52%</td>
<td>oracle</td>
</tr>
<tr>
<td>pfrinstr_DIVN</td>
<td>1.42%</td>
<td>oracle</td>
</tr>
<tr>
<td>pfrinstr_ADDN</td>
<td>1.42%</td>
<td>oracle</td>
</tr>
<tr>
<td>pesmod</td>
<td>1.23%</td>
<td>oracle</td>
</tr>
<tr>
<td>pisonu</td>
<td>1.21%</td>
<td>oracle</td>
</tr>
<tr>
<td>lnxmod</td>
<td>1.14%</td>
<td>oracle</td>
</tr>
<tr>
<td>lnxtru</td>
<td>1.06%</td>
<td>oracle</td>
</tr>
<tr>
<td>pfrinstr_MOVAN</td>
<td>0.76%</td>
<td>oracle</td>
</tr>
<tr>
<td>__intel_new_memcpy</td>
<td>0.71%</td>
<td>oracle</td>
</tr>
<tr>
<td>pfrinstr_BCAL</td>
<td>0.66%</td>
<td>oracle</td>
</tr>
<tr>
<td>__sigsetjmp</td>
<td>0.63%</td>
<td>libc-2.12.so</td>
</tr>
<tr>
<td>__intel_fast_memcpy</td>
<td>0.56%</td>
<td>oracle</td>
</tr>
<tr>
<td>lnxcopy</td>
<td>0.52%</td>
<td>oracle</td>
</tr>
</tbody>
</table>
Displaying Recorded Stacks

- Tree format

Not easy to read!

There is a way of making stack samples easier to read…
Flame Graphs

• Visualization of stack samples

• Author: http://www.brendangregg.com/
How to create a flame graph

1. Collect stack samples of our process under investigation

```bash
perf record -a -g -F99 -p <pid of process>
```

2. Dumpstack traces in a text file

```bash
perf script > myperf_script.txt
```

3. Get scripts: [https://github.com/brendangregg/FlameGraph](https://github.com/brendangregg/FlameGraph)

4. Create a flame graph

```bash
grep -v 'cycles:' myperf_script.txt | ./FlameGraph-master/stackcollapse-perf.pl | ./FlameGraph-master/flamegraph.pl --title "My graph"
```
Flame Graph for our SQL

- Is called lnxdiv in at least 2 different places
4a) Extract `sed` commands from `os_explain` script (by Tanel Poder)

```bash
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

```bash
grep -v 'cycles:' myperf_script.txt | sed -f os_explain.sed | ../FlameGraph-master/stackcollapse-perf.pl | ../FlameGraph-master/flamegraph.pl --title "My FG" >Figure1.svg
```
Flame Graph for our SQL

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>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 ORDER BY</td>
<td></td>
<td>296M</td>
<td>10G</td>
<td>42899 (100)</td>
<td>00:08:35</td>
</tr>
<tr>
<td>2</td>
<td>HASH GROUP BY</td>
<td></td>
<td>296M</td>
<td>10G</td>
<td>42899 (61)</td>
<td>00:08:35</td>
</tr>
<tr>
<td>3</td>
<td>HASH JOIN</td>
<td></td>
<td>296M</td>
<td>10G</td>
<td>18028 (6)</td>
<td>00:03:37</td>
</tr>
<tr>
<td>4</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>EVENTHISTORY_TEST</td>
<td>2500</td>
<td>65000</td>
<td>337</td>
<td>00:00:05</td>
</tr>
<tr>
<td>5</td>
<td>INDEX RANGE SCAN</td>
<td>EVENTHISTORY_TEST_TS</td>
<td>2500</td>
<td></td>
<td>9</td>
<td>00:00:01</td>
</tr>
<tr>
<td>6</td>
<td>TABLE ACCESS FULL</td>
<td>EVENTHISTORY_TEST</td>
<td>11M</td>
<td>146M</td>
<td>16710 (1)</td>
<td>00:03:21</td>
</tr>
</tbody>
</table>
FG for an Execution Plan

• Create flame graph for query execution operations only:

grep -i -e qer -e opifch -e ^$ myperf_script.txt | sed -f os_explain.sed | ..FlameGraph-master/stackcollapse-perf.pl | ..FlameGraph-master/flamegraph.pl --title "Flame Graph Rowsource: my select" >Figure2.svg

Flame Graph: Execution plan

Function: SORT: Fetch (56,767 samples, 100.00%)
What was the join condition of the query?

- \texttt{compute(range\_scan.VALUE\_NUMBER,1000)} = \texttt{compute(full\_table.VALUE\_NUMBER,100)}

Predicate Information (identified by operation id):
-----------------------------------------------
3 - access("COMPUTE"("T1"."VALUE\_NUMBER",1000)="COMPUTE"("T2"."VALUE\_NUMBER",100))
5 - access("T1"."TS">=TO\_TIMESTAMP('01-NOV-09 04.06.44.759000000 PM') AND "T1"."TS"<=TO\_TIMESTAMP('20-NOV-09 10.06.44.759000000 PM'))

create function \texttt{compute} (val in number, j number) return varchar2 as
  ret number:=0;
begin
  FOR i IN 1..j loop
    ret:=ret + \texttt{mod}(val * i,100) / i;
  end loop;
return ret;
end;
FG for Server Profiling

- Entire server workload captured from 20 sec
Perf & Flame Graphs: Summary

- **Perf**
  - user space exploration
  - available >RHEL 6
  - other useful features (events tracing and probes)

- **Flame graph**
  - call stack visualization

- **Perf + flame graph**
  - Performance investigation
    - When wait-event interface does not deliver relevant information – CPU intensive processing
Advanced Tracing for Linux

- Solaris has DTrace since 2005, Linux is catching up
- Currently many tools available
  - Oracle Linux DTrace, Dtrace4linux, SystemTap, perf_events, ftrace, ktap, LTTng, eBPF, sysdig
- Most of them still in development
DTrace and Linux

- DTrace license is **CDDL**, incompatible with GPL
- There are 2 ports of DTrace for Linux
  - Both still in active development
  - Oracle’s port for OEL (for **ULN** subscribers)
    - Notably it does not yet have userspace tracing with the ‘pid provider’
  - ‘dtrace4linux’: a one-person effort
    - unstable but with more functionality
How to Measure Latency with Dynamic Tracing

The main ingredients:

• Trigger execution probe at the start of a system call (or a users function)
• Run a probe at the return from the call
• Measure the elapsed time
• Aggregate data in a latency histogram
An Example with DTrace

• Measure latency histogram of pread64 calls
• Note: IOPS and latency of random reads very important for troubleshooting OLTP performance

