Identifying performance issues beyond the Oracle wait interface

Stefan Koehler
Freelance Consultant (Soocs)
Coburg, Germany

Keywords
Oracle performance, Stack trace, System call trace, Perf, DTrace, GDB, Oradebug, Performance counters for Linux, Systemtap, (CPU) Flame graph, Systematic troubleshooting, Strace

Introduction
The Oracle code is already instrumented very well and provides a lot of wait events, performance statistics and performance metrics by default. Unfortunately all of these statistics and metrics do not provide enough information to drill down to the root cause of a performance issue in various cases (mostly CPU based issues). In such cases the Oracle code needs to be disassembled and profiled. This session explores different possibilities to profile the Oracle code and how to interpret the results.

About the author
Stefan Koehler follows his passion about Oracle since +12 years and specialized in Oracle performance tuning, especially in Oracle cost based optimizer (CBO) and database internal related topics. Nowadays he primarily works as a freelance Oracle database performance consultant in large mission critical environments (e.g. +10 TB databases) for market-leading companies or multi-billion dollar enterprises in various industries all over the world. He usually supports his clients in understanding and solving complex Oracle performance issues and troubleshooting nontrivial database issues on short-term contracting basis.

Systematic performance troubleshooting approach
Discovering the root cause of a slow application is often not easy and finding a proper solution to the root cause(s) may even be harder, if the Oracle wait interface and performance metrics do not provide any useful hints. A systematic and scientific approach like the following is usually needed to address a performance issue in the most efficient and economic way:

1. Identify the performance bottleneck based on response time with Method R by Cary Millsap [1]
   - 1.1 Select the user actions for which the business needs improved performance.
   - 1.2 Collect properly scoped diagnostic data that will allow you to identify the causes of response time.
   - 1.3 Execute the candidate optimization activity that will have the greatest net payoff to the business. If even the best net-payoff activity produces insufficient net payoff, then suspend your performance improvement activities until something changes.
   - 1.4 Go to step 1.1.

2. Interpret the execution plan with help of additional SQL execution statistics (or Real-Time SQL Monitoring) and the wait interface
   - 2.1 For example with the proper diagnostic data from point 1.2 and with PL/SQL package DBMS_XPLAN or DBMS_SQLTUNE

3. Capture and interpret the session statistics and performance counters
   - 3.1 For example with tools like Snapper by Tanel Poder [2]

4. Capture and interpret system call traces or stack traces dependent on the issue

The DOAG conference session and this paper is only about step 4 in the systematic performance troubleshooting approach. Step 4 is like disassembling the Oracle code and should only be considered if all the previous steps do not point to the root cause of the issue or for researching purpose of course. Why do we need this at all? Generally speaking we need to know in what kind of Oracle kernel code the process is stuck or in which Oracle kernel code the most (CPU) time is spent.

"System call" and "Call stack"

System call [3]

The system call is the fundamental interface between an application and the Linux kernel.

System calls and library wrapper functions

System calls are generally not invoked directly, but rather via wrapper functions in glibc (or perhaps some other library). For details of direct invocation of a system call, see intro(2). Often, but not always, the name of the wrapper function is the same as the name of the system call that it invokes. For example, glibc contains a function truncate() which invokes the underlying "truncate" system call. Often the glibc wrapper function is quite thin, doing little work other than copying arguments to the right registers before invoking the system call, and then setting errno appropriately after the system call has returned.

Note: System calls indicate a failure by returning a negative error number to the caller; when this happens, the wrapper function negates the returned error number (to make it positive), copies it to errno, and returns -1 to the caller of the wrapper.

Sometimes, however, the wrapper function does some extra work before invoking the system call. For example, nowadays there are (for reasons described below) two related system calls, truncate(2) and truncate64(2), and the glibc truncate() wrapper function checks which of those system calls are provided by the kernel and determines which should be employed.

Exemplary tools to trace system calls: strace (Linux), truss (AIX / Solaris), tusc (HP-UX)

Call stack

A call stack is the list of names of methods that are called at run time from the beginning of a program until the execution of the current statement.

A call stack is mainly intended to keep track of the point to which each active subroutine should return control when it finishes executing. A call stack acts as a tool to debug an application when the method to be traced can be called in more than one context. This forms a better alternative than adding tracing code to all methods that call the given method.

Exemplary tools to capture stack traces: oradebug (Oracle), gdb and its wrapper scripts or perf or systemtap (Linux), dtrace (Solaris), procstack (AIX)

Generally speaking the main difference is that the call stack trace includes the called methods or functions of an application (including the system calls) like an Oracle process and that the system call trace includes only the (function) requests to the operating system kernel.

System call trace anomaly on Linux
The main focus in this paper is on capturing and interpreting call stacks, but one anomaly by tracing system calls on Linux should be mentioned as it is causing confusion from time to time.

For example: A particular SQL is executed with hint „gather_plan_statistics" to enable rowsource statistics. In this case a lot of gettimeofday() (= up to and including Oracle 11g implementation) or clock_gettime() (= Oracle 12c implementation) system calls should usually appear as Oracle needs that timing information to calculate the rowsource statistics.

