

Tuning the Oracle Server - Identifying Internal Contention

Virag Saksena

Senior Principal Consultant

System Performance Group, Oracle Corporation, USA

Abstract

The readers will learn about contention for different resources in the server - latches, locks and ST enqueues. They will learn to determine if contention for any of these resources is responsible for performance problems. They will learn to drill down to identify the sessions causing the contention and get useful tips for fixing the contention.

Sometimes we might see a machine being utilised at a fraction of its CPU capacity, with lots of free memory, and no appreciable disk activity. Despite absence of any significant contention for disk, CPU or memory, database users might be experiencing extremely poor performance and response times. This may be caused by contention for internal resources within Oracle Server. We will discuss three of these resources

1. Locks - We will discuss how to identify the session holding the locks others might be waiting for.
2. Other enqueues - We will discuss the ST enqueue, its causes and fixes.
3. Latches - What are latches, when are they used, how do they work? What kind of activities result in some of the latches becoming a bottleneck.

Locks and other Enqueues

Unlike certain databases' practice of writers blocking readers, Oracle server uses enqueues to co-ordinate access to shared resources. Enqueues can be locks for data objects (like individual rows or whole tables) or could co-ordinate access to shared structures within the SGA. In this paper, three types of enqueues which are encountered most frequently on single instance systems (TX, TM and ST) will be discussed.

Symptoms

System has adequate CPU, disk and memory resources available. Most of the users are getting quick response times, however some see (or experience) slow response times. In this case there is a fairly high probability that the users experiencing poor response times are waiting for some kind of enqueue or lock.

There are two fixed views (`v$session_wait` and `v$lock`) which can help determine if there is any locking activity

v\$session_wait - This view has one row for each session, describing the event/resource that the session is waiting for. For any session waiting for a lock/enqueue, the event is *enqueue*

```
select sid,event,wait_time
       from v$session_wait
       where event = 'enqueue' ;
```

Fig 1. Query to find sessions waiting for locks/enqueues

This approach can be easily reversed. It is good practice to use this view to determine the bottlenecks on the system [4] If the users experiencing slow response times are waiting for the event *enqueue*, then there are locking issues responsible.

To drill down further, the fixed view **v\$lock** should be queried. This view has information on all the locks/enqueues held and requested. Information about the session holding the lock(s) and the resource(object) that the users are contending for can be obtained.

The column **lmode** describes the mode lock/enqueue is acquired in, so if this column is NULL for a session, that session is waiting for a lock/enqueue. The column **request** describes the mode the session is requesting for the lock. If this column is not null then that session is waiting for a lock/enqueue.

```

select sid
  from v$sqllock
 where lmode is null

select sid
  from v$sqllock
 where request is not null

select sid
  from v$$session_wait
 where event = 'enqueue'

```

Fig 2. More queries to find sessions waiting for locks

By joining the rows on resource identifier **id1**, the sessions waiting for and holding the locks/enqueues can be related.

```

select lpad(' ',decode(nvl(request
,0),0,0,2))||sid session
      ,type, id1, id2
  from v$sqllock
 where id1 in
      (select id1
        from v$sqllock
         where lmode is null)

```

Fig 3. A Tree Structured output of sessions holding and requesting locks/enqueues

The column **type** indicates the type of enqueue being requested/held. If the type is **TM** then the session is requesting/holding a DML lock on database object. The column **id1** points to the **object_id** of the object being locked. When locking the table in an exclusive mode, the DML enqueue is obtained in an exclusive mode (6).

```

break on id1 on sid
select lpad(' ',decode(a.request
,0,0,3))||a.sid ssid,a.id1
      ,a.lmode,a.request,c.name
  from sys.obj$ c,v$sqllock a
 where a.id1 in
      (select id1
        from v$sqllock
         where lmode = 0 )
      and c.obj# = a.id1
      and a.type = 'TM'
 order by a.id1,a.request,
 a.sid,c.name

```

Fig 4. Sessions holding/requesting locks with locked objects

Since the Oracle Server supports row level locking, most sessions will try to lock specific rows in the table, rather than the entire table. This results in a lock of type TX - Transaction enqueue lock. In this case the **id1** does not contain the **object_id** of the object being locked but has information about the rollback segment being locked.

```

select l.sid, r.name
  from v$$rollname r,
       v$sqllock l
 where l.type = 'TX'
       and r.usn = trunc(l.id1/65536)

