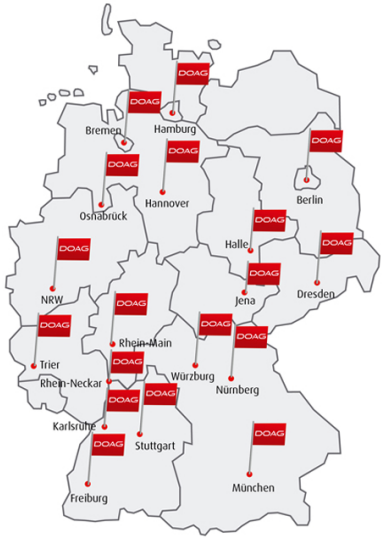
Troubleshooting Oracle performance issues beyond the Oracle wait interface



Stefan Koehler

About me

Stefan Koehler

- Independent Oracle performance consultant and researcher
- 13+ years using Oracle RDBMS - Independent since 2011
- Oracle performance and internals geek
- Main interests: Cost based optimizer and Oracle RDBMS internals

Services: “All about performance & troubleshooting”

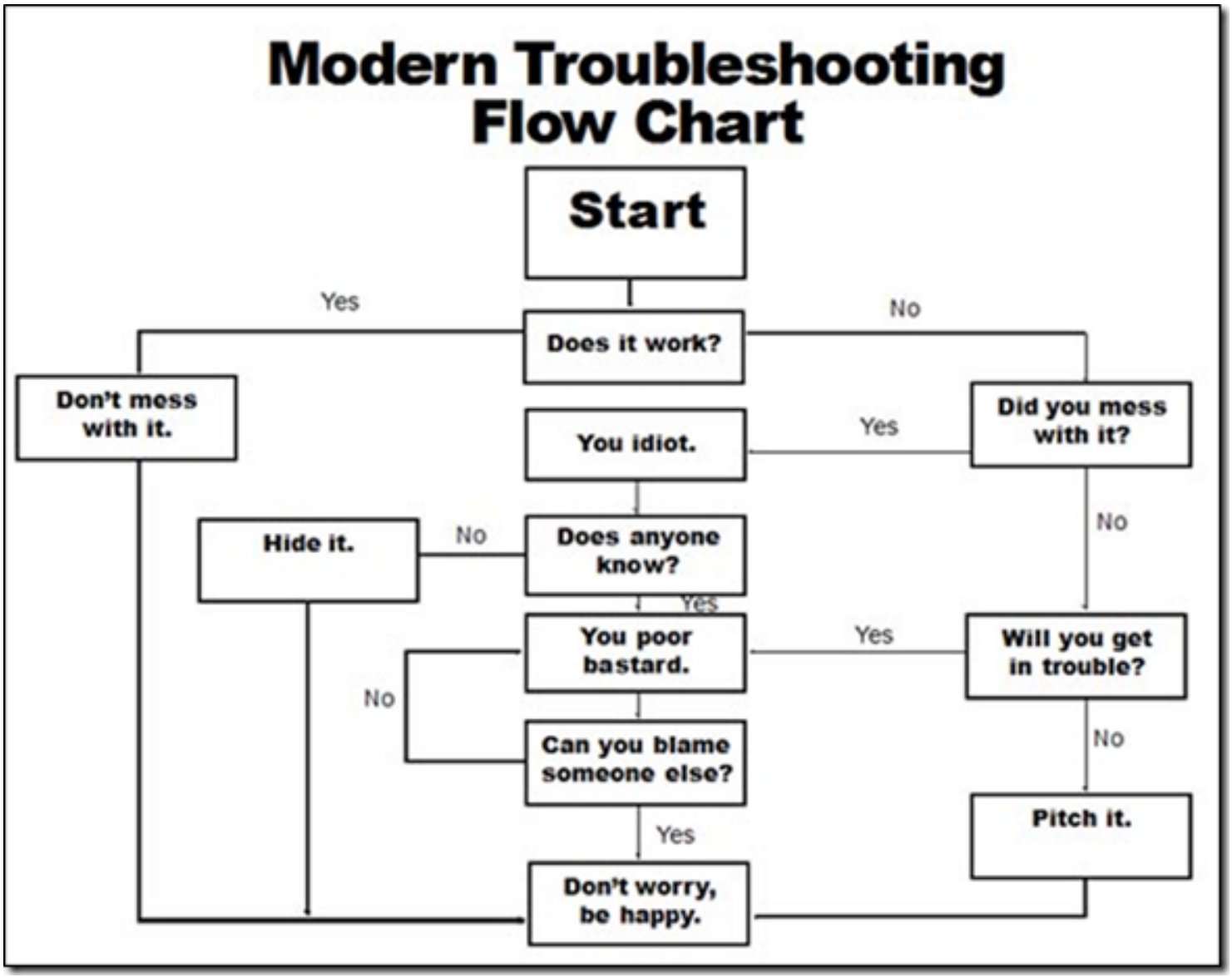
- Oracle performance tuning (e.g. Application, CBO, Database, Design, SQL)
- Oracle core internals researching (e.g. DTrace, GDB, Perf, etc.)
- Troubleshooting nontrivial Oracle RDBMS issues (e.g. Heap dumps, System state dumps, etc.)
- Services are mainly based on short-term contracting

www.soocs.de contact@soocs.de @OracleSK

Agenda

- Systematic troubleshooting - What are we talking about?
- Basics of Oracle's extended SQL trace & event instrumentation
- "System call trace" vs. "Stack trace"
- Capturing and interpreting "Stack traces" with focus on Linux
  - Oradebug (Oracle), GDB and its wrapper (OS), Perf (OS), Others
- Safety warning - Are "Stack traces" safe to use in production?
- Combine Oracle wait interface and "Stack traces"
- Use cases
  - Long parse time
  - Real life root cause identified + fixed with help of "Stack traces"