The following demo is run on Oracle 12.1.0.1 and Linux kernel 2.6.39-400.109.1.el6uek.x86_64.

```
NAME                                  VALUE
------------------------------------- -------
_rowsource_statistics_sampfreq          128

SYS@T12DB:181> select /*+ gather_plan_statistics */ count(*) from dba_objects;

shell> strace -cf -p 1915
Process 1915 attached - interrupt to quit
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 75.81    0.000514          24        21           mmap
 24.19    0.000164           9        19           getrusage
  0.00    0.000000           0         2           read
  0.00    0.000000           0         2           write
  0.00    0.000000           0         2           munmap
  0.00    0.000000           0         8           times
------ ----------- ----------- --------- --------- ----------------
100.00    0.000678                    54           total
```

Expecting to see a lot of gettimeofday() or clock_gettime() system calls during the SQL execution as the sampling rate is set to 128 (default value), but strace does not show any of these system calls at all.

The reason for this behavior is the vDSO / vsyscall64 implementation in Linux. [4]
Generally speaking a virtual dynamic shared object (vDSO), is a shared library that allows applications in user space to perform some kernel actions without as much overhead as a system call.

Enabling the vDSO instructs the kernel to use its definition of the symbols in the vDSO, rather than the ones found in any user-space shared libraries, particularly the glibc. The effects of enabling the vDSO are system wide - either all processes use it or none do. When enabled, the vDSO overrides the glibc definition of gettimeofday() or clock_gettime() with it's own. This removes the overhead of a system call, as the call is made direct to the kernel memory, rather than going through the glibc.

```
shell> ldd $ORACLE_HOME/bin/oracle
               linux-vdso.so.1 =>  (0x00007fff3c1f1000)
               …

shell> cat /proc/sys/kernel/vsyscall64
1
```

The vDSO boot parameter has three possible values:
0 = Provides the most accurate time intervals at µs (microsecond) resolution, but also produces the highest call overhead, as it uses a regular system call
1 = Slightly less accurate, although still at µs resolution, with a lower call overhead
2 = The least accurate, with time intervals at the ms (millisecond) level, but offers the lowest call overhead

Back to the example and the strace output. The corresponding gettimeofday() or clock_gettime() system calls can be noticed if vDSO is “by-passed”.

```
shell> echo 0 > /proc/sys/kernel/vsyscall64

shell> strace -cf -p 1915
Process 1915 attached - interrupt to quit
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 97.58    0.106397           9     11946           clock_gettime
  1.83    0.001999        1000         2           read
  0.26    0.000282          35         8           times
  0.17    0.000181           9        21           mmap
  0.16    0.000176           8        23           getrusage
  0.00    0.000000           0         2           write
  0.00    0.000000           0         2           munmap
  0.00    0.000000           0         1           gettimeofday
------ ----------- ----------- --------- --------- ----------------
100.00    0.109035                 12005           total
```

System calls to clock_gettime() suddenly appear “without” vDSO, but there is an overhead of round about 0.1 seconds for such a simple „light-weight“ query with rowsource statistics enabled.

Capturing stack traces on Linux

In general several tools are available to capture stack traces of an Oracle process. The following list includes some examples:

- Tool “Oradebug” provided by Oracle and available on all OS platforms
- Tool “GDB” and its wrapper scripts on Linux
- Tool “Perf” and the kernel (>= 2.6.31) interface “perf_events” on Linux
- Tool “Systemtap” on Linux kernel >= 3.5 for userspace otherwise “utrace patch” is needed [5]
- Tool “DTrace” on Solaris
- Tool “Procstack” on AIX

The upcoming sections focus on the tools “Oradebug”, “GDB” and its wrapper scripts and “Perf”.

Oradebug [6]

“Oradebug” is a tool provided by Oracle and primarily used by Oracle support to check or manipulate Oracle internals. The tool itself is mainly undocumented, but provides a good help interface. It also provides the capability to get an abridged call stack of the attached process. Internally “Oradebug” sends a SIGUSR2 signal to the corresponding target process, which captures the signal in function sspuser(). The function sspuser() performs further debugging actions afterwards. [7] Tracing the corresponding system calls confirms the described internal behavior.

```
shell> ps -fu oracle | grep LOCAL=NO
oracle    1870     1  0 10:55 ?        00:00:00 oracleT12DB (LOCAL=NO)

SQL> oradebug setospid 1870
Oracle pid: 43, Unix process pid: 1870, image: oracle@OEL

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

shell> strace -f -p 1870
Process 1870 attached - interrupt to quit
read(35, 0x7fcaa163fece, 2064)          = ? ERESTARTSYS (To be restarted)
--- SIGUSR2 (User defined signal 2) @ 0 (0) ---
…
```