```

Fig 5. Query to determine rollback segments being used by transactions

How can we identify an object if a row level enqueue is acquired ? Whenever a row level txn lock is acquired, a **TM** lock on the object is also acquired in row-share mode. This is to prevent anyone else from doing DDL on the object. Let's assume that session #1 locks 1000 rows from table A, and 2000 rows from table B. Now session #2 successfully locks 150 rows from table A, and is waiting for a row in table A locked by session #1. Session #1 will have three rows in **v\$sqllock** - two DML locks for table A and B, and a **TX** lock for it's transaction. Session #2 will have three rows in **v\$sqllock** - one DML lock for table A, one **TX** lock for it's transaction and another **TX** lock for the lock it is waiting on. So we can determine the object being locked using the following query

```

break on id1 on sid
select lpad(' ',decode(a.request
,0,0,3))||a.sid ssid,a.id1
      ,a.lmode,a.request,c.name
  from sys.obj$ c,v$sqllock b,v$sqllock a
 where a.id1 in
      (select id1 from v$sqllock
         where lmode = 0 )
      and c.obj# = a.id1
      and b.type = 'TM'
 order by a.id1, a.request, a.sid
      ,c.name

```

Fig 5. Sessions holding/requesting locks with locked objects

The SQL being executed by the session requesting the lock can be determined. This SQL will show the object that the sessions are contending for. A sample query for identifying SQL being executed by sessions waiting for locks is given below. [4]

```

break on hash_value
select hash_value, sql_text
  from v$$sql_text
 where hash_value in
      (select sql_hash_value
        from v$$session
         where sid = &sid)

```

Fig 6. SQL Statement being executed by a given session

Here we can use the session identifier "sid" obtained from any of the queries in Fig 2. By looking at the SQL being executed by the session we can tell which object the session is trying to lock and why.

We can also use the **sid** of the session holding the lock (determined from Fig 3) to determine the SQL it is executing or the resources it is waiting for.

Locking caused by coding practices

There are several application coding practices which cause sessions to cause locking problems

Oracle Server provides sequences for generating unique ids. Most of the other database vendors do not provide this facility. Some application vendors, choose not to use sequences but use control tables to generate unique id's. This decision (motivated by reasons of portability) limits the scalability of such applications. The approach works like this -

There are one or more control tables which have one row for each column in every table which needs a system generated unique identifier. When a new number is required, the corresponding row is updated with the *current value + increment*. Since updating the row requires a row level lock, if two users try to do it simultaneously only one of them will succeed. The new value is then used as the system generated number.

The main problem with this approach is that while doing insert processing it might be necessary to hold the lock to facilitate rollback. Holding the only lock will serialise the process.

Typically inserting a row might consist of several steps

- a) Do some processing
- b) Get the new system generated id
- c) Do some more processing
- d) Repeat a-c for more rows if necessary
- e) Commit the changes

What this means is that the row which is generating the unique identifiers for the entire table is locked throughout the processing of c-d. c-d represent an exclusive state that only one transaction can be in at any given time. If we have a system with 4 CPUs with 200 users inserting records and a processing time of 6 seconds between the time an unique id is generated and the time the changes are saved,

we have limited the throughput to 10 records a minute. If most of the 6 seconds time was not delay time queuing for CPU but actual processing time, the individual transaction is not going to get much quicker. So the system throughput is not going to improve, and the application is not going to scale. A more detailed discussion on queuing analysis may be found in [1]

What are the approaches available to us Reduce the processing in *c-d* as much as possible i.e. Delay obtaining the unique ids as much as possible before the commit, optimise the code running in *c-d*. This requires additional expertise and time when writing the application code.

Do a commit immediately after obtaining the id in step *b* - This could lead the system in an inconsistent stage because of the code already executed in step a. So we can eliminate the code executed in step A by doing unique id generation as Step #1. This will cause a lot of missing numbers because in a lot of cases (exception handling, user rollbacks) the row will not be inserted.

This will also preclude inserting multiple rows and then rolling them back as the processing for second row will commit the first row ! With heavy enough activity this can also cause latch contention (examined later).

Both of the above solutions involve additional complexity in the application code to circumvent the locking issues. The code becomes more stringent and development costs go up. The simplest and most elegant solution is to dispense with the control tables and use sequences, which are provided by the Oracle Server for this purpose.

