Hunting Down CPU related issues with Oracle: A functional Approach

Nilendu Misra
nilendu@nilendu.com

This article discusses several functional approaches one might take while analyzing, diagnosing and solving CPU related problem issues. The discussion covers most flavors of Unix and Oracle 8i. TOAD is the only third party software used for test cases while this document was prepared.

INTRODUCTION

Most of the Operating System related bottlenecks could be tracked down to one or a combination of the following:

1. Disk I/O Contention
2. Memory Problem
3. CPU Issues

This article will confine the discussion to approach the CPU related issues only.

An easy way to solve the issue is to increase the capacity of the machine. For instance, to solve an ongoing CPU spike, the CPU count is improved. This works somewhat satisfactorily on a scaled-down system. A scaled-down system is one that was not designed to withstand the full load as it experiences today. But for carefully designed system, which tried to look forward to the load limit, this 'adding capacity' is a bane! It either proves the earlier design faulty or it cannot stop recurrence of the problem that caused one to go for added capacity. Thereby it renders the entire design activity and cost incurred as a bad decision.

For carefully designed systems, often the design phase is horizontally split. A consultant expert in the particular area designs each component among Network, Operating System, Database and Application separately. Later, the whole thing is integrated in a bottom-up approach. This more than often leaves a blazing hole between two design areas. For instance, the scale of application might not match the scale of Operating System. So for this type of system it is imperative that one spends considerable time and effort after dissecting the problem before doubling (or quadrupling) the capacity. The issue has to be **understood, analyzed, diagnosed and finally resolved** to take a decision. The resolution might be a correction of error that led to extra consummation of resource, or a change in design or increased spending after infrastructure.

In this paper we will discuss how to go with these four stages mentioned above with respect to a CPU problem.

UNDERSTANDING of the CPU issue

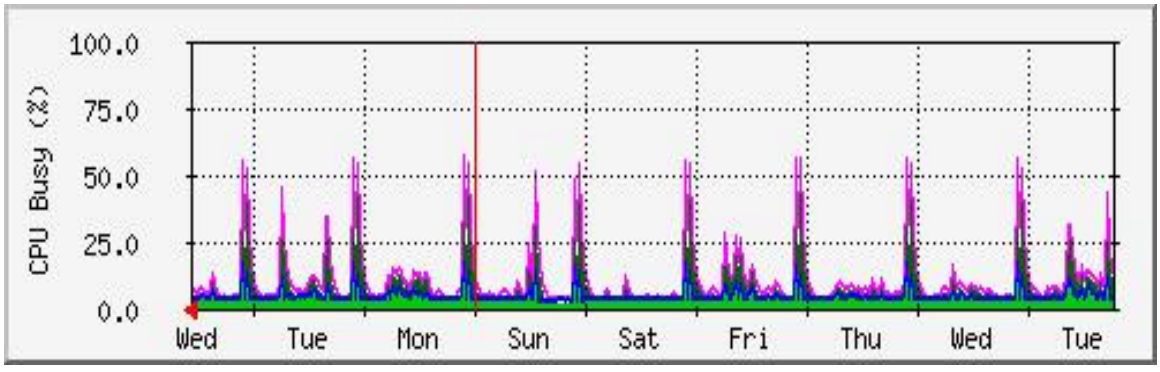
The traditional way to tune starts from the point of receiving complaints! It works just well if most of the system actions are already being taken in a proactive basis. The issue starts either followed by *a distinct complaint from the user of the system or as a result of the regular maintenance scheme* exercised by the DBA on the system.