Systematic troubleshooting - What are we talking about? (1)



Systematic troubleshooting - What are we talking about? (2)

1. Identify performance bottleneck based on response time
   Method R by Cary Millsap

Business process is affected by single SQL running on CPU only

2. Interpret execution plan with help of additional SQL execution statistics (or Real-Time SQL Monitoring) and/or wait interface
   - PL/SQL package DBMS_XPLAN or DBMS_SQLTUNE

No execution plan issue found

3. Capture and interpret session statistics and performance counters
   - Tools like Snapper by Tanel Poder

Still no obvious root cause for the high CPU load

4. Capture and interpret system call or stack traces
   - **This is what this session is about. Disassembling Oracle code.**

Basics of Oracle's extended SQL trace and event instrumentation (1)

- Extended SQL trace basics
  - **Database call (= OPI call) lines**
    begin with the keyword PARSE, EXEC, FETCH, CLOSE, RPC EXEC, UNMAP, SORT UNMAP, LOBREAD, or LOBARRTMPFRE. Such a line indicates that an application client has made a database call, and that the database has responded to it.

```
FETCH #12390720:c=8000,e=1734,p=1,cr=7,cu=0,mis=0,r=1,dep=2,og=4,plh=3992920156,tim=1294438818783887
```

More CPU time than elapsed time?
- CPU time value is precise to ±10.000 microseconds
- Elapsed time value is precise to ±1 microseconds

  - **System call lines**
    begin with the keyword WAIT. Such a line indicates that the Oracle kernel process has made one (or more) syscall, and that the OS has responded to it.

```
WAIT #12397272: nam='db file sequential read' ela= 221 file#=1 block#=2735 blocks=1 obj#=423 tim=1294438818791494
```

Basics of Oracle's extended SQL trace and event instrumentation (2)

- Oracle's event instrumentation basics

  - **Database call lines (simplified)**

```
12 function dbcall {
13   e0 = gettime();                                          # mark the time before the call
14   c0 = getrusage();                                        # mark the CPU time consumed before the call
15
16   ...                                                      # execute the dbcall here; maybe make calls to dbcall() or syscall()
17
18   e1 = gettime();                                          # mark the time after the call
19   c1 = getrusage();                                        # mark the CPU time consumed after the call
20   nam = ...;                                               # name assigned to the call: PARSE, EXEC, FETCH, ...
21   cid = ...;                                               # cursor handle ID
22   e = e1 - e0;                                             # elapsed duration between wall times
23   c = (c1.utime + c1.stime) - (c0.utime + c0.stime)        # CPU time consumed by the call
24   write(TRC, "%s #%d:c=%d,e=%d...", nam, cid, c, e, ...); # write the dbcall record to the trace file
25 }
```

