

Tuning latch contention: Cache buffers chain latches

Recently, I had an opportunity to tune latch contention for cache buffers chain (CBC) latches. Problem statement is that high CPU usage combined with poor application performance. Quick review of statspack report of 15 minutes showed a latch free wait as top event and consuming 3600 seconds approximately, in a 8 CPU server. Further CPU usage was quite high, which is a typical symptom of latch contention, due to spinning involved. v\$session_wait showed that hundreds of sessions were waiting for latch free event.

```
SQL> @waits10g
```

SID	PID	EVENT	P1_P2_P3_TEXT
294	17189	latch free	address 15873156640-number 127-tries 0
628	17187	latch free	address 15873156640-number 127-tries 0
...			
343	17191	latch free	address 15873156640-number 127-tries 0
599	17199	latch: cache buffers chains	address 17748373096-number 122-tries 0
337	17214	latch: cache buffers chains	address 17748373096-number 122-tries 0
...			
695	17228	latch: cache buffers chains	address 17748373096-number 122-tries 0
...			
276	15153	latch: cache buffers chains	address 19878655176-number 122-tries 1

We will use two pronged approach to find root cause scientifically. First, we will find SQL suffering from latch contention and objects associated with access plan for that SQL. Next, we will find buffers involved in latch contention, map that back to objects. Finally, we will match these two techniques to pinpoint root cause.

Before we go any further, let's do a quick summary of internals of latch operations.

Brief Introduction to CBC latches and not-so-brief reason why this is a complicated topic to discuss briefly

Latches are internal memory structures to coordinate access to shared resources. Locks aka enqueues are different from latches. Key difference is that enqueues, as name suggests, provides a FIFO queueing mechanisms and latches do not provide a queueing mechanism. On the other hand, latches are held very briefly and locks are usually held longer.

In Oracle SGA, buffer cache is the memory area data blocks are read in to, aka buffer cache. [If ASMM - Automatic Shared Memory Management is in use, then part of Shared pool can be tagged as KGH:NO ALLOC and remapped to buffer cache area too].

Each buffer in the buffer cache has an associated element the buffer header array, externalized as x\$bh. Buffer headers keeps track of various attributes and state of buffers in the buffer cache. This Buffer header array is allocated in shared pool. These buffer headers are chained together in a doubly linked list and linked to a hash bucket. There are many hash buckets (# of buckets are derived and governed by _db_block_hash_buckets parameter). Access (both inspect and change) to these hash chains are protected by cache buffers chains latches.

Further, buffer headers can be linked and delinked from hash buckets dynamically.

Simple algorithm to access a buffer is:(I had to deliberately cut out so as not to deviate too much from our primary discussion.)

1. Hash data block address (DBA: Combination of tablespace, file_id and block_id) to find hash bucket.
2. Get latch protecting hash bucket .
3. If (success) then Walk the hash chain reading buffer headers to see if a specific version of the block is already in the chain.
If found, access the buffer in buffer cache, with protection of buffer pin/unpin actions.
If not found, then find a free buffer in buffer cache, unlink the buffer header for that buffer from its current chain, link that buffer header with this hash chain, release the latch and read block in to that free buffer in buffer cache with buffer header pinned.
4. If (not success) spin for spin_count times and go to step 2.
If not gotten with spinning, then sleep, with increasing exponential back-off sleep time.

Obviously, latches are playing crucial role controlling access to critical resources such as hash chain. My point is that repeated access to few buffers can increase latch activity.

There are many CBC latch children (derived by size of buffer cache). Parameter `_db_block_hash_latches` control # of latches and derived based upon buffer cache size. Further, In Oracle 10g, sharable latches are used and inspecting an hash chain needs to acquire latches in share mode, which is compatible with other shared mode operations. Note that these undocumented parameters are usually sufficient and changes to these parameters must get approval from Oracle support.

Back to our problem...

Let's revisit our problem at hand. Wait graph printed above shows that this latch contention is caused by two types of latches. Latch # 127 is simulator lru latch and #122 is cache buffers chains latch.

```
select latch#, name from v$latch where latch# in (127, 122);
```

Problem with 'simulator lru' latch is simple. There is a bug with `db_cache_advice` and bug number is 5918642. If `db_cache_advice` is set to ON, then latch contention due to simulator lru latches can be observed for large buffer caches. This issue was fixed quickly by `db_cache_advice` to OFF.

After resolving 'simulator lru' latch, we had some relief in performance but not much.

Querying `v$session` to see what SQL(s) causing Latch contention. State column below indicates that processes are not currently waiting for latches, but waited in the past. 24 sessions are executing same SQL statement and last wait in the past is 'latch free' event for these sessions and yes, these are active sessions. If the latch contention is prevalent, then querying `v$session` as below, will provide SQLs to focus on.

```
select event, sql_hash_value,state, count(*) from v$session w
where event='latch free' and status='ACTIVE'
group by sql_hash_value, state , event
/
```

EVENT	SQL_HASH_VALUE	STATE	COUNT(*)
latch free	3629331128	WAITED KNOWN TIME	24

```

latch free          673277007 WAITED KNOWN TIME          1
latch free          1378683334 WAITED SHORT TIME          1
latch free          3629331128 WAITED SHORT TIME          5
latch free          2920275581 WAITED SHORT TIME          3

```

We can find SQL statement querying v\$sql_text with above hash value 3629331128 and SQL suffering from latch contention is printed below. Of course, care has been taken to change actual object names for security reasons.

```

select * from v1 WHERE (
coll IN (
  3, 20, 21, 44, 45, 47, 48, 49, 50, 51, 57, 58, 59, 67, 68,
  69, 76, 78, 79, 80, 81, 82, 84,85, 106, 450, 451, 452, 453,
  454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465,
  466, 467, 468, 469, 470, 471, 472, 473, 474, 476, 478, 500,
  501, 502)
OR col2 IN (3, 20, 21, 44, 45, 47, 48, 49, 50, 51, 57, 58,
59, 67, 68, 69, 76, 78, 79, 80, 81, 82, 84, 85, 106, 450, 451,
452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463,
464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 476,
478, 500, 501, 502))
AND UPPER(col3) LIKE :1 and rownum> 200

```

Explain plan for the above shows that multiple tables are accessed in this view. But, at this point, we don't know which step in this plan is causing latch contention. If you have to guess, which of the following tables, is causing the issue? Check your guess with correct answer later (and become a BAAG member immediately).