A standard checklist to detect if everything is OK regarding CPU could be:

- Know what's your CPU configuration – How many and what MHz (or GHz)!
- If the Server Side CPU is heavily used
- If the client side CPU is heavily used
- If the CPU 'spike' happens on some particular point in time (i.e., during night when no one does anything on the database a CPU usage of over 15% could be worrisome)
- What's the CPU usage on peak load?
- What's the CPU usage on idle state?
- If the issue is known not to be generic, i.e., spike reported during particular query execution – even then these points should be checked for the system

In this particular case, I used a web-based traffic-reporting tool called MRTG (Multi Router Traffic Grapher) that is freely available under GNU license. This was activated to keep track of CPU load on the system (on all the tier) to report the CPU busy activities after a given time interval. On the single machines, Unix commands like ***vmstat 10 10***, ***sar -qu 5 5, top*** were used to watch on both during peak load and off-peak load.

MRTG can be worked upon to produce graph like this online. The statistics can be taken in various intervals and graphs can be produced for archival value.:



A command shows the output in the following format –

$ vmstat 5 5

```
 procs     memory          page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr f0 m0 m1 m3   in   sy   cs us sy id
 0 0 0 2286936 291672 1   6  6  0  0  0  0  0  0  0  0 4294967260 224 80 6 1 93
 0 0 0 2106152 115288 0   1  0  0  0  0  0  0  0  0  0  292  224  116  2  1 98
 0 0 0 2106152 115288 0   0  0  0  0  0  0  0  2  0  0  292   86   49  1  3 96
 0 0 0 2106152 115288 0   0  0  0  0  0  0  0  0  0  0  306  233   77  3  0 97
 0 0 0 2106152 115288 0   0  0  0  0  0  0  0  0  0  0  252  123   50  0  0 99
```

Here the first line is the average for each statistic since the start. The last few columns report CPU activity.

us – reports % of time spent for user cycles
sy - reports % of time spent for system cycles
id - reports % of time that went unutilized during this time

For instance, in the above the CPU was idle for around 93% of the time vmstat was run for. A good thumb rule is to aim for less than 50% of CPU time spent in system mode. That would indicate that the system spends too much time in kernel mode servicing interrupts, swapping processes etc.

top is another useful command which can be used to track down CPU usage. It shows CPU statistics in the following format :

CPU states: 96.4% idle, 1.2% user, 0.6% kernel, 1.8% iowait, 0.0% swap

However, decision to use should be carefully taken as that itself is a resource-intensive process.

As a matter of fact, high usage of CPU should not be a bad thing in itself. It could just mean the system is well utilized for which it was built. But a consistently high CPU value without much load or the general system performance problem coupled with high CPU usage definitely makes the issue worth a look.

Other commands that are just as useful are ***ps –ef*** and ***uptime***.

While the server side CPU might almost always indicate problem with query, a CPU spike in one of the ‘client machines’ (for instance, web server) might indicate there is a problem in network, or the client does some extra database processing which should be traced.

Before analyzing the issue further baseline CPU usage has to be found. A baseline CPU usage of 15% indicates that if the peak CPU usage is around 85% then the load itself causes around 70% usage on the CPU.

ANALYZING the CPU issue

A checklist in this area could contain the following:

- If there is a problem to be addressed!
- Was the system behaving 'well' at any point before?
- How much of the CPU load is attributed to Oracle (our discussion will concentrate on this area from now on)
- How much CPU load is extra-Oracle?
- Can somehow the 'spike' be reproduced to happen?
- If on Oracle, does it generally slow down the database and all transactions?
- Does it happen on peak load, usual load or even in off-peak load?
- Can running a particular application module could cause the spike?

Analysis phase would help us to focus on the 'core' area of the problem – which the paper would assume to be some Oracle process!