Screenshots from ISBN 1518897886

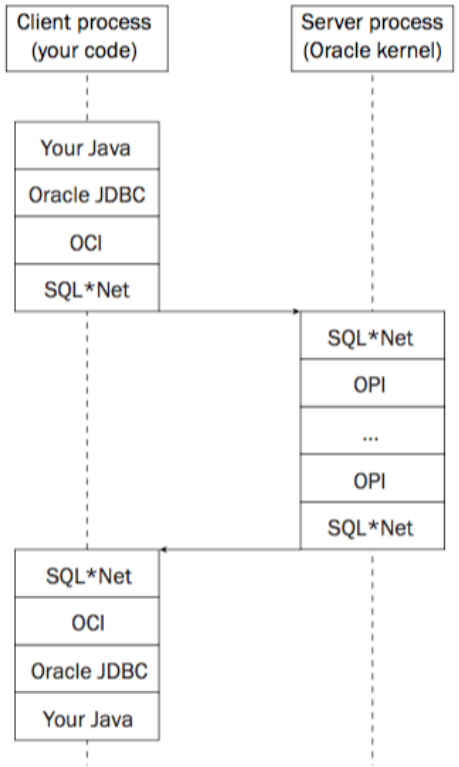
  - **System call lines (simplified)**

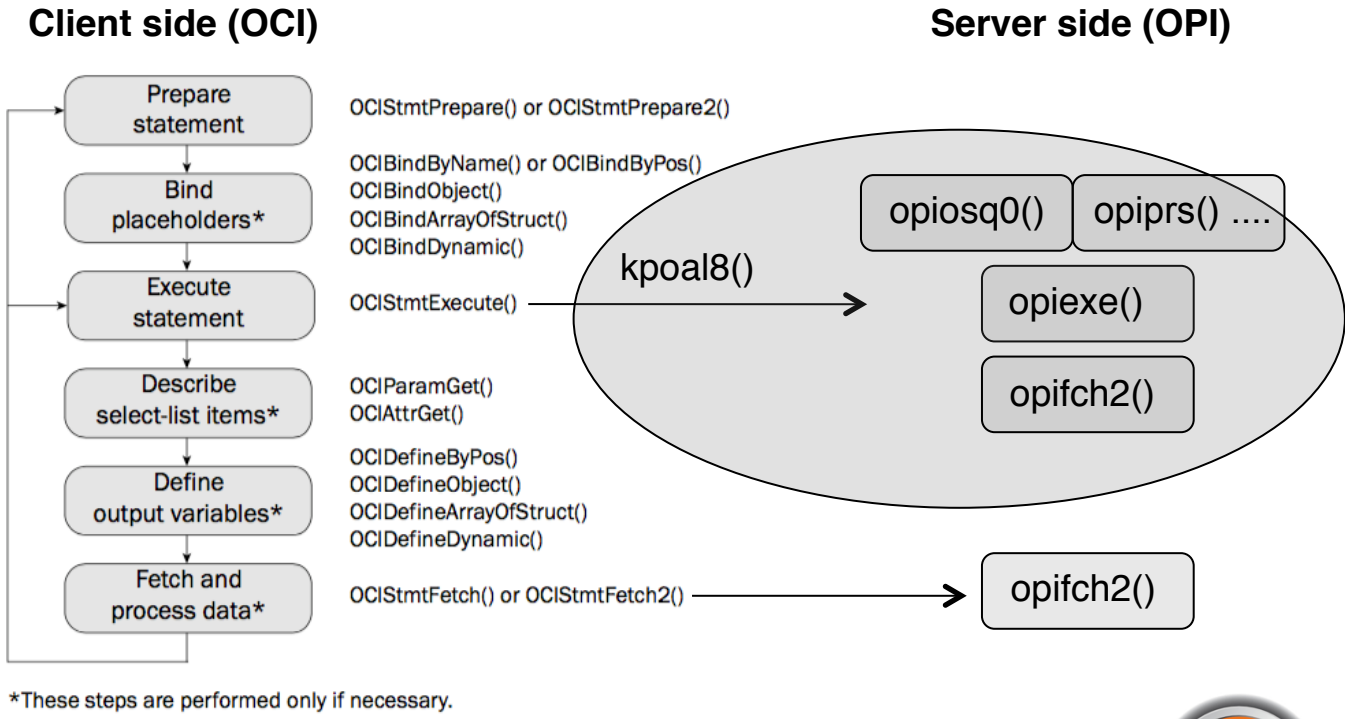
```
1  function syscall {
2    ela0 = gettime();                                        # mark the time before the call
3
4    ...                                                      # execute the syscall here
5
6    ela1 = gettime();                                        # mark the time after the call
7    nam = ...;                                               # name assigned to the code path by some Oracle developer
8    cid = ...;                                               # cursor handle ID
9    ela = ela1 - ela0;                                       # elapsed duration between wall times
10   write(TRC, "WAIT #%d: nam='%s'...", cid, nam, ela, ...);# write the syscall record to the trace file
11 }
```

Instrumentation overhead of round about 6 microseconds for wait events in average

Basics of Oracle's extended SQL trace and event instrumentation (3)

- Are all OPI calls instrumented in Oracle's extended SQL trace?





Screenshots from ISBN 1518897886


```
SQL> select count(*) from v$toplevelcall;
SQL> alter session set events '10051 trace name context forever,
     level 1';
```

“System call trace” vs. “Stack trace”

- System call trace
  - A system call is the fundamental interface between an application and the (Linux) kernel and is generally not invoked directly, but rather via wrapper functions in glibc (or some other library). For example: *truncate() → truncate() or truncate64()*
  - Example of Oracle using system calls: *gettimeofday(), clock_gettime(), pread()*
  - Tools: *Strace (Linux), Truss (AIX / Solaris), Tusc (HP-UX)*
  - Be aware of vDSO / vSyscall64 feature when tracing system calls on Linux (control option “kernel.vsyscall64” has been removed with kernel version 3.0)

- Stack trace / Stack backtrace
  - A call stack is the list of names of methods called at run time from the beginning of a program until the execution of the current statement
  - Tools: *Oradebug (Oracle), GDB + wrappers or Perf or SystemTap (Linux), DTrace (Solaris),* Procstack *(AIX)*

The stack trace includes the called methods / functions of an Oracle process and the system call trace includes only the (function) requests to the OS kernel

Capturing “Stack traces” with focus on Linux (1)

- Tool “Oradebug” (Oracle tool and platform independent)


```
SQL> oradebug SETMYPID / SETOSPID <PID>
SQL> oradebug SHORT_STACK
```

Code path of oradebug request - SIGUSR2 signal +<NUM> = Offset in bytes from beginning of symbol (function) where child function call happened

```
ksedsts()+213<-ksdxfstk()+34<-ksdxcb()+914<-sspuser()+191<-__sighandler()<-read()+14
<-nttfprd()+309<-nsbasic_brc()+393<-nsbrecv()+86<-nioqrc()+520<-opikndf2()+995<-opit
sk()+803<-opiino()+888<-opiodr()+1170<-opidrv()+586<-sou2o()+120<-opimai_real()+151<
-ssthrdmain()+392<-main()+222<-__libc_start_main()+253
```