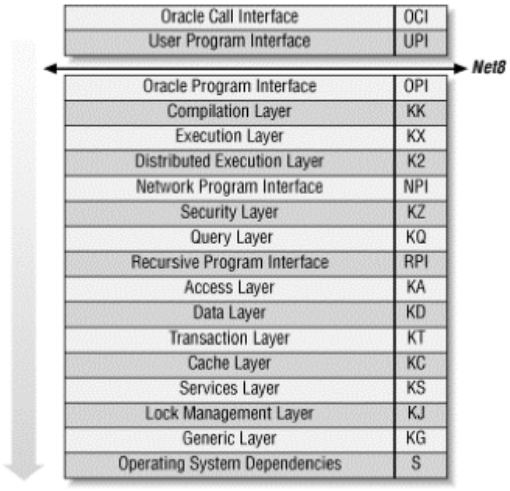
Let's have a look how "Oradebug" works in general with an Oracle process that is in idle mode and waiting for SQL commands via SQL*Net. The following demo is run on Oracle 12.1.0.1 and Linux kernel 2.6.39-400.109.1.el6uek.x86_64.

```
shell> ps -fu oracle | grep LOCAL=NO
oracle    1888     1  0 13:44 ?        00:00:00 oracleT12DB (LOCAL=NO)

SQL> oradebug setospid 1888
Oracle pid: 43, Unix process pid: 1888, image: oracle@OEL

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

A call stack needs to be read bottom up. The first function "__libc_start_main()" performs any necessary initialization of the execution environment like security checks, etc.. In the previous example the Oracle process starts in function main(), which calls function ssthrdmain(), which calls function opimai_real() and so on. You may recognize the main() call structure if you are familiar with writing C code.
However as previously explained everything above and including function sspuser() is caused by the "Oradebug" request. The process itself is currently executing a system call read() which waits for data on the socket (network input). The syntax „+<NUMBER>" specifies the offset in bytes from the beginning of the function (symbol) where the child function is called. This is also the next instruction to continue with when the child function returns. In addition each Oracle (kernel) layer is also present and interpretable in the stack trace. [8]



Graphic source: Book "Oracle 8i Internal Services" – Thankfully approved by Steve Adams [8]

The Oracle program interface (OPI)

Oracle program interface is the highest layer of the server-side call stack. In most configurations, Net8 bridges the gap between UPI and OPI. However, in single-task executables there is no gap, and the UPI calls correspond directly to OPI calls.

```
opikndf2()+995<-opitsk()+803<-opiino()+888<-opiodr()+1170<-opidrv()+586<-sou2o()+120<-
opimai_real
```

A pretty detailed explanation of each OPI function in this stack can be found in Tanel Poder's blog post "Advanced Oracle Troubleshooting Guide, Part 2: No magic is needed, systematic approach will do". [9]

GDB and its wrapper scripts

The “GDB” (GNU debugger) allows you to see what is going on inside another program while it executes or what another program was doing at the moment it crashed. “GDB” can do four main kind of things (plus other things in support of these):

- Start a program, specifying anything that might affect its behavior
- Make a program stop on specified conditions
- Examine what has happened, when a program has stopped
- Change things in a program

“GDB” (and its wrapper scripts) should just be used to capture stack traces by having the focus on this paper.

Let’s have a look how it works in general with an Oracle process that is in idle mode and waiting for SQL commands via SQL*Net (same process state as with “Oradebug”). The following demo is run on Oracle 12.1.0.1 and Linux kernel 2.6.39-400.109.1.el6uek.x86_64.

```
shell> ps -fu oracle | grep LOCAL=NO
oracle    1834     1  1 10:16 ?        00:00:00 oracleT12DB (LOCAL=NO)

(gdb) attach 1834
Attaching to process 1834
Reading symbols from /oracle/rdbms/12101/bin/oracle...(no debugging symbols found)...done.
…

(gdb) bt
#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 0x000000000b393a7 in opimai_real ()
#13 0x0000000002542898 in ssthrdmain ()
#14 0x000000000b392de in main ()
```

The first eye-catching message “no debugging symbols found” is works-as-designed as Oracle is not compiled with debugging information (e.g. gcc option “-g”). This has some impact on debugging Oracle functions (and its parameters), but you can even get the function names without the debug information. In sum this is all we need in case of stack tracing.

The content of the stack trace looks like the “Oradebug” one, expect the signal handling part. However “GDB” also misses the offset byte addresses, but this is also not an issue in this case.

"GDB" also got some wrapper scripts (e.g. pstack / gstack) where only the process id <PID> is handed over but the end result (stack trace output) is the same.

```
shell> which pstack
/usr/bin/pstack

shell> ls -la /usr/bin/pstack
lrwxrwxrwx. 1 root root 6 Jun 26  2013 /usr/bin/pstack -> gstack

shell> file /usr/bin/gstack
/usr/bin/gstack: POSIX shell script text executable


shell> cat /usr/bin/gstack
#!/bin/sh
…
# Run GDB, strip out unwanted noise.
$GDB --quiet $readnever -nx /proc/$1/exe $1 <<EOF 2>&1 |
…
$backtrace
…