```
# dtrace -n ' 

syscall::pread64:entry { self->s = timestamp; }

syscall::pread64:return /self->s/ { 
@pread["ns"] = quantize(timestamp -self->s);
self->s = 0;
}

tick-10s {
printa(@pread);
trunc(@pread);
}
'
SystemTap

- Backed by Red Hat, started in 2005
  - Version 1.0 in 2009
- Works by compiling and loading kernel modules
- Scripting language similar to C, allows adding C extensions
- Easy to start working with it:
  - Look at example probes and build from there
  - Many similarities between DTrace and SystemTap probes
SystemTap Userspace Probes

- Probes into executable processes (**userspace**)
  - Read function arguments
  - Read from process **memory** (ex: SGA and PGA)
- Linux support
  - **UTRACE** -> available with SystemTap also in RHEL6
  - **UPROBES** -> replace UTRACE for kernel version from 3.5, available with SystemTap and more tools
  - Dtrace4linux can also do userspace tracing
How to check if userspace tracing is available/active on your system

• This is how to check if UTRACE extensions are configured:
  
  ```bash
  # grep CONFIG_UTRACE /boot/config-`uname -r`
  CONFIG_UTRACE=y
  ```

• This is how to check if UPROBES are available:
  
  ```bash
  # grep CONFIG_UPROB /boot/config-`uname -r`
  CONFIG_UPROBES=y
  CONFIG_UPROBE_EVENT=y
  ```
## Key functions to probe the Oracle wait event interface

<table>
<thead>
<tr>
<th>Function name</th>
<th>Purpose</th>
<th>Selected parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>KSKTHBWT</strong></td>
<td>Kernel Service Kompile Thread Begin Wait. This function is called at the start of an Oracle wait event. The suffix &quot;bwt&quot; most likely stands for &quot;begin wait&quot;. kslwtbctx is its parent function call and marks the start of a wait event.</td>
<td>register r13 -&gt; points into X$KSUSE (V$SESSION) SGA segmented array register rsi -&gt; timestamp of the beginning of the wait (in microseconds) register rdx -&gt; wait event number</td>
</tr>
<tr>
<td><strong>KSKTHEWT</strong></td>
<td>Kernel Service Kompile Thread End Wait. This function is called at the end of an Oracle wait event. The suffix &quot;ewt&quot; most likely stands for &quot;end wait&quot;. kslwtectx is its parent function call marking the end of a wait event.</td>
<td>register r13 -&gt; points into X$KSUSE (V$SESSION) SGA segmented array register rdi -&gt; timestamp of the beginning of the wait (in microseconds) register rsi -&gt; wait event number</td>
</tr>
</tbody>
</table>
Systemtap can read from the Oracle wait event interface