Oracle Call Interface	OCI
User Program Interface	UPI
Net8	
Oracle Program Interface	OPI
Compilation Layer	KK
Execution Layer	KX
Distributed Execution Layer	K2
Network Program Interface	NPI
Security Layer	KZ
Query Layer	KQ
Recursive Program Interface	RPI
Access Layer	KA
Data Layer	KD
Transaction Layer	KT
Cache Layer	KC
Services Layer	KS
Lock Management Layer	KJ
Generic Layer	KG
Operating System Dependencies	S

Oracle kernel layers from ISBN 1-56592-598-X

Translation of Oracle kernel function names

1. MOS ID #175982.1 (Google it)
2. ORADEBUG DOC COMPONENT
3. My Oracle support bug section
4. Google

Capturing “Stack traces” with focus on Linux (2)

- Tool “GDB” (GNU debugger) and its wrapper script pstack

```
shell> gdb
(gdb) attach <PID>
(gdb) backtrace
```

```
#0  0x000000336d00e530 in __read_nocancel () from /lib64/libpthread.so.0
#1  0x000000000b924620 in snttread ()
#2  0x000000000b923a95 in nttfprd ()
#3  0x000000000b90eac9 in nsbasic_brc ()
#4  0x000000000b90e8d6 in nsbrecv ()
#5  0x000000000b913778 in nioqrc ()
#6  0x000000000b5b9a43 in opikndf2 ()
#7  0x0000000001e44f93 in opitsk ()
#8  0x0000000001e49c28 in opiino ()
#9  0x000000000b5bc3c2 in opiodr ()
#10 0x0000000001e4118a in opidrv ()
#11 0x00000000025381f8 in sou2o ()
#12 0x0000000000b393a7 in opimai_real ()
#13 0x0000000002542898 in ssthrdmain ()
#14 0x0000000000b392de in main ()
```

```
shell> /usr/bin/pstack <PID>
```

```
#0  0x000000336d00e530 in __read_nocancel () from /lib64/libpthread.so.0
#1  0x000000000b924620 in snttread ()
#2  0x000000000b923a95 in nttfprd ()
#3  0x000000000b90eac9 in nsbasic_brc ()
#4  0x000000000b90e8d6 in nsbrecv ()
#5  0x000000000b913778 in nioqrc ()
#6  0x000000000b5b9a43 in opikndf2 ()
#7  0x0000000001e44f93 in opitsk ()
#8  0x0000000001e49c28 in opiino ()
#9  0x000000000b5bc3c2 in opiodr ()
#10 0x0000000001e4118a in opidrv ()
#11 0x00000000025381f8 in sou2o ()
#12 0x0000000000b393a7 in opimai_real ()
#13 0x0000000002542898 in ssthrdmain ()
#14 0x0000000000b392de in main ()
```

GDB is based on ptrace() system calls

Capturing “Stack traces” with focus on Linux (3)

- Performance counters for Linux (Linux kernel-based subsystem)
  - Framework for collecting and analyzing performance data, e.g. hardware events, including retired instructions and processor clock cycles and many more
  - Based on sampling (default avg. 1000 Hz respectively 1000 samples/sec)
  - Caution in virtualized environments when capturing cpu-cycles events (VMware KB #2030221)
  - Tool ”Perf” is based on perf_events interface exported by Linux kernel (>= 2.6.31)


```
shell> perf record -e cpu-cycles -o /tmp/perf.out -g -p <PID>
```

Hardware event (cpu-cycles) = Usage of kernel’s performance registers
Software event (cpu-clock) = Depends on timer interrupt

- Poor man’s stack profiling
  - When no other tool is available and you need a quick insight into sampled stacks

```
shell> export LC_ALL=C ; for i in {1..20} ; do pstack <PID>
       | ./os_explain -a ; done | sort -r | uniq -c
```

Script by Tanel Poder to translate C function names into known functionality

Capturing “Stack traces” with focus on Linux (4)

- Listing other capturing tools for completeness
  - OStackProf by Tanel Poder (needs to be run from Windows SQL*Plus client as based on oradebug short_stack and VBS script for post processing)
  - DTrace on Solaris (e.g. DTrace toolkit script “hotuser” by Brendan Gregg or analysis with PID provider)
  - DTrace on Linux lacks in case of userspace integration / probing
  - SystemTap (with Linux kernel >= 3.5 for userspace probing) otherwise “utrace patch” needs to be applied