Ghost sessions holding onto locks

Sometimes clients may terminate abnormally, without notifying their associated shadow server process. The server processes (not realising that they should terminate) continue to hold on to the resources. The client session could be

1. A DB client session running on a desktop computer
2. A telnet session to another host which started the DB client

If another session tries to access the same data, it will find the data locked by a "ghost" session. We need to terminate the "ghost session" in order to release the lock.

For Case 1, with SQL*Net 2.1+, a time-out parameter can be specified, after which it'll ping the client to see if the client is still there. If the client is no longer there then it will terminate the session

In case 2, the host might not even get the signal that the telnet process has been killed, and the client process can still exist on the client host. In this case the SQL*Net time-out parameter will not help.

The Space Transaction (ST) Enqueue

The ST enqueue is used for space transaction management (allocation and de-allocation of extents in the database). This enqueue has time-outs - whenever a session times out, a 1575 - *time-out waiting for space management resource* is posted to the alert logs.

The queries from figure 3 and 7 can be used to determine who is holding the enqueue.

```
select p.program
       from v$process p
            ,v$session s
       where p.addr = s.paddr
            and s.sid = &sid /* The session
holding the ST enqueue from Fig 3 */
```

Fig 7 - Who is holding the ST enqueue

If the ST enqueue is being held by the smon, it is coalescing free space. In tablespaces with high DDL activity, the free extents table might get fragmented, causing the Oracle server system monitor process (smon) to take a longer time when coalescing them into bigger chunks.

For certain tablespaces like the TEMPORARY tablespace it is not even desirable to coalesce free space into continuous chunks. When a sort goes to disk the extent allocated is determined by the default storage specified for the tablespace. Hence all extents allocated will be the same size. When a request is made for a free extent, it requires less work if a free extent of the same size is found. Hence it makes sense to leave the temporary tablespace fragmented. This can be achieved by simply setting the pctincrease to 0. Free space in such a tablespace will not be coalesced automatically by the smon process.

Sizing strategies to avoid segment and tablespace fragmentation are discussed in great detail in [2] and [3].

Most of the time the ST enqueue problems are because of sorts to disk. When a sort grows too big (larger than the init.ora parameter

sort_area_size) it goes to disk. A temporary segment is allocated and the sorting proceeds, allocating more extents as necessary. Upon the completion of the sort the temporary extent is de-allocated.

We can monitor the sorting on system by querying the fixed view *v\$sysstat*

```
select name,value
       from v$sysstat
       where statistic_name in
            ('sorts (memory)'
            , 'sorts (disk)')
```

Fig 8 - Sorting on the system

By using snapshotting where we take the statistics before and after a snapshot we can determine the sorts taking place in a given time interval. We can use the Oracle scripts *bstat*, *estat* for this purpose

The statistic '*DBWR checkpoints*' in *v\$sysstat* can help us compute the average size of a sort to disk. *DBWR checkpoint* (different from the background checkpoint) is done when-ever an extent is de-allocated. The process dropping the extent passes the range of block addresses for the extent to the Oracle Server process database writer (dbwr) and asks it to flush any buffers in that address range from the SGA to the disk.

By assuming that most of the space management activity is occurring due to sorting to disk (a valid assumption on a properly administered production database), the ratio of 'db checkpoints' and sorts(disk) should give us the average number of extents per sort. Multiplying that by the default extent size specified for the TEMPORARY tablespace will then give us the size of average sort.

```
select name, value
       from v$sysstat
       where name in
            ('sort (disk)',
            'DBWR checkpoints')
```

Fig 8 - Determining the average sort size

We can query the fixed view *v\$sesstat* to find the sessions doing most of the sorting to disk.

```
select s.osuser, s.username
       ,s.program, stat.value "Sorts"
       from v$session s, v$sesstat stat
       where stat.statistic# =
            (select statistic#
              from v$statname
              where name = 'sorts (disk)')
            and stat.value > &min_sorts
            and s.sid = stat.sid
```

Fig 9 - Finding sessions causing sorts to disk

After identifying the sessions doing the sorts we can tune their SQL to eliminate the need for sorting. Eliminating most of the sorts to disk is the most effective way to avoid this contention.