shell> pstack 1834
#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 ()
```

Performance counters for Linux (Linux kernel-based subsystem)

Capturing stack traces with the Oracle tool "Oradebug" or with a C debugger like "GDB" got one common main issue. It needs to be done manually and it is just a snap-reading method, but the target is to know where most of the (CPU) time is spent or if a process is stuck in a specific C function. A lot of stack trace samples need to be gathered in high frequency and afterwards they need to be aggregated to get the whole picture. Tools like "Oradebug" or "GDB" are usually not fast enough to sample the stack traces at very high frequency, but luckily the Linux kernel provides a solution for this called „Performance counters for Linux".

Perf (sometimes "Perf Events", originally "Performance Counters for Linux") is a performance analyzing tool in Linux. It is available with kernel version 2.6.31 or higher. Performance Counters for Linux (PCL) is a kernel-based subsystem that provides a framework for collecting and analyzing performance data. The PCL subsystem can be used to measure hardware events, including retired instructions and processor clock cycles. It can also measure software events, including major page faults and context switches. For example PCL counters can compute the Instructions Per Clock (IPC) from a process's counts of instructions retired and processor clock cycles. A low IPC ratio indicates the code makes poor use of the CPU. Other hardware events can also be used to diagnose poor CPU performance.
However be very careful with the specifically used events (hardware respectively software) due to potential incorrect measurements in virtualized environments. [10] [11]

The tool “Perf” itself is based on the perf_events interface which is exported by recent versions of the Linux kernel.

The common used perf option (“perf record”) for Oracle processes is based on sampling or better said perf_events is based on event-based sampling in such cases. The perf_events interface allows two modes to express the sampling period:
- The number of occurrences of the event (period)
- The average rate of samples/sec (frequency)

The tool “Perf” defaults to the average rate, which is set to 1000Hz or 1000 samples/sec. It means that the kernel is dynamically adjusting the sampling period to achieve the target average rate. In contrast, with the other mode, the sampling period is set by the user and does not vary between samples. In general the default setting is sufficient for troubleshooting Oracle CPU issues in most cases.

Let’s have a look how it works in general with an Oracle process that executes some SQL statement and fetches data via SQL*Net. This demo case includes some workload as it makes no sense to profile and sample an idling Oracle process (no stack traces are captured if the process is not on CPU and even if it would work this way the call stack would always be the same).
The following demo is run on Oracle 12.1.0.1 and Linux kernel 2.6.39-400.109.1.el6uek.x86_64.

```
SQL> create table TEST as select * from DBA_SOURCE;
SQL> select /*+ use_hash(t2) gather_plan_statistics */ count(*) from TEST t1, TEST t2
     where t1.owner = t2.owner;

shell> ps -fu oracle | grep LOCAL=NO
oracle    1883     1  6 09:54 ?        00:00:03 oracleT12DB (LOCAL=NO)

shell> perf record -e cpu-cycles -o /tmp/perf.out -g -p 1883
[ perf record: Woken up 47 times to write data ]
[ perf record: Captured and wrote 11.597 MB /tmp/perf.out (~506674 samples) ]

shell> ls -la /tmp/perf.out
-rw------- 1 oracle dba 12162816 Sep 19 09:56 perf.out
```

The previous perf command has captured round about 506.674 samples and has written the trace data to output file “/tmp/perf.out”. This file can be processed later on with various tools to interpret the output.
“Perf” was also instructed to use the hardware event “cpu-cycles” (parameter “-e“) which basically means that the kernel’s performance registers are used to gather information about the process that is running on CPU. “Perf” will automatically fallback to the software event “cpu-clock” which is based on timer interrupts if the hardware event “cpu-cycles” is not available. [10] [11]

Poor man's stack profiling

Sometimes just a quick half-hierarchical and aggregated stack profile is needed or Oracle is running on a platform without any (installed) profiler like "Perf". The following solution is pretty slow as it is based on "GDB" or better said on its wrapper script "pstack". In consequence very high sample rates (like with tool "Perf") can not be reached.

Let's have a look how it works in general with an Oracle process that executes some SQL statement and fetches data via SQL*Net. The following demo is run on Oracle 12.1.0.1 and Linux kernel 2.6.39-400.109.1.el6uek.x86_64.
The used script "os_explain" by Tanel Poder [12] decodes specific well known C functions to useful meanings (e.g. qerhj* to HASH JOIN, etc.).

```
SQL> create table TEST as select * from DBA_SOURCE;
SQL> select /*+ use_hash(t2) */ count(*) from TEST t1, TEST t2 where t1.owner = t2.owner;

shell> ps -fu oracle | grep LOCAL=NO
oracle    4663     1  2 11:12 ?        00:00:00 oracleT12DB (LOCAL=NO)

shell> export LC_ALL=C ; for i in {1..20} ; do pstack 4663 | ./os_explain -a ; done |
       sort -r | uniq -c
 20    main
 20     ssthrdmain
 20      opimai_real
 20       sou2o
 20        opidrv
 20         opiodr
 20          opiino
 20           opitsk
 20            ttcpip
 20             opiodr
 20              kpoal8
 20               SELECT FETCH:
 20                GROUP BY SORT: Fetch
 20                * TABLE ACCESS: Fetch
 20                 HASH JOIN: Fetch
 20                    kdsttgr
 20                     kdstf000010100001km
 20                      HASH JOIN: InnerProbeHashTableRowset
 20                       HASH JOIN: InnerProbeProcessRowset
 20                        HASH JOIN: WalkHashBucket
  3                         qesrAddRowFromCtx
  7                         kxhrUnpack
  5                         kxhrPUcompare
  1                         HASH JOIN: ReturnRowset
  5                          rworupo
  1                          _intel_fast_memset.J
  2                          _intel_fast_memcmp
  4                           qksbgGetCursorVal
```

In the previous example 20 stack samples were taken with "GDB" and the base function "HASH JOIN: WalkHashBucket" was processed all the time. So basically the CPU usage is caused by the hash join (respective qerhj* functions) in this very simplistic case. However this is not very surprising as the SQL is hinted to perform exactly this action, but this example should only demonstrate the general procedure.

Interpreting stack traces on Linux

Visualization and interpretation of stack traces is dependent on the source data. The profiling tool "Perf" can write its output to a file which can be processed later on with various tools.
This section introduces the reporting functionality of "Perf" and one of the most well known tools (also my personal favorite) for interpreting stack traces called "Flame graph" by Brendan Gregg. [13]

Performance counters for Linux with "Perf" tool

In the previous chapter an Oracle process (PID 1883) was already profiled and the traced call stacks were written to file "/tmp/perf.out". In this section this file will be processed.