Example: how to write a probe tracing the beginning of each wait event:

```plaintext
probe process("oracle").function("kskthbwt") {

    xksuse = register("r13")-3928 # offset for 12.1.0.2
    ksusenum = user_uint16(xksuse + 1704)

    printf("DB WAIT EVENT BEGIN: timestamp_ora=%ld, pid=%d, sid=%d, event#=%u\n", register("rsi"), pid(), ksusenum, register("rdx"))
}
```
How to read X$KSUSE from SGA

- X$KSUSE -> underlying table of V$SESSION
  - It’s a segmented array
  - Base of the array record: from CPU register R13
    - With offset that is version-dependent
  - The column offsets (record values) are available by querying X$KQFCO and X$KQFTA
  - Records contain info on: userid, sql_hash, wait elapsed time, …
Example: How to collect wait event histograms with microsec resolution

- V$EVENT_HISTOGRAM useful to study latency
  - However only milisec precision, a problem when studying SSD latency
  - Note 12.1.0.2 has V$EVENT_HISTOGRAM_MICRO
- Solution: userspace tracing of Oracle processes
  - Provides way to collect and display microsec-precision histograms for all Oracle versions
  - Capture event# and wait time in microseconds
  - Collect data in a SystemTap aggregate
  - Print output as a histogram
Example of wait event histograms collected with SystemTap

```plaintext
# stap -v histograms_oracle_events_11204.stp -x <pid>
# Note: omit -x to trace all oracle processes

Histogram of db file sequential read waits in microseconds (us):

<table>
<thead>
<tr>
<th>value</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>33</td>
</tr>
<tr>
<td>256</td>
<td>60</td>
</tr>
<tr>
<td>512</td>
<td>61</td>
</tr>
<tr>
<td>1024</td>
<td>93</td>
</tr>
<tr>
<td>2048</td>
<td>260</td>
</tr>
<tr>
<td>4096</td>
<td>951</td>
</tr>
<tr>
<td>8192</td>
<td>538</td>
</tr>
<tr>
<td>16384</td>
<td>47</td>
</tr>
<tr>
<td>32768</td>
<td>71</td>
</tr>
<tr>
<td>65536</td>
<td>34</td>
</tr>
<tr>
<td>131072</td>
<td>153</td>
</tr>
<tr>
<td>262144</td>
<td>62</td>
</tr>
<tr>
<td>524288</td>
<td>16</td>
</tr>
</tbody>
</table>
```
SystemTap Probes for Oracle Logical and Physical I/O

Identify the Oracle internal functions of interest:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| kcbgtrc  | Kernel Cache Buffers Get Consistent Read  
           | Used for consistent reads |
| kcbgcur  | Kernel Cache Buffers Current Read  
           | Used for current reads |
| kcbzib   | kcbZIB should stand for: Z (kcbz.o is a module for physical IO helper functions), IB: Input Buffer  
           | Oracle will perform physical read(s) into the buffer cache |
| kcbzgb   | The suffix GB in kcbZGB should stand for: Get (space for) Buffer. Oracle allocates space in the buffer cache for a given block (typically before I/O operations). |
| kcbzvb   | Invoked after Oracle has performed I/O to read a given block  
           | Note: this function is used both for reads in the buffer cache and for direct reads |
Find the key function call parameters and their meaning

- Identify the function parameters of interest (block number, file number, etc)
- Example for `kcbgtdcr` and `kcbgcur`

```
tbs# = user_int32(%rdi)
rel file n# = user_int32(%rdi+4) >> 22 & 0x003FFFFF
block# = user_int32(%rdi+4) & 0x003FFFFF
data_object_id# = user_int32(%rdi+8)
object_id# = user_int32(%rdi+12)
```

Note: for bigfile tablespaces: `block# = user_int32(%rdi+4)`
Putting it all together: Trace wait events + logical and physical I/O

- Provide insights on how Oracle does the I/O
- What are the I/O-related wait events really measuring?
- Can we rely on the measurements of wait elapsed time to understand I/O latency?