If a large fraction of the sorts are going to disk are not very large and there is sufficient memory available on the system, the parameter *sort_area_size* can be increased. However that amount of memory will be locked up by every process which does a large sort. On Unix systems, after a sort is finished the Oracle Memory Manager shrinks the memory allocated for a sort down to *sort_area_retained_size* from the *sort_area_size* memory that had been allocated but does not free up the remaining memory to the operating system, keeping it in the process memory heap for subsequent allocations for other purposes. This is done because on Unix the free call does not release memory back to the OS until the process terminates.

Another process which will cause tables to be created and dropped is the *tkprof* utility for analyzing trace files. If the *explain* option is used to generate execution plans for the statements in the trace file and the *table* option to use an existing plan table is not used then *tkprof* will create and drop a temporary plan table. The *explain* option should never be used without the *table* option.

Latches

Latches protect code paths to shared structures. Enqueues are used to protect structures when operations might take a sufficiently long time that a waiting process is better off sleeping. There are some operations which are so quick that if a process is asked to wait then it might get context switched out, and the context switch might be an order of magnitude greater than the total time the operation on the structure takes. Such code paths are typically protected by *latches*. *Latches* are shared memory structures (typically implemented using fast semaphores) which protect the code paths to the shared structures.

Before a process can perform an operation protected by a latch, it needs to get the latch. There are two modes a latch might be acquired in :

Immediate mode (mostly when the process is already holding another latch) If the process

tries in immediate mode and fails, it returns with a failure and appropriate action is taken.

Willing to Wait mode If the process fails to obtain the latch on the first attempt, it will continue trying until it gets the latch.

For a *willing to wait* attempt the sequence of events might look something like this -

If the initial get fails, and the system has multiple CPUs, the process will try to obtain the latch until (init.ora parameter) *spin_count* attempts have been made. Now one of two things can happen -

1. if the post wait driver is enabled (using init.ora parameter *_latch_wait_posting*) for this latch, the process will go to sleep until posted by the holder of the latch that it is relinquishing the latch
2. if the post wait driver is not enabled, the process will sleep for a specified time. When it wakes up it will once again *spin* for the latch (if on a multi-CPU system). It will continue in this cycle of *spinning* (trying to obtain the latch *spin_count* times) and sleeping, until it obtains the latch. The initial sleep times are 0, 10, 20ms and increase exponentially to a maximum of 2 seconds with suitable randomizations.

Spinning is effective for latches held for really short duration of time, to keep the wait time same order of magnitude as the service time. If a process goes to sleep, then the wait will be much greater than the time latch is held, negatively impacting the throughput. A more detailed discussion on service times, wait times and throughputs can be found in [1]

If the post-wait driver on the system is enabled then certain latches use post wait drivers. So after spinning *spin_count* times the process will sleep until posted by the previous waiter. When the process relinquishes the latch it will post the next waiter.

By querying the fixed view *v\$session_wait* and observing the events most of the sessions are waiting for, it can be determined if the system is experiencing latch contention. If a large number of sessions are waiting for the event '*latch free*' then the system might be experiencing latch contention.

For the event '*latch free*' the column **p2** in *v\$session_wait* specifies the number of latch that the session is waiting on and the column

p3 specifies the number of time this process has slept while waiting for this latch.

```
select sid, p2 "latch#",
       p3 "sleeps"
  from v$session_wait
 where event = 'latch free'
 order by 2,3
```

Fig 10 - Sessions waiting for latches

To obtain the name of the latch we can join to *v\$latchname*

```
select sid, l.name "latch"
       , p3 "sleeps"
  from v$latchname l
       ,v$session_wait w
 where w.event = 'latch free'
       and l.latch# = w.p2
 order by 2,3
```

Fig 10 - Sessions waiting for latches

v\$latch is a fixed view which stores *get/miss* statistics on all the latches in the system.