```
shell> perf report -i /tmp/perf.out -g none -n --stdio
# ========
# captured on: Sat Sep 19 10:01:21 2015
# hostname : OEL
# os release : 2.6.39-400.109.1.el6uek.x86_64
# perf version : 2.6.32-358.11.1.el6.x86_64.debug
# arch : x86_64
# nrcpus online : 3
# nrcpus avail : 3
# cpudesc : Intel(R) Core(TM) i7-2675QM CPU @ 2.20GHz
# cpuid : GenuineIntel,6,42,7
# total memory : 3605136 kB
# cmdline : /usr/bin/perf record -e cpu-cycles -o /tmp/perf.out -g -p 1883
# event : name = cpu-clock:HG, type = 1, config = 0x0, config1 = 0x0, config2 = 0x0,
          excl_usr = 0, excl_kern = 0, excl_host = 0, excl_guest = 0, precise_ip = 0,
          id = { 7 }
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to display
# ========
#
# Samples: 45K of event 'cpu-clock:HG'
# Event count (approx.): 45248
#
# Overhead  Samples            Command      Shared Object                        Symbol
# ........ ..........  ................  ..................  ..........................
#
  27.54%      12462  oracle_1883_t12  oracle              [.] qerhjWalkHashBucket
  14.44%       6534  oracle_1883_t12  oracle              [.] kxhrPUcompare
  14.21%       6429  oracle_1883_t12  oracle              [.] rworupo
  10.17%       4602  oracle_1883_t12  oracle              [.] qesrAddRowFromCtx
   9.49%       4294  oracle_1883_t12  oracle              [.] qksbgGetCursorVal
  …
   0.01%          3  oracle_1883_t12  librt-2.12.so       [.] clock_gettime
   0.01%          3  oracle_1883_t12  [vdso]              [.] 0x00007fffecbff9c1
  …
   0.00%          1  oracle_1883_t12  [kernel.kallsyms]   [k] kref_get
   0.00%          1  oracle_1883_t12  [kernel.kallsyms]   [k] scsi_run_queue
```

The trace file was analyzed with "Perf" and option "-g none", which basically just prints out the CPU consuming functions at the top of the call stack.
"Perf record" was started with the hardware event "cpu-cycles", but the samples were taken with the software event "cpu-clock" as my platform (Virtual Box) does not support the kernel's performance registers at the time of writing. This silent fallback was already mentioned in the previous chapter.

The application (userspace) functions like "qerhjWalkHashBucket" or "kxhrPUcompare" are marked with "[.]" in contrast to the OS kernel space functions like "kref_get" or "scsi_run_queue" which are marked with "[k]". The vDSO functions are also included of course.

As a result 27.54% of the CPU time was spent in Oracle (user space) function "qerhjWalkHashBucket", 14.44% of the CPU time was spent in Oracle (user space) function "kxhrPUcompare" and so on. This method enables to drill down the CPU consumer in detail.

“Perf report” also has the possibility to include the whole stack trace in the output. The following output is truncated to the top userspace functions “qerhjWalkHashBucket” and “rworupo” for demonstration purpose.