Interpreting “Stack traces” with focus on Linux

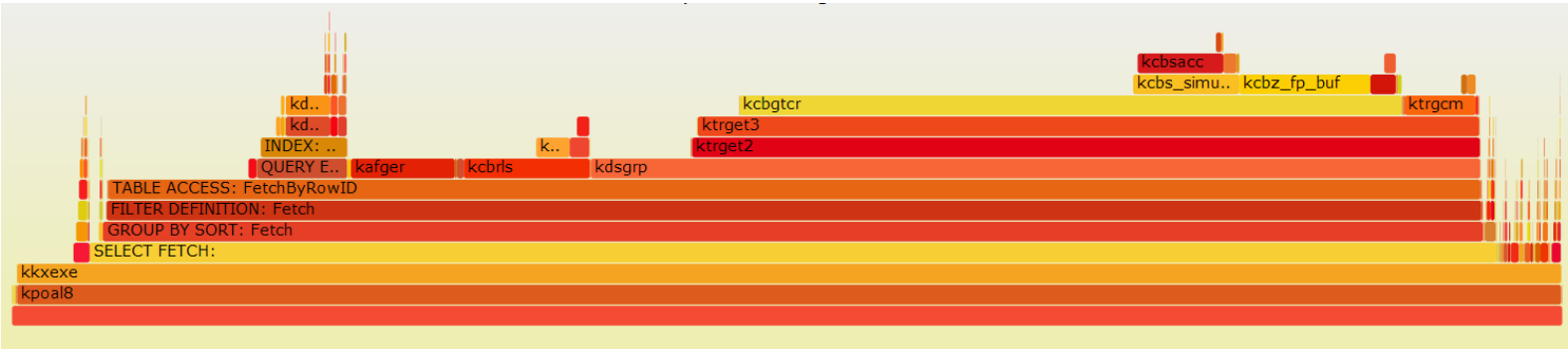
- Performance counters for Linux (Linux kernel-based subsystem)
  - Tool “Perf”

```
shell> perf report -i /tmp/perf.out -g none -n --stdio
shell> perf report -i /tmp/perf.out -g graph -n --stdio
```

Problem: Depending on the stack trace content there may be too much data to interpret in this format. **Main question: Where is the bulk of CPU time spent?**

  - Tool “Flame Graph” by Brendan Gregg (works with DTrace & SystemTap too)

```
shell> perf script -i /tmp/perf.out | ./stackcollapse-
       perf.pl > out.perf.folded
shell>./flamegraph.pl out.perf.folded > perf-out.svg
```



Safety warning - Are “Stack traces” safe to use in production?

- If your database is already in such a state …
  … then don’t worry about the possible consequences and issues by capturing stack traces

- Be aware of different behavior by capturing stack traces, if only some specific business processes are affected
  - Tool “Oradebug” - “Unsafe” as it alters code path / SIGUSR2 (e.g bug #15677306) 👎
  - Tool “GDB” (and its wrappers) - “Unsafe” as it suspends the process (ptrace syscall) with possible impact on communication to kernel or other processes 👎
  - Tool “Perf” based on Linux performance counters - Safe by design, but fallback to the other tools is still needed, if the process is not running on CPU and stuck somewhere else 👍
  - DTrace (Solaris) - Safe by design 👍

Combine Oracle wait interface and “Stack traces” (1)

- Fulltime.sh by Craig Shallahamer and Frits Hoogland
  - Based on V$SESSION_EVENT and Linux performance counters

```
shell> fulltime.sh <PID> <SAMPLE_DURATION> <SAMPLE_COUNT>
```

```
PID: 1861  SID: 12  SERIAL: 5  USERNAME: TEST  at 22-Sep-2015 11:06:26
CURRENT SQL: select /*+ use_hash(t2) */ count(*) from TEST t1, TEST t2 where t1.own

total time: 11.376 secs,  CPU: 6.584 secs (57.88%),  wait: 4.792 secs (42.12%)

                                                                  Time
Time Component                                                    secs       %
----------------------------------------------------------------- ---------- -------
wait: SQL*Net message from client                                    4.767   41.90
cpu : [.] qerhjWalkHashBucket                                        1.976   17.37
cpu : [.] rworupo                                                    1.064    9.35
cpu : [.] kxhrPUcompare                                              0.839    7.37
cpu : [.] qesrAddRowFromCtx                                          0.749    6.58
cpu : [.] qksbgGetCursorVal                                          0.508    4.47
cpu : [k] finish_task_switch                                         0.491    4.32
cpu : [.] kxhrUnpack                                                 0.317    2.78
cpu : [.] _intel_fast_memcmp                                         0.231    2.03
cpu : [.] qksbgGetVal                                                0.227    2.00
cpu : [?] sum of funcs consuming less than 2% of CPU time            0.182    1.60
wait: direct path read                                               0.025     .22
wait: db file sequential read                                        0.000     .00
wait: SQL*Net message to client                                      0.000     .00
```

Combine Oracle wait interface and “Stack traces” (2)

- Oracle 12c enhancement - Diagnostic event “wait_event[]” in "new" kernel diagnostics & tracing infrastructure

```
SQL> oradebug doc event name wait_event
wait_event: event to control wait event post-wakeup actions

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

WAIT #140311192202592:
nam='direct path read' ela= 2166 file number=4 first dba=179 blockcnt=13 obj#=92417 tim=589834270
kslwtectx<-ksfdaio<-kcflbi<-kcbldio<-kcblrd<-kcblgt<-kcbldrget<-kcbgtcr<-ktrget3<-ktrget2<-
kdst_fetch<-kdstf000010100001km<-kdsttgr<-qertbFetch<-qerstFetch<-rwsfcd<- qerstFetch<-
qerhjFetch<-qerstFetch<-qergsFetch<-qerstFetch<-opifch2<-kpoal8<-opiodr<- ttcpips
```