The column **gets** specifies the number of times a session tried to obtain a latch in *the willing to wait* mode. The whole process of spinning

```
select a.name, a.gets gets
       ,a.misses*100/decode(a.gets,0,1,a.gets) miss
       ,to_char(a.spin_gets*100/decode(a.misses,0,1,a.misses),'990.9')||
       to_char(a.sleep6*100/decode(a.misses,0,1,a.misses),'90.9') cspins
       ,to_char(a.sleep1*100/decode(a.misses,0,1,a.misses),'90.9')||
       to_char(a.sleep7*100/decode(a.misses,0,1,a.misses),'90.9') csleep1
       ,to_char(a.sleep2*100/decode(a.misses,0,1,a.misses),'90.9')||
       to_char(a.sleep8*100/decode(a.misses,0,1,a.misses),'90.9') csleep2
       ,to_char(a.sleep3*100/decode(a.misses,0,1,a.misses),'90.9')||
       to_char(a.sleep9*100/decode(a.misses,0,1,a.misses),'90.9') csleep3
       ,to_char(a.sleep4*100/decode(a.misses,0,1,a.misses),'90.9')||
       to_char(a.sleep10*100/decode(a.misses,0,1,a.misses),'90.9') csleep4
       ,to_char(a.sleep5*100/decode(a.misses,0,1,a.misses),'90.9')||
       to_char(a.sleep11*100/decode(a.misses,0,1,a.misses),'90.9') csleep5
  from v$latch a
 where a.misses <> 0
 order by 2 desc
```

Fig 11 - Query on *v\$latch* to find latch contention

NAME	GETS	MISS	spin s106	s101 s107	s102 s108	s103 s109	s104 s110	s105 s111
cache buffers lru	191,612,306	20.8	91.2	6.7	1.3	0.4	0.2	0.1
library cache pin	170,038,693	2.1	63.0	33.8	2.4	0.5	0.2	0.1
library cache	107,799,453	1.9	49.7	44.6	4.4	0.8	0.3	0.1
			0.1	0.0	0.0	0.0	0.0	0.0

Fig 12 -The output from the query in Fig 11

Here we see that for the *cache buffers lru* latch, 20.8 % of the attempts missed at least once, however the latch was gotten without sleeping even once for 91% of the *misses*, and sleeping once for 6% of the *misses*

Effect of *spin_count* - People tend to increase *spin_count* in an attempt to tune latch

and sleeping until the latch is obtained counts as a single increment of the **gets** columns.

The column **misses** similarly specifies the number of **gets** when the process did not obtain the latch in the first attempt and had to spin for the latch.

The column **spin_gets** specifies the number of misses when the process obtained the latch simply by spinning and did not have to sleep event once.

sleep1-sleep10 represent the misses when the process obtained the latch after sleeping the specified number of times.

sleep11 represents all those misses where the process had to sleep 11 or more times before successfully obtaining the latch.

Rather than looking at the absolute numbers, the relative fractions of the numbers can give us an idea of the latch contention going on.

contention. This is a brute force approach where we are increasing the probability of obtaining the latch by increasing the number of requests for the latch. This is a classic case of fixing the symptom not the cause because it does not address the specific issues which might be causing the latch contention for particular latches

Library Cache, Library Cache Pin, Shared Pool latches

These latches protect operations on the objects in the shared pool (typically shared cursors) library cache. *Shared pool* latch is used to allocate/deallocate space in the shared pool. *Library cache* and *library cache pin* latches are used during different phases of parsing and executing the statements.

Oracle7 introduced the concept of shared cursors. By allowing users to share cursors, the scalability of the system was vastly improved. Some of the major benefits were :

- Savings in memory since the invariant (static) part of the cursor could be shared. With 200 different users executing the same SQL statement, there wouldn't be 200 mostly identical cursors, but 1 shared cursor and 200 private context areas.
- Significantly reductions in the parse time due to the elimination of parses for second and subsequent parses when a user process making a parse call found the cursor in the library cache.

Since these objects are shared, access to them needs to be synchronized. That is the purpose of these latches

Not using bind variables - At most of the Oracle sites experiencing latch contention this is the primary cause. Oracle Server allows the values for the variables to be bound at the execution time rather than parse time. This allows a greater degree of sharing among cursors than by using literal values.

Example Suppose User A is executing :

```
select name
  from customers
 where customer_id = 20046
```

and user B executes :

```
select name
  from customers
 where customer_id = 20063
```

The optimizer sees both the statements as different hence both the statements get parsed as separate cursors, and use almost double the

For packages, procedures and functions the usage is

```
dbms_shared_pool.keep('owner.name','P')
```

Triggers may be pinned with the **R** option

```
dbms_shared_pool.keep('owner.name','R')
```

space that a similar query using bind variables would have been :

```
select name
  from customers
 where customer_id = :cust_id
```

There is increased load on the latches because of increased activity both in parsing and memory management.