```
shell> perf report -i /tmp/perf.out -g graph -n --stdio
…
# Overhead  Samples            Command      Shared Object                          Symbol
# ........ ..........  ................  ..................  ............................
#
    27.54%      12462  oracle_1883_t12  oracle              [.] qerhjWalkHashBucket
        |
        --- qerhjWalkHashBucket
            qerhjInnerProbeProcessRowset
            qerhjInnerProbeHashTableRowset
            qerstRowP
            qerstRowP
            kdstf000010100001km
            kdsttgr
            qertbFetch
            qerstFetch
            rwsfcd
            …
            ssthrdmain
            main
            __libc_start_main
            …
    14.21%       6429  oracle_1883_t12  oracle              [.] rworupo
        |
        --- rworupo
            |
            |--13.54%-- kxhrUnpack
            |           qerhjWalkHashBucket
            |           qerhjInnerProbeProcessRowset
            |           qerhjInnerProbeHashTableRowset
            |           qerstRowP
            |           qerstRowP
            |           kdstf000010100001km
            |           kdsttgr
            |           qertbFetch
            |           qerstFetch
            |           rwsfcd
            |           …
            |           ssthrdmain
            |           main
            |           __libc_start_main
            |
             --0.67%-- qerhjWalkHashBucket
                       qerhjInnerProbeProcessRowset
                       qerhjInnerProbeHashTableRowset
                       qerstRowP
                       qerstRowP
                       kdstf000010100001km
                       kdsttgr
                       …
                       ssthrdmain
                       main
                       __libc_start_main
```

The whole stack trace reveals how the top CPU consuming functions are called. For example:

- Function “qerhjWalkHashBucket” (where 27.54% of the CPU time is spent) is just called from one stack (function “__libc_start_main” up to “qerhjInnerProbeProcessRowset”, which finally calls “qerhjWalkHashBucket”).
- Function “rworupo” (where 14.21% of the CPU time is spent) is called from two different child functions (= two different stacks). In this case from stack function “__libc_start_main” up to “kxhrUnpack”, which finally calls “rworupo” and from stack function “__libc_start_main” up to “qerhjWalkHashBucket”, which finally calls “rworupo”. “Perf” also shows a break down of the CPU time to these different stacks (e.g. 13.54% from 14.21% overall CPU time is spent on stack from stack function “__libc_start_main” up to “kxhrUnpack”).

The Oracle (kernel) application function names are cryptic, but some can be translated with help of MOS ID #175982.1 - “ORA-600 Lookup Error Categories”. The note was deleted by Oracle some time ago, but it still can be found in the world wide web very easily.

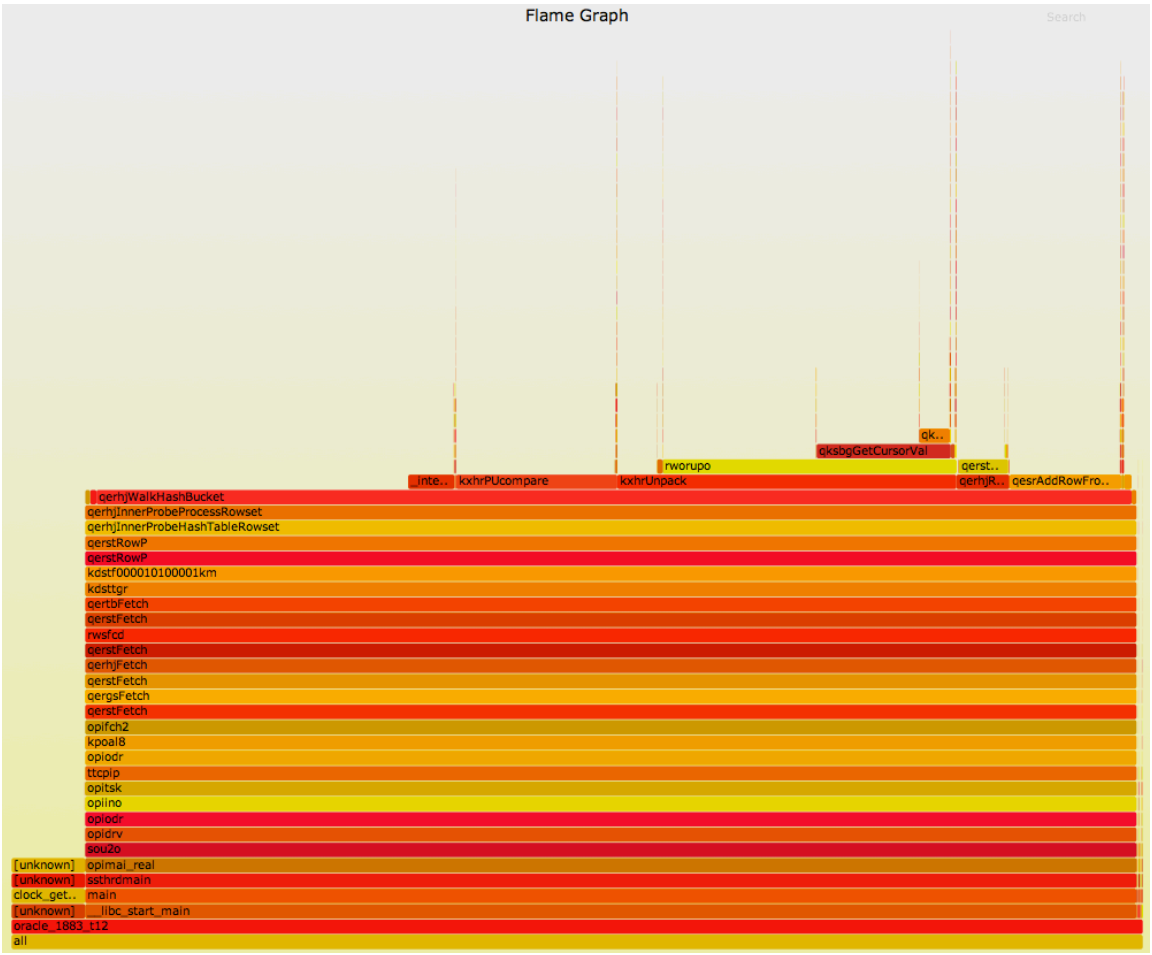
However interpreting stack traces with the tool “Perf” has one major issue. Basically all we need to know is where the bulk of the CPU time is spent, but the track can be lost very easily, if the CPU consuming functions are called from various stacks and if the calling functions are consuming a lot of CPU too. The previous case is a very simple stack, but it already demonstrates such a situation. Function “qerhjWalkHashBucket” causes 27.54% of the CPU time, but this function is also included in the stack of the other top consuming function “rworupo”. So basically the CPU consumption of function “rworupo” may indirectly caused (as called) by “qerhjWalkHashBucket” as well, but this is not obvious at the first sight. It is just too much data to interpret with tool “Perf” in more complex scenarios (and especially with a lot of similar call stacks).

Flame graph by Brendan Gregg [13]

(CPU) flame graph by Brendan Gregg solves the issue with “perf report” that was mentioned previously. Flame graph is a visualization of profiled software, allowing the most frequent code-paths to be identified quickly and accurately. The output is an interactive SVG graphic.

In the previous chapter an Oracle process (PID 1883) was already profiled with tool “Perf” and the sampled and traced call stacks were written to file “/tmp/perf.out”. In this section the file (the same file that was previously processed with “perf report”) will be processed with flame graph.

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



The following is a description of the graph layout by Brendan Gregg:

- Each box represents a function in the stack (a "stack frame").