Combine extended SQL trace & event wait_event[]
Function kslwtectx marks end of wait event

```
SQL> alter session set events 'wait_event["<wait event name>"]
{wait:minwait=1000} trace(''event "%", p1 %, p2 %, p3 %, wait
time %\nStk=%\n'', evargs(5), evargn(2), evargn(3),evargn(4),
evargn(1), shortstack())';
```

Additional condition (e.g. min wait duration in microseconds) and different syntax is also available

Use case: Long parse time (1)

- Common expression of “parsing a SQL” is a two step approach
  - **Parsing (simplified)**
    Oracle checks if the SQL statement is a valid one (syntactic analysis). Afterwards it checks and figures out things like object types, columns in object types, constraints, triggers, indexes, privileges, etc. (semantic analysis).
  - **Optimizing (simplified)**
    After the statement is parsed, the optimizer starts its work by checking stats and doing its arithmetic (logical and physical).
- Which db call does parse / optimize a SQL and consumes CPU?

Same SELECT
1 slightly diff

- PARSE #?

145.88 sec CPU time on PARSE #

```
PARSE #139908600390448:c=145882000,e=147715884,p=24,cr=110,cu=0,mis=1,r=0,dep=0,og=1,plh=2871710578,tim=6634689356
EXEC  #139908600390448:c=5000,e=5312,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,plh=2871710578,tim=6634697928
```

- EXEC #?

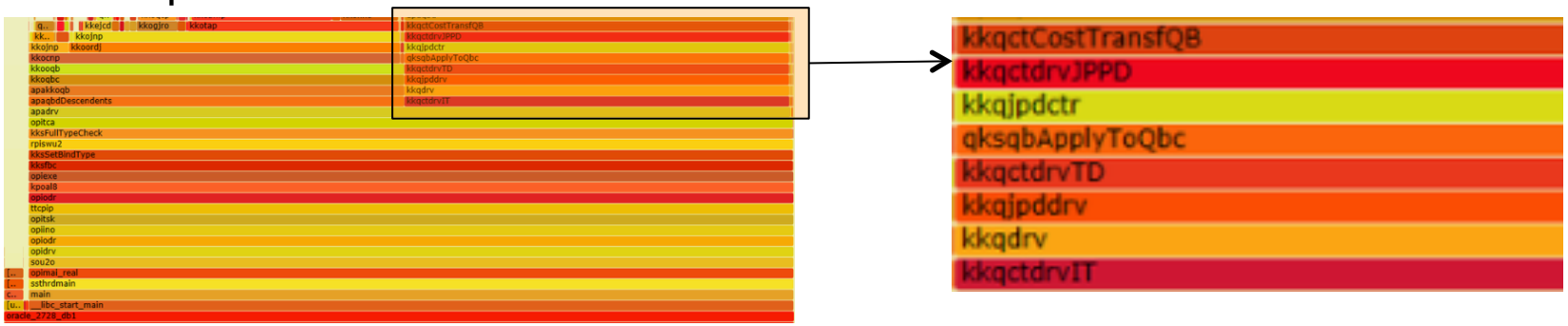
173.18 sec CPU time on EXEC #

```
PARSE #140105239001656:c=158000,e=158917,p=11,cr=47,cu=0,mis=1,r=0,dep=0,og=1,plh=0,tim=6689158919
EXEC  #140105239001656:c=173185000,e=173276889,p=11,cr=63,cu=0,mis=0,r=0,dep=0,og=1,plh=2871710578,tim=6862439553
```

Deferred optimization

Use case: Long parse time (2)

- Oracle version < 12.1.0.2.160419 (PSU Apr 2016)
  - Need to profile the CBO C call stack



- Oracle version >= 12.1.0.2.160419 (PSU Apr 2016)
  - CBO (10053) trace enhancement #16923858 - MOS ID #16923858.8
  - May still need to profile the CBO C call stack if parsing is stuck forever

```
     BUGNO     VALUE  DESCRIPTION
---------- ---------- ----------------------------------------------------------------
  16923858          6  Display timings in Optimizer trace when over 10^<value> microsec

...
TIMER: Star Transformation SEL$1 cpu: 0.000000 sec elapsed: 0.000349 sec
TIMER: Join Factorization SEL$1 cpu: 0.001000 sec elapsed: 0.000596 sec
TIMER: Cost-Based Join Predicate Push-Down SEL$1 cpu: 0.000000 sec elapsed: 0.000749 sec
TIMER: OLAP AW Join Pushing SEL$1 cpu: 0.000000 sec elapsed: 0.000012 sec
TIMER: Type Checking after CBQT SEL$1 cpu: 0.003000 sec elapsed: 0.002969 sec
TIMER: Cost-Based Transformations (Overall) SEL$1 cpu: 0.030000 sec elapsed: 0.029888 sec
TIMER:       bitmap access paths cpu: 0.000000 sec elapsed: 0.000106 sec
TIMER:      costing general plans cpu: 0.000000 sec elapsed: 0.000134 sec
TIMER:       bitmap access paths cpu: 0.000000 sec elapsed: 0.000023 sec
...
TIMER: Access Path Analysis (Final) SEL$1 cpu: 0.139000 sec elapsed: 0.143430 sec
TIMER: SQL Optimization (Overall) SEL$1 cpu: 0.254000 sec elapsed: 0.257128 sec
...
```