Finding sessions doing excessive parsing -

There are several statistics related to parsing and execution in *v\$sesstat* / *v\$sysstat* which can be used to find sessions doing most of the parsing (and using the latches)

parse count - This statistic reflects the parse calls (not necessarily hard parses) made by the session/system.

execute count - Number of executions done by the session/system

parse time cpu - Amount of cpu time (in 10ms ticks) used by the session/system in parsing.

session cursor cache hits - Number of times the cursor was cached in the session and even a parse call was not made.

```
select sid,value
  from v$sesstat
 where statistic# = &stat /*
obtain the statistic# from v$statname */
 and value > &min_value
 order by 2 desc
```

Fig 13 - Resource-intensive sessions

Not pinning big packages/procedures - If bind variables are not used, then with *new* statements the shared pool library cache is going to run out of space eventually. When a new cursor comes in it will deallocate existing cursor(s) to make space. The de-allocated cursor might be needed again and so will have to be parsed again.

In extreme cases we can have one large object coming into the shared pool swapping out large number of small objects.

To alleviate this problem it is advisable to pin the commonly used and large objects. To pin an object use the package *dbms_shared_pool.keep*

Even anonymous PL/SQL blocks and cursors can be pinned using their hash_values and addresses (obtainable from v\$sqlarea)

```
dbms_shared_pool.keep('address.hash_value')
```

It is a good idea to query v\$sqlarea and v\$db_object_cache to find any objects with large executions and memory which are not pinned and pin them at startup time.

```
select name, owner, executions
       , sharable_mem
       from v$db_object_cache
       where kept = 'NO'
          and ( sharable_mem > &min_mem
              or executions > &min_exec)
       order by 3,2
```

```
select hash_value, address
       , executions, sharable_mem
       from v$sqlarea
       where kept_versions > 0
          and ( sharable_mem > &min_mem
              or executions > &min_exec)
       order by 3,2
```

Fig 14 Expensive unpinned objects in the shared pool

Partition the shared pool - With 7.1.6+ the shared pool can be partitioned so that the bigger statements go into a separate part of the pool and don't swap out lots of smaller cursors.

The amount of shared pool reserved for large objects is controlled by (init.ora parameter) *shared_pool_reserved_size*. The default value is 0 and the max is limited to 50% of the *shared_pool_size* (recommended value is 10% of the *shared_pool_size*). Any object above the threshold determined by (init.ora parameter) *shared_pool_reserved_min_alloc* (min value 5000 bytes) goes into the reserved space.

By adjusting these parameter we can ensure that a large object coming in will not cause large number of small objects to be wiped out (If we are pinning the large objects at startup then they won't be coming in anyway).

Ensure modules are valid - If program unit in the data dictionary gets invalidated over a period of time then it will be recompiled at the execution time. This will cause more stress on the latches. By finding these objects and compiling them, we can avoid this load at peak time when an user might access the object.

```
select owner, object_type
       , object_name
       from dba_objects
       where status = 'INVALID'
```

Fig 15 - Invalid objects

Cache buffers chains and cache buffers lru chain latches

Like the previous latches protect the access paths to the shared cursors in the library cache, these latches protect the access paths to the db block buffers in the buffer cache.

Till 7.3 we had a single LRU latch and multiple cache buffer chains latches.

Every database block in the SGA is on two of the three lists

1. either on LRU list or on dirty list
2. cache buffer chain

When a processes needs a buffer it computes the hash value for the buffer's address and looks in the appropriate hash bucket (cache buffer chain). It might not find the buffer there or might find a buffer which has been modified by another process. So either it has to read the buffer from the disk or create a read consistent clone. For any of the two purposes it needs a free buffer.

To locate the free buffer it scans the LRU list from the least recently used end. The buffers it encounters can be in three states -

1. **pinned** being used by some other process, it skips past it and continues searching
2. **dirty** buffer has been modified and needs to be written out to disk it moves the process to the dirty list and continues searching
3. **free** buffer can be used, it moves it to the most recently used end, removes it from the current hash bucket, reads (constructs) the buffer into it and places it on the appropriate hash bucket.

The contention for LRU latch is caused mostly by poorly optimized SQL running in a sufficiently large SGA. If there are two queries accessing large number of buffers, most of which are cached, this latch will be the main bottle-neck. The only solution in this case is to find the SQL being executed by these sessions and tune the SQL statements.