- The y-axis shows stack depth (number of frames on the stack). The top box shows the function that was on-CPU. Everything beneath that is ancestry. The function beneath a function is its parent, just like the stack traces shown earlier.
- The x-axis spans the sample population. It does not show the passing of time from left to right, as most graphs do. The left to right ordering has no meaning (it's sorted alphabetically to maximize frame merging).
- The width of the box shows the total time it was on-CPU or part of an ancestry that was on-CPU (based on sample count). Functions with wide boxes may consume more CPU per execution

than those with narrow boxes, or, they may simply be called more often. The call count is not shown (or known via sampling).
- The sample count can exceed elapsed time if multiple threads were running and sampled concurrently.

Zooming into the stacks and searching for some specific function is also possible with the newest flame graph version.
However the graphic reveals what was mentioned in the previous chapter. The function "qerhjWalkHashBucket" is consuming 27.54% of the CPU time on its own, but it is basically "active" at the whole time as it also calls all the other functions like "kxhrPUcompare" (+ on top functions), "kxhrUnpack" (+ on top functions), "qerhjReturnRowset" (+ on top functions) and so on. Flame graph solves this lack of information in the "Perf" output and allows a quick identification of the potential base issue.

Safety warning! Are debuggers and capturing "stack traces" safe to use (in production)?
The safety of debugging Oracle depends on the used approach or tools and may be dangerous as the corresponding process might crash or get some ORA errors without any obvious reason. It usually does not matter when the database is already completely stuck and nobody can work anymore, but do not use some specific debuggers on critical background processes (in production) if only some sessions or processes are affected.

Tool "Oradebug" = Unsafe

This tool alters the execution path (e.g. "ksedsts()+213<-ksdxfstk()+34<-ksdxcb()+914< sspuser()+191<-__sighandler()") of the attached process when dumping the call stack. This might crash the attached process (or the whole database) in case of an Oracle bug. An example of such an issue with LGWR would be Oracle bug #15677306 - "ORACLE LOGWRITER HARD HANG WHEN SIGUSR INTERRUPTS LIBAIO".

Tool "GDB" and its wrapper scripts = Unsafe

These tools suspend the process to get a call stack by attaching via ptrace() syscall. The attached process may be affected by ptrace() when communicating with the operating system kernel or other processes (e.g. issues with sending/receiving I/O interrupts, etc.).

Do not confuse it with pstack on Solaris or procstack on AIX. These tools do not rely on ptrace() syscalls and are safe to use. Basically both tools just read the stack information from the /proc file system and memory.

However be aware that the tool "strace" is also based on ptrace(). I personally have never experienced an Oracle process crash due to "strace", but it is possible in theory as well. After a short chat with Tanel Poder, he also confirmed that he was suffered by crashing processes with "strace" under certain circumstances.

Performance counters for Linux (Linux kernel-based subsystem) = Safe

Performance counters for Linux respectively "Perf" relies on its own kernel interface and do not use ptrace(). It also does not alter the execution path of a process or suspend it. So basically it is safe to use. However the tool "Perf" only samples stacks of processes that are running on CPU (e.g. like in the previous demo case with CPU intensive functions that are used by the hash join). A fallback to the other tools is still needed, if the process is not running on CPU and stuck somewhere else.

Combine the Oracle wait interface and stack traces [14]
This paper described detailed CPU consumption analysis so far, but combining the Oracle wait interface (e.g. for I/O wait states, etc.) and "Perf" analysis in one shot is also possible. This is extremely useful, if the whole response time picture of a process is needed. Craig Shallahamer and Frits Hoogland developed a script called "fulltime.sh" that is based on the Oracle wait interface (v$session_event) and performance counters for Linux ("Perf").

Let's have a look how it works in general with an Oracle process that executes some SQL statement and fetches data via SQL*Net. The following demo is run on Oracle 12.1.0.1 and Linux kernel 2.6.39-400.109.1.el6uek.x86_64.