Use case: Real life root cause identified + fixed with help of “Stack traces” (1)

- Environment and issue
  - Large SAP system with Oracle 11.2.0.2 running on AIX 6.1
  - Most of the SAP work processes are stuck in a simple INSERT statement and burning up all CPUs on database server
  - Index key compression and OLTP compression is enabled
  - SQL statement:

```
SQL> INSERT INTO "BSIS”  VALUES(:A0 , ... ,:A81);
```

- Applying systematic troubleshooting
  - Identify performance bottleneck based on response time with Method R

    Performance bottleneck is clearly caused by the INSERT statement as 100% of the end user response time is spent on it and all application processes are affected by this

    No further response time analysis needed here

Use case: Real life root cause identified + fixed with help of “Stack traces” (2)

- Applying systematic troubleshooting
  - Interpret execution plan with help of additional SQL execution statistics (or Real-Time SQL Monitoring) and wait interface

```
-----------------------------------------------
| Id  | Operation                | Name | Cost  |
-----------------------------------------------
|   0 | INSERT STATEMENT         |      |     1 |
|   1 |  LOAD TABLE CONVENTIONAL |      |       |
-----------------------------------------------
```

```
SQL> select LAST_DDL_TIME, CREATED, STATUS, OBJECT_NAME from dba_objects where OBJECT_NAME like 'BSIS%';
LAST_DDL_TIME       CREATED             STATUS  OBJECT_NAME
------------------- ------------------- ------- -------------
30.07.2011 23:28:44 30.07.2011 16:41:53 VALID   BSIS
30.07.2011 23:28:47 30.07.2011 19:46:40 VALID   BSIS~0
30.07.2011 23:28:47 30.07.2011 21:18:13 VALID   BSIS~1

SQL> select INDEX_NAME, VISIBILITY, STATUS from dba_indexes where table_name = 'BSIS';
INDEX_NAME                     VISIBILIT STATUS
------------------------------ --------- --------
BSIS~1                         VISIBLE   VALID
BSIS~0                         VISIBLE   VALID

SQL> select data_object_id from dba_objects where object_name = 'BSIS';
DATA_OBJECT_ID
---------------
437114
```

Use case: Real life root cause identified + fixed with help of “Stack traces” (3)

- Applying systematic troubleshooting
  - Capture and interpret session statistics and performance counters

```
-- Session Snapper v3.52 by Tanel Poder @ E2SN ( http://tech.e2sn.com )
-----------------------------------------------------------------------------------------------------------------------
    SID, USERNAME  , TYPE, STATISTIC                                    ,     HDELTA, HDELTA/SEC,    %TIME, GRAPH
-----------------------------------------------------------------------------------------------------------------------
   1486, SAPSR3    , STAT, session logical reads                        ,    190.06k,     31.68k,
   1486, SAPSR3    , STAT, concurrency wait time                        ,         30,          5,
   1486, SAPSR3    , STAT, non-idle wait time                           ,         30,          5,
   1486, SAPSR3    , STAT, non-idle wait count                          ,         40,       6.67,
   1486, SAPSR3    , STAT, db block gets                                ,    189.65k,     31.61k,
   1486, SAPSR3    , STAT, db block gets from cache                     ,    189.64k,     31.61k,
   1486, SAPSR3    , STAT, db block gets from cache (fastpath)          ,    187.67k,     31.28k,
   1486, SAPSR3    , STAT, consistent gets                              ,          4,       .67,
   1486, SAPSR3    , STAT, consistent gets from cache                   ,          4,       .67,
   1486, SAPSR3    , STAT, consistent gets from cache (fastpath)        ,          4,       .67,
   1486, SAPSR3    , STAT, calls to kcmgcs                              ,      3.87k,       645,
   1486, SAPSR3    , WAIT, latch: cache buffers chains                  ,    378.47ms,    63.08ms,     6.3%, |@         |

-----------------------------------------------------------------------
Active% | SQL_ID          | EVENT                     | WAIT_CLASS
-----------------------------------------------------------------------
    88% | 74wzh02xwat0c   | ON CPU                    | ON CPU
    12% | 74wzh02xwat0c   | latch: cache buffers chai | Concurrency
```

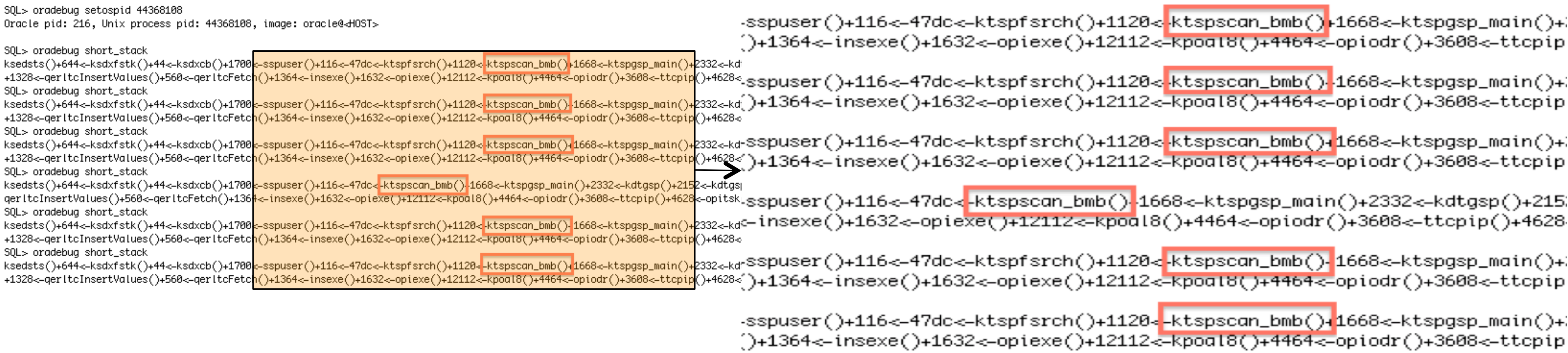
Use case: Real life root cause identified + fixed with help of “Stack traces” (4)