DIAGNOSING the CPU issue

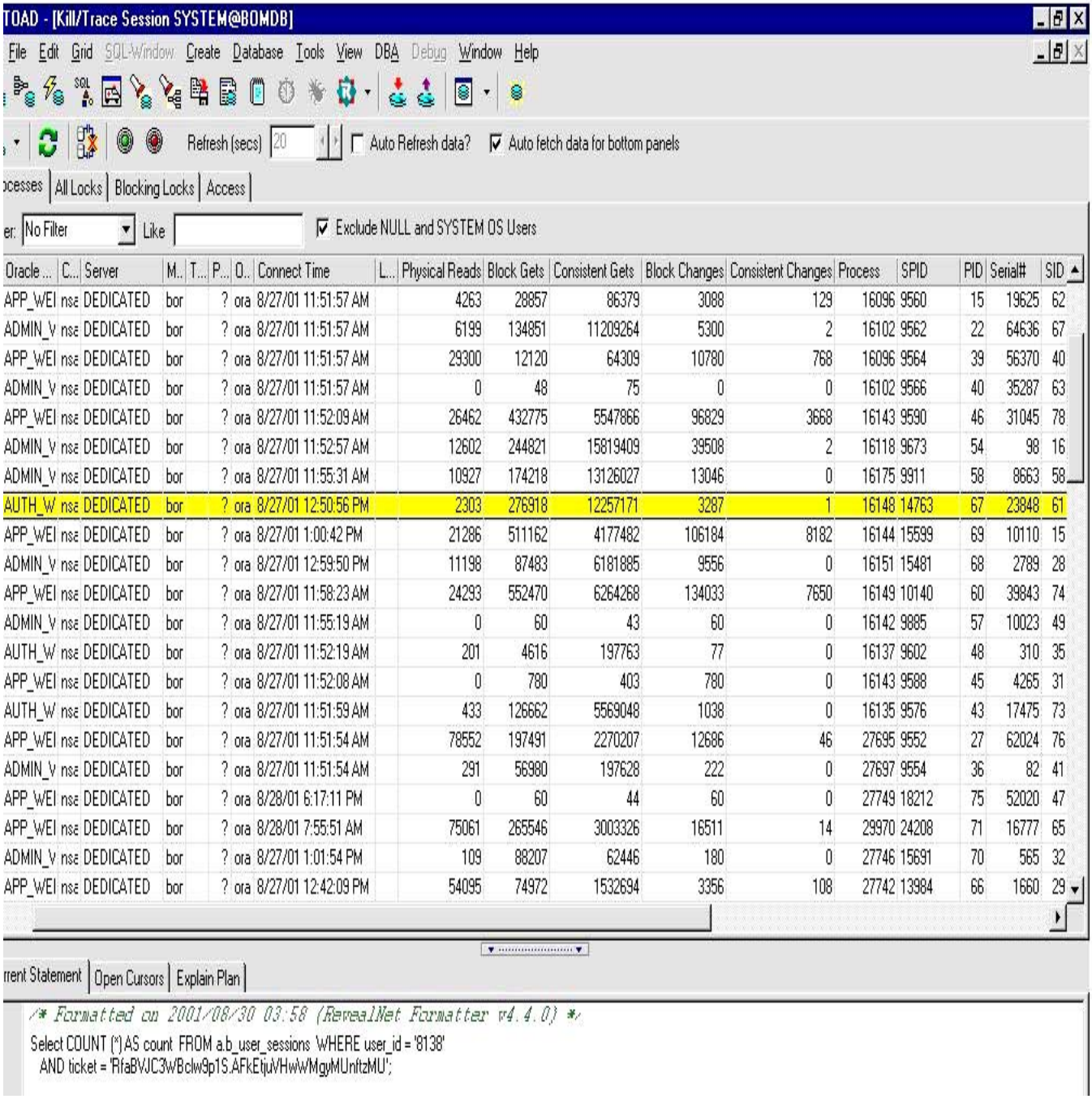
This can be done in various ways.

Method ONE – To see the particular session using a Tool

By doing a general 'sar' or 'top' on the server find out the session which consumes maximum of CPU slice. Please be careful with 'top' on the production server, as you will often notice 'top' itself consumes the most resource.

```
  PID USERNAME THR PRI NICE  SIZE   RES STATE    TIME    CPU COMMAND
14763 oracle     1  59  -20  280M  261M sleep   12:02  12.75% oracle
11670 oracle     1  59  -20  281M  261M sleep    6:24   7.70% oracle
 8369 oracle     1  58    0 2664K 1720K cpu/2    0:00   2.45% top
 9554 oracle     1  59  -20  280M  261M sleep    0:39   1.08% oracle
```

Now the PID in this report (14763 for instance) can be mapped with a particular SQL statement! I personally prefer TOAD. Login TOAD as a DBA user. Go to Kill / Trace Session. It will be a window like this –



Note: the PID gotten from TOP (14763) is the SPID in the TOAD window. We also get the SID of the session (61) and the SQL statement this session is executing! Now you can even explain plan it by clicking the right menu bar on the lower window and tune the SQL.