```
SQL> select /*+ use_hash(t2) */ count(*) from TEST t1, TEST t2 where t1.owner = t2.owner;

shell> ps -fu oracle | grep LOCAL=NO
oracle    1861     1  0 10:40 ?        00:00:00 oracleT12DB (LOCAL=NO)

shell> ./orapub_fulltime_cpu_wait_perf.sh 1861 10 1

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
```

The output shows a detailed break down of the CPU usage and the additional wait events that occur in the captured time window of 10 seconds (second script parameter).
The wait event "SQL*Net message from client" is on top as the SQL was started after the profiling script "orapub_fulltime_cpu_wait_perf.sh". All the other time components are related to the SQL execution.

In addition Oracle 12c expanded its kernel diagnostics & tracing infrastructure with the diagnostic event "wait_event[]". This event provides the capability to capture a stack trace after a specific wait event has occurred and so it interacts with the general Oracle wait interface. This may be useful for analyzing and drilling down the Oracle code path for/after a specific wait event.

The syntax for this new diagnostic event is the following:

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

Let's have a look how it works in general with an Oracle process that executes some SQL statement and fetches data via SQL*Net. The following demo is run on Oracle 12.1.0.1 and Linux kernel 2.6.39-400.109.1.el6uek.x86_64.

```
SQL> alter session set events 'wait_event["direct path read"]  trace("%s\n",
     shortstack())';
SQL> exec DBMS_MONITOR.SESSION_TRACE_ENABLE(NULL,NULL,TRUE,FALSE,'NEVER');
SQL> select /*+ use_hash(t2) */ count(*) from TEST t1, TEST t2 where t1.owner = t2.owner;

Trace file
====================
PARSING IN CURSOR #140311192202592 len=83 dep=0 uid=116 oct=3 lid=116 tim=589822292
hv=3504242112 ad='6f59e608' sqlid='0w1v4xg8dwzf0'
select /*+ use_hash(t2) */ count(*) from TEST t1, TEST t2 where t1.owner = t2.owner
END OF STMT
PARSE #140311192202592:c=20997,e=76035,p=1,cr=36,cu=0,mis=1,r=0,dep=0,og=1,
       plh=3302976337,tim=589822283
EXEC #140311192202592:c=0,e=121,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,
      plh=3302976337,tim=589822564
WAIT #140311192202592: nam='SQL*Net message to client' ela= 13 driver id=1413697536
     #bytes=1 p3=0 obj#=19689 tim=589822643
WAIT #140311192202592: nam='Disk file operations I/O' ela= 53 FileOperation=2 fileno=4
     filetype=2 obj#=92417 tim=589823314
WAIT #140311192202592: nam='db file sequential read' ela= 7470 file#=4 block#=178 blocks=1
     obj#=92417 tim=589830842
WAIT #140311192202592: nam='direct path read' ela= 2166 file number=4 first dba=179 block
     cnt=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
WAIT #140311192202592: nam='direct path read' ela= 26985 file number=4 first dba=193 block
      cnt=15 obj#=92417 tim=590021945
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
…
```

In this example a call stack is dumped every time after the wait event "direct path read" has occurred. The function kslwtectx() is called at the end of a wait event (in this case "direct path read") and the rest of the call stack is related to the I/O request itself.

Real life root cause identified and fixed with help of stack traces
A real life root cause analysis with stack tracing is described in one of my blog posts. The whole issue including the solution is described here: http://scn.sap.com/community/oracle/blog/2013/01/10/oracle-advanced-performance-troubleshooting-with-oradebug-and-stack-sampling

References

[1] http://shop.oreilly.com/product/9780596005276.do
[2] http://blog.tanelpoder.com/files/scripts/snapper.sql
[3] http://man7.org/linux/man-pages/man2/syscalls.2.html
[4] https://lwn.net/Articles/446528/
[5] https://fritshoogland.wordpress.com/2014/04/27/systemtap-revisited/
[6] http://www.orafaq.com/papers/oradebug.pdf
[7] http://blog.tanelpoder.com/2008/10/31/advanced-oracle-troubleshooting-guide-part-9-process-stack-profiling-from-sqlplus-using-ostackprof/
[8] http://shop.oreilly.com/product/9781565925984.do
[9] http://blog.tanelpoder.com/2007/08/27/advanced-oracle-troubleshooting-guide-part-2-no-magic-is-needed-systematic-approach-will-do/
[10] https://fritshoogland.wordpress.com/2013/12/17/when-the-oracle-wait-interface-isnt-enough-part-2-understanding-measurements/
[11] http://kb.vmware.com/selfservice/microsites/search.do?cmd=displayKC&externalId=2030221
[12] http://blog.tanelpoder.com/files/scripts/tools/unix/os_explain
[13] http://www.brendangregg.com/flamegraphs.html
[14] http://resources.orapub.com/Fulltime_sh_Report_Oracle_Wait_and_CPU_Details_p/fulltime-sh.htm

Contact address:

Stefan Koehler
Freelance Consultant (Soocs)
Gustav-Freytag-Weg 29
D-96450 Coburg, Germany

Phone: +49 (0) 172 / 1796830
E-Mail contact@soocs.de
Internet: http://www.soocs.de
Twitter: @OracleSK