- Applying systematic troubleshooting
  - Capture and interpret session statistics and performance counters

```
PARSING IN CURSOR #4576739504 len=508 dep=0 uid=27 oct=2 lid=27 tim=13805989431919 hv=3150275596 ad='700000fb6b04460' sqlid='74wzh02xwat0c'
INSERT INTO "BSIS"
VALUES(:A0 ,:A1 ,:A2 ,:A3 ,:A4 ,:A5 ,:A6 ,:A7 ,:A8 ,:A9 ,:A10 ,:A11 ,:A12 ,:A13 ,:A14 ,:A15 ,:A16 ,:A17 ,:A18 ,:A19 ,:A20 ,:A21 ,:A22 ,:A23 ,:A
46 ,:A47 ,:A48 ,:A49 ,:A50 ,:A51 ,:A52 ,:A53 ,:A54 ,:A55 ,:A56 ,:A57 ,:A58 ,:A59 ,:A60 ,:A61 ,:A62 ,:A63 ,:A64 ,:A65 ,:A66 ,:A67 ,:A68 ,:A69 ,:
END OF STMT
PARSE #4576739504:c=0,e=90,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,plh=0,tim=13805989431918
WAIT #4576739504: nam='db file sequential read' ela= 228 file#=98 block#=179725 blocks=1 obj#=437114 tim=13805989433364
WAIT #4576739504: nam='db file sequential read' ela= 1054 file#=98 block#=184097 blocks=1 obj#=437114 tim=13805989436881
WAIT #4576739504: nam='db file sequential read' ela= 317 file#=98 block#=184545 blocks=1 obj#=437114 tim=13805989441522
WAIT #4576739504: nam='db file sequential read' ela= 451 file#=98 block#=184673 blocks=1 obj#=437114 tim=13805989442104
WAIT #4576739504: nam='db file sequential read' ela= 1255 file#=98 block#=183906 blocks=1 obj#=437114 tim=13805989444506
WAIT #4576739504: nam='db file sequential read' ela= 804 file#=98 block#=184354 blocks=1 obj#=437114 tim=13805989445951
WAIT #4576739504: nam='db file sequential read' ela= 4082 file#=98 block#=184482 blocks=1 obj#=437114 tim=13805989450630
WAIT #4576739504: nam='db file sequential read' ela= 330 file#=98 block#=183843 blocks=1 obj#=437114 tim=13805989451276
WAIT #4576739504: nam='db file sequential read' ela= 865 file#=98 block#=184163 blocks=1 obj#=437114 tim=13805989452311
WAIT #4576739504: nam='db file sequential read' ela= 417 file#=98 block#=184291 blocks=1 obj#=437114 tim=13805989453053
WAIT #4576739504: nam='db file sequential read' ela= 1730 file#=98 block#=184419 blocks=1 obj#=437114 tim=13805989455987
WAIT #4576739504: nam='db file sequential read' ela= 366 file#=98 block#=184100 blocks=1 obj#=437114 tim=13805989457359
WAIT #4576739504: nam='db file sequential read' ela= 3358 file#=98 block#=184228 blocks=1 obj#=437114 tim=13805989460902
WAIT #4576739504: nam='db file sequential read' ela= 448 file#=98 block#=184676 blocks=1 obj#=437114 tim=13805989461931
WAIT #4576739504: nam='db file sequential read' ela= 1900 file#=98 block#=184804 blocks=1 obj#=437114 tim=13805989464172
WAIT #4576739504: nam='db file sequential read' ela= 7831 file#=98 block#=183909 blocks=1 obj#=437114 tim=13805989472339
```

Use case: Real life root cause identified + fixed with help of “Stack traces” (5)

- Applying systematic troubleshooting
  - Capture and interpret system call or stack traces



- Process is stuck in main call stack “ktspscan_bmb” + on-top functions. The high CPU usage (“session logical reads”) is the consequence of it
- Table “BSIS” is stored in an ASSM tablespace and the call stack “ktspfsrch <- ktspscan_bmb” is related to “first level bitmap block search”
- MOS search results in bug #13641076 – “HIGH AMOUNT OF BUFFER GETS FOR INSERT STATEMENT - REJECTIONLIST DOES NOT FIRE”
- Root cause found and can be fixed by applying corresponding patch

Questions and answers

Download links and further information about all mentioned tools and procedures can be found on website www.soocs.de/public/talk/

www.soocs.de contact@soocs.de @OracleSK