Method TWO – To see the particular session writing queries

(1) The first step is same as before. After getting the PID of the most CPU cosuming Oracle process we have got to match that with SPID (a column in Data Dictionary view V$PROCESS). V$SESSION contains SID- Session Identifier which Oracle understands and uses to manipulate with individual sessions.

(2) So the query to get the session details (SID) from OS PID (SPID) would look like :

```
select s.sid from v$process p, v$session s
where s.paddr=p.addr and p.spid = 14763;
```

(3) After we get to know SID, we can easily track down the SQL statement that is being executed by that session. V$SQLTEXT contains the information about SQL statement each session is running. V$SQLTEXT_WITH_NEWLINES can store even bigger statements.

```
SELECT SQL_TEXT from V$SQLTEXT_WITH_NEWLINES where HASH_VALUE
=  (select sql_hash_value from v$session
where SID = <problem_SID_you_got_from_last_step>) ;
```

Now we have the SQL statement which is caused us burn the oil on weekends! Trace and Tune it accordingly!!

Note : Remember for a PL/SQL code the CPU statistics are available to Oracle only after full execution of the same. So you will be lucky only if you are running a SQL. For PL/SQL there are other processes.

Method THREE – To see the session consuming most amount of CPU

(1) Find out the session consuming most amount of CPU by executing

```
SELECT * FROM v$sesstat WHERE statistic# = 12 order by value;
```

(2) Since now we have SID we can easily verify what SQL the particular session is executing (look at Step# 3 of Method 2 above).

Method FOUR – To see the session consuming most amount of CPU

Oracle has given few very useful and powerful scripts to be used by experienced DBA. However, officially Oracle doesn't support use of these scripts but as read-only scripts they are just fine to be run on any system. The script in the present context is named hSession.sql, Full set of h*.sql scripts may be downloaded from the Metalink site.

These scripts together creates several packages in SYS schema. The usage for hSession is as follows :

```
set serverout on
execute hsession.top('CPU used by this session',interval);
```

The package snapshots the statisic value for ALL sessions when called, sleeps for <interval> seconds and then samples the statistic a second time. The output is the top 10 sessions for the given <statistic name> in the given <interval>. All statistics name exactly as given in V$STATNAME could be used as parameter.

Method FIVE – To see the details of CPU consumption own session

```
select n.name,s.value
from v$statname n,V$sesstat s
where n.statistic# = s.statistic#
and value > 0
and s.sid = (select a.sid from v$process p,v$session a
where p.addr =a.paddr
and a.terminal = userenv('terminal'))
order by n.class,n.name
/
```

Once CPU consumed by a particular session is known, the WAIT statistics for the session (time the session waits for various resources) should be read from V$SESSTAT. Then the event that makes the session wait for maximum % of time of the total CPU time for that session probably is the key problem area. It could be – for example – the wait for enqueue. In that case the session might be having an issue with locking!

Note: V$SYSSTAT and V$SESSTAT keep CPU related statistics on System- and Session-level respectively.

RESOLVING the issue

Rest of the process deals with tuning activity for the particular SQL statement. Which is beyond scope of current discussion. But to go ahead, the easiest way is to Time the SQL statement execution. Compare the time of execution (SQL> SET TIMING ON) with the CPU time for parsing, fetch and execution of the statement. The later data could be collected from the analyzed trace files. TIMED_STATISTICS must be on to get the CPU statistics from the trace file through TKPROF.

What are the reasons Oracle could consume much of CPU?