Trace:

```
# stap -v
trace_oracle_logicalio_wait_events_physicalio_12102.stp
-x <pid> | sed -f eventsname.sed
```
Example of tracing ‘db file sequential read’ wait event

======
DB LOGICAL IO Consistent Read (kcbgtcr) for block: tbs#=7, rfile#=0, block#=2505675, obj#=32174
  ->kcbzib, Oracle logical read operations require physical reads into the buffer cache
  -> kcbzgb, Oracle has allocated buffer cache space for block: tbs#=7, rfile#=0, block#=2505675, obj#=32174
======
DB WAIT EVENT BEGIN: timestamp_ora=498893930487, pid=15559, sid=21, event=db file sequential read

OS: ->pread: timestamp=498893930555, program=oracle_15559_or, pid=15559, fd=264, offset=83048882176, count(bytes)=8192
OS:  ->ioblock.request, timestamp=498893930588, pid=15559, devname=sdl, sector=162204848, size=8192, rw=0, address_bio=18446612144946364800
OS:  <-ioblock.end, timestamp=498893930588, pid=0, devname=sdl, sector=162204864, rw=0, address_bio=18446612144946364800
OS: <-pread: timestamp=498893934592, program=oracle_15559_or, local_clock_us(), pid=15559, return(bytes)=8192

DB WAIT EVENT END: timestamp_ora=498893934633, pid=15559, sid=21, name=SYSTEM, event=db file sequential read, p1=7, p2=2505675, p3=1, wait_time=4146, obj=32172, sql_hash=964615745

======
  ->kcbzvb, Oracle has performed I/O on: file#=7, block#=2505675, rfile#=0
======
What the trace shows about ‘db file sequential read’

- Oracle starts with a **logical I/O**
- If the block is not in the buffer cache a physical read is initiated
  - A block in the buffer cache is allocated
  - The wait event **db file sequential read** is started
- Oracle calls `pread` to read 8KB
  - This passed on to the block I/O interface
- After the read is done, the wait event ends
- Comment on the wait time: **db file sequential read** is dominated by synchronous I/O wait time
The Case of Direct Reads and Tracing Oracle Asynchronous I/O

- Asynchronous I/O is used by Oracle to optimize I/O throughput
  - OS calls used: IO_SUBMIT and IO_GETEVENTS
  - We consider the case of ASM on block devices

Findings:
- Oracle can perform reads that are not instrumented by the wait event interface
- The wait event ‘direct path read’, does not instrument all the reads
- The wait event elapsed time is not the I/O latency
The case of direct reads and asynchronous I/O

OS:

```plaintext
-- io_submit: timestamp=769804010693, program=oracle_18346_or, pid=18346, nr(num I/O)=1
  1: file descriptor=258, offset=93460627456, bytes=1048576, opcode=0
OS: <%= io_submit: timestamp=769804010897, program=oracle_18346_or, pid=18346, return(num I/O)=1
```

...many more io_submit and also io_getevents..

```
LIBAIO:-- io_getevents_0_4: timestamp=769804024035, program=oracle_18346_or, pid=18346, min_nr=1, timeout.tv_sec=600
OS: <%= io_getevents: timestamp=769804024060, program=oracle_18346_or, pid=18346, min_nr=1, timeout={.tv_sec=600, .tv_nsec=0}
LIBAIO:-- io_getevents_0_4: timestamp=769804028567, program=oracle_18346_or, pid=18346, min_nr=1, timeout.tv_sec=600
OS: <%= io_getevents: timestamp=769804034834, program=oracle_18346_or, pid=18346, min_nr=1, timeout={.tv_sec=600, .tv_nsec=0}
OS: <%= io_getevents: timestamp=769804037359, program=oracle_18346_or, pid=18346, return(num I/O)=4
  0: fildes=265, offset=93436510208, bytes=1048576
  1: fildes=267, offset=89061851136, bytes=1048576
  2: fildes=269, offset=78286684160, bytes=1048576
  3: fildes=268, offset=83802259456, bytes=983040
```

DB WAIT EVENT BEGIN: timestamp_ora=769804024008, pid=18346, sid=250, event#=direct path read

LIBAIO:-- io_getevents_0_4: timestamp=769804028511, program=oracle_18346_or, pid=18346, return(num I/O)=4
  0: fildes=260, offset=79065776128, bytes=1048576
  1: fildes=261, offset=89295683584, bytes=1048576
  2: fildes=263, offset=84572897280, bytes=1048576
  3: fildes=262, offset=94479843328, bytes=1048576

DB WAIT EVENT END: timestamp_ora=769804037433, pid=18346, sid=250, name=SYSTEM, event#=direct path read, p1=7, p2=4324864, p3=128, wait_time =13425, obj=32176, sql_hash=1782650121
Oracle wait events for asynchronous I/O cannot be used to study latency