With 7.3 we have multiple LRU latches. The number of suggested latches can be specified by (init.ora parameter) *db_block_lru_latches*

There are multiple Cache buffer chains latches (specified by the init.ora parameter

_db_block_hash_buckets). On this latch we get contention when we get a lot of clones for the same block or lots of processes try to access the same block.

An application using control tables to generate unique ids and doing commits immediately after each update will cause clones to be generated by different processes accessing that block.

To determine if some buffers have large number of clones we can query the buffer headers in the SGA using the fixed table *x\$bh* (valid for Releases 7.1.6-7.3.2)

```
select dbafil "File"
      ,dbablk "Block"
      ,count(*)
  from x$bh
 group by dbafil,dbablk
 having count(*) > 2
```

Fig 16. Buffers with excessive clones

We can then determine what segment the buffer belongs to

```
select segment_name,owner
  from dba_extents
 where file_id = &file_id
       and &block_id between block_id
       and block_id + blocks - 1
```

Fig 17. Relating the block to a segment

If the tables are accessing the same row (as determined from the SQL being executed by the sessions) then the application logic needs to be examined.

If different sessions are accessing different rows which are packed in same blocks then by playing with the *pctincrease* and *pctfree* storage parameter to simulate a smaller block size and spread out the data we can ease the bottleneck.

Redo allocations and copy latches

These two latches are used by the user processes to write redo to the log buffer.

redo allocation latch (1 latch per instance) is used to allocate space in the buffer for the redo.

redo copy latches (number of latches equal to *init.ora* parameter *log_simultaneous_copies* on a multiple CPU system) are used to copy the redo into the buffer.

A process first obtains the *redo allocation* latch. After allocating the space, if the redo size is bigger than (the *init.ora* parameter) *log_small_entry_max_size*, it tries to obtain

the *redo copy* latch. The attempt for the second latch is made in an immediate mode because the process is holding the only *redo allocation* latch on the system.

If the redo entry size is smaller than *log_small_entry_max_size* or if the *redo copy* latch immediate get fails, then the redo is copied while holding the *redo allocation* latch. For smaller entries this is done because the time to obtain the second latch might be comparable to the time to copy the redo.

Otherwise the process having obtained the *redo copy* latch, relinquishes the *redo allocation* latch and proceeds with copying the redo.

If there is contention for redo allocation or redo copy latches, then both the parameters *log_small_entry_max_size* and *log_simultaneous_copies* can be increased.

The redo might be scattered in memory in different locations and it might require several operations to copy it to the allocated space. If we coalesce all the redo entries in one space then it will require only one operation to copy the redo and the latch will be held for lesser time. By setting the *init.ora* parameter *_log_entry_prebuild_threshold* to a suitable value, all redo operations whose size is smaller than the parameter's value will have their entries coalesced before obtaining any latches.

Bibliography

1. Jain, Raj. *The Art of Computer Systems Performance Analysis*. Wiley, 1991
2. Millsap, Cary V. *The OFA Standard, Oracle7 for Open Systems*. Oracle Part No. A19308-1, May 1994
3. Shallahamer, Craig A. *Avoiding A Database Reorganization*. Proceedings of the 1994 International Oracle User Week (IOUW) Conference, Paper #142. *Oracle Magazine*, Summer 1994.
4. Saksena, Virag. *Identifying resource intensive SQL in a production environment*. Proceedings of the EOUG Conference, April 1996. Oracle Open World Asia Pacific, Nov 1996.
5. *Oracle7 Server Administrator's Guide*
6. *Oracle7 Server Tuning Guide*
7. *Oracle7 Server Concepts Manual*

Virag Saksena is a Senior Principal Consultant with Oracle Services' System Performance Group. The team is responsible for building new tools and capabilities like the ones described in this paper for Oracle and its customers. The System Performance Group provides technical architecture services including capacity planning, and performance management services to customers world-wide. His specialisation is performance management and application tuning. Virag is based out of Redwood Shores, California and can be reached at 415.506.5087 or on the internet at vsaksena@us.oracle.com. This paper is available via the World-Wide Web at <http://www.oracle.com/consulting> <http://www.europa.com/~orapub> and on guts.us.oracle.com.

Acknowledgements

I would like to acknowledge the valuable advice and support I have received from my managers, peers and colleagues at Oracle, without which this paper would have never been in this form - Cary Millsap, Craig Shallahamer, Dominic Delmolino, Graham Wood, Harald Eri, Ferran Brichs and Chris D'Allesandro.