1. *High amount of Parsing* – Parsing accounts for almost 80% of the cost involved with execution of a SQL statement. Parsing should be as much avoided as possible. Re-parsing similar SQL statements is an unnecessary load for the system. However, while analyzing trace files it should be remembered that at least one parse in the trace files cannot be avoided. This one is called '**soft parse**'. After a SQL is fired it's hashed and the hashed value is matched with the hash values present in Library Cache (Shared SQL Area). This is 'soft parse'. As evident a soft parse in inevitable. All SOFT parses ARE counted and listed in TKPROF trace files! So at least ONE parse per statement in the TRACE file is just for hashing the statement and comparing the hashed value with those present in Library Cache.

   To find out statements which are doing too many parses, following query can be executed -

   select sql_text, parse_calls, executions from v$sqlarea order by parse_calls desc;

   To see total number of hard parsing against total number of Parsing happening in the system -

   select name, value from v$sysstat where name = 'parse count%';

   Remember, only HARD parse count can be reduced. Parsing could be reduced by using bind variables, increasing number of cached cursors per session etc.

   2. *Badly written SQL*. Even main memory access is very expensive in terms of CPU cycles, and accounts for most of Oracle's CPU usage. SQL that has excessive I/O requirements will also heavily load on CPU. This is because most of the overhead of Oracle logical I/O occurs in memory. In many times, a query that doesn't use an Index will cause huge CPU load. Proper indexing will decrease the load manifold.

   Similarly, too big sorts taking place in memory could also spike CPU. This is why statements with large numbers of buffer gets as these are typically heavy on CPU. To find those query V$SQLAREA for buffer_gets.

   3. *Other Wait Events*. If the parse CPU is only a small percentage of the total CPU used then the next task is to determine where the CPU is being consumed the most. Query the V$SESSTAT and V$SYSSTAT to compare the wait_events against the total CPU time. Ignore waits for *idle events* like SQL*Net Message, SMON and PMON timer etc.

   .

A Real Life Example

UNDERSTANDING

I have had experience of dealing with a system which used to consume around 20% CPU on even completely idle state (baseline CPU utilization). This was really strange as the peak-load CPU utilization didn't usually go above 70%. This is where a graphic monitoring tool comes really handy. Thanks to MRTG the baseline CPU statistics and peak CPU statistics were observed. Standard Unix tools were run again and again to take a sample on the server over a long-enough time interval.

ANALYZING

It was noted that CPU utilization was within the range 4-8% if the interval is long. But on a short interval it shot up very high occasionally. Some more trial runs revealed the interval to be roughly around 5 minutes. It was also confirmed that Oracle consumes more than 12% of the CPU load even on idle off-peak hours. The process ids that consumed most of the CPU slice were *traced* on Oracle. The SQL statements that were mapped to those PIDs were laid out. One statement was *SELECT SYSDATE from DUAL;* This statement was executed every 5 minutes by the High Availability software that was monitoring Oracle. But CPU usage for this session came nowhere near the ultimate value. Even trace results suggested so.

Also by running different scripts that query Data Dictionary tables (some given by Oracle itself like hSession.sql) the session utilizing maximum of CPU was noted.

DIAGNOSING

There was another statement however. This one was found to take roughly 6%~8% of the CPU slice. On questioning developers it was found to be one statement that was run every 5 minutes (again!) to *time-out* an already logged inactive user. This statement was run separately and traced in the session it was run. Trace files were read. The statement execution time against CPU time was noted. Being run every five minutes, there was the maximum chance for the statement to be found on the buffer. The PCU time taken during **execute** phase of the statement was high. Form the explain plan it was observed that during a Nested Loop join, one of the tables (both similar sized) was doing a full scan.

As it often happens, lack of proper Indexing is often reflected as a huge load on CPU!

RESOLUTION

An index was created on the table. The query started running in 0.35 seconds down from 5.36 seconds. CPU slice was down from 8% to less than 1%.

In less than 15 minutes of creating the index MRTG generated graphs started flattening!