Example of how to measure I/O latency from the block I/O interface using SystemTap:

```c
global latencyTimes, requestTime[10000]

probe ioblock_trace.request {
    requestTime[$bio] = gettimeofday_us()
}

probe ioblock.end {
    t = gettimeofday_us()
    s = requestTime[$bio]
    if (s > 0) {
        latencyTimes <<< (t-s)
        delete requestTime[$bio]
    }
}
```
Another way to measure I/O from the OS: using Ftrace

- https://github.com/brendangregg/perf-tools

```plaintext
# ./iolatency 10
Tracing block I/O. Output every 10 seconds. Ctrl-C to to end.

>=(ms) .. <(ms)  :  I/O  |Distribution
0   ->  1  :  95  |##      
1   ->  2  :  74  |##      
2   ->  4  :  475 |########
4   ->  8  :  2035 |########################################|
8   -> 16 :  1245 |####################################|
16  -> 32 :  37  |#
32  -> 64 :  11  |#
64  -> 128 :  7  |#
128 -> 256 :  23  |#
256 -> 512 :  10  |#
512 -> 1024 :  4  |#      
```
Example: Probe all blocks subject to physical I/O for performance investigations

- Goal: analyse physical reads: how many are ‘new’ and how many are repeated reads
- Aid for sizing DB cache and SSD storage cache
- SystemTap probe on kcbzvb (block read)
- Can drill down per file/object number/process

Example:

```
# stap -g -v oracle_read_profile.stp
number of distinct blocks read: 24513631
total number of blocks read: 86711189
```
Build Your Own Lab and Experiment

• Install a test environment (under VirtualBox)
  • RHEL/OEL 6.5 or higher
  • RHEL/OEL 7.0 with 3.10 kernel as preference

• Install additional packages
  • kernel-devel
  • debuginfo and debuginfo-common packages (available from https://oss.oracle.com)

• Install the advanced tracing tools
  • SystemTap version 2.5 or higher
Additional Tips for Userspace Investigations of Oracle

- Information on Oracle internal functions from MOS
  - Get a copy of “Note 175982.1”

- gdb (GNU debugger)
  - Read memory, stack backtraces and registers with gdb
  - Know the Linux call convention: args are in %rdi, %rsi,…

- Stack profile visualisations with flamegraphs
  - Help understand which functions are called more often

- DTrace-based tracing:
  - ‘Digger’ by Alexander Anokhin (best on Solaris DTrace)
Wish List: Statically Defined Probes in Oracle Code

- Statically defined probes
  - Make userspace tracing more clean and stable across versions
  - An elegant and direct way of collecting and aggregating info from the Oracle engine and correlate with OS data

- Examples of database engines that have static probes:
  - MySQL and PostgreSQL
Wish List: More Info on Oracle Functions, Variables, SGA Structures

- Oracle provides symbols in the executable
  - However no info on the kernel functions
  - Ideally we would like to have Oracle debuginfo
  - Documentation on what the functions do, which parameters they have, etc

- We can profit from knowledge sharing in the community
  - There is much more to investigate!
Acknowledgements and Contacts

• CERN Colleagues and in particular the Database Services Group
  • Our shared blog: http://db-blog.web.cern.ch/

• Additional credits
  • Frits Hoogland for original work and collaboration on the research
  • Many thanks to for sharing their work and original ideas: Tanel Poder, Brendan Gregg, Alexander Anokhin, Kevin Closson
Example SystemTap Scripts for Oracle Userspace Investigations

- Download from: http://cern.ch/canali/resources.htm

- histograms_oracle_events_11204.stp
- histograms_oracle_events_12102.stp
- histograms_oracle_events_version_independent.stp
- trace_oracle_events_11204.stp
- trace_oracle_events_12102.stp
- trace_oracle_logicalio_wait_events_physicalio_11204.stp
- trace_oracle_logicalio_wait_events_physicalio_12102.stp
- trace_oracle_logical_io_basic.stp
- trace_oracle_logical_io_count.stp
- trace_oracle_wait_events_asyncio_libaio_11204.stp
- trace_oracle_wait_events_asyncio_libaio_12102.stp
- measure_io_patterns
  - blockio_latency.stp
  - Oracle_read_profile.stp
  - Oracle_read_profile_drilldown_file.stp
  - Oracle_read_profile_drilldown_objectnum.stp
- experimental
  - logical_io_latency.stp
  ..
Conclusions

- **Linux tools for advanced troubleshooting**
- OS dynamic tracing, userspace tracing, ..
- Extend and complement Oracle wait interface data
- Collect data not available with other methods

- **Perf and Systemtap**
  - Already available on RHEL6 or higher
  - Powerful and fun to work with
  - Easy to start: build on example scripts

- Happy testing!

Luca.Canali@cern.ch, Zbigniew.Baranowski@cern.ch