[Few columns removed from the plan output to improve readability]

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		17778	223
* 1	COUNT STOPKEY			
2	CONCATENATION			
* 3	FILTER			
* 4	HASH JOIN		17777	216
* 5	TABLE ACCESS FULL	ORG	4	2
6	NESTED LOOPS		197	213
7	NESTED LOOPS		195	18
8	TABLE ACCESS FULL	ORG	1	2
* 9	TABLE ACCESS FULL	ORDER	195	16
* 10	TABLE ACCESS BY INDEX ROWID	TRADE	1	1
* 11	INDEX UNIQUE SCAN	TRADE_PK	1	0
* 12	FILTER			
13	NESTED LOOPS		1	7
14	NESTED LOOPS		1	6
15	NESTED LOOPS		1	4
* 16	TABLE ACCESS FULL	ORG	1	2
* 17	TABLE ACCESS FULL	ORG	1	2
* 18	TABLE ACCESS FULL	ORDER	3	2
* 19	TABLE ACCESS BY INDEX ROWID	TRADE	1	1
* 20	INDEX UNIQUE SCAN	TRADE_PK	1	0

Researching further..

Re-querying v\$session_wait, we see that couple of latches are hot. We will consider one latch children with latch address 19875043200 as an example and drill down further.

SID	PID	EVENT	P1_P2_P3_TEXT
578	17220	latch:CBC	address 19875043200-number 122-tries 0
664	17226	latch:CBC	address 19875043200-number 122-tries 0
695	17228	latch:CBC	address 19875043200-number 122-tries 0
701	23987	latch:CBC	address 19875043200-number 122-tries 0

Converting this latch address 19875043200 from decimal to hex yields 4A0A51780. But latch address is 16 bytes and so prefixing with zeros and querying v\$latch_children to see activity against that latch children.

```
select addr, latch#, child#, level#, gets
from v$latch_children where addr='00000004A0A51780'
SQL>/
ADDR                LATCH#    CHILD#    LEVEL#    GETS
-----
00000004A0A51780    122       10437      1       23672075

SQL> /
ADDR                LATCH#    CHILD#    LEVEL#    GETS
-----
00000004A0A51780    122       10437      1       23672209
```

Repeated the execution of above SQL almost immediately. An increase 134 gets in sub-seconds. Above step also helps to validate latch address and comparing with latch type we see that this latch address is indeed Cache buffers chains latch.

Hang those buffers!

Next, we need to find buffers protected by this latch children and then find the buffers causing latch contention. Many such hash buckets (and so, numerous buffers) are protected by a latch children. Fortunately, column tch can be used effectively to identify hot block(s). Almost every access to a buffer increments tch value for that buffer header. Idea here is to find buffers protected by that latch and identify buffers with higher touch counts. Those buffers are probable candidates for further analysis.

x\$bh table and v\$latch_children can be joined to find those buffer attributes. (BTW, Following SQL can be rewritten with ease to print buffers protected by top latch, say by sleeps.)

```
select hladdr, file#, dbablk, decode(state,1,'cur ',3,'CR',state) ST, tch
from x$bh where hladdr in
(select addr from (select addr from v$latch_children where addr='00000004A0A51780'
order by sleeps, misses,immediate_misses desc )where rownum <=2)
```

HLADDR	FILE#	DBABLK	ST	TCH
00000004A0A51780	1	52351	cur	3
00000004A0A51780	16	701009	cur	24
00000004A0A51780	16	23959	cur	182
00000004A0A51780	15	16215	cur	2855 <--
00000004A0A51780	26	693	cur	9
00000004A0A51780	9	52872	cur	2935 <--
00000004A0A51780	8	45128	cur	1831 <--
00000004A0A51780	16	635473	cur	560
00000004A0A51780	25	233403	cur	51
00000004A0A51780	25	97993	cur	110
00000004A0A51780	4	97273	cur	43
00000004A0A51780	25	268340	cur	0

12 rows selected.

There are at least three blocks with higher activity since their tch value is much higher. A note of caution, buffer cache activity is quite dynamic and this analysis need to be performed during the period

of latch contention. Performing this analysis few hours after latch contention will lead to incorrect diagnosis.

We need to associate these three hot blocks with object names. File# and DBAblk can be used to find object using following script.

```
accept h_file_id prompt ' Enter file_id ==&gt;';
accept h_block_id prompt ' Enter block_id==&gt;';
set verify off
column owner format A10
column segment_name format A20
column segment_type format A10
column hdrfile format 9999
column curfile format 9999
column curblk format 99999999
column hdrblock format 99999999
select owner, segment_name,partition_name, segment_type, file_id,block_id from dba_extents
where file_id = &&h_file_id and
       block_id &&h_block_id;
set verify on
```

For example, supply 15 for the file_id and 16215 for block_id to find the object_name for buffer with tch of 2855.

I don't prefer above script, since performance is not so optimal. It is much easier to dump the blocks and convert them to object ids. Let's dump these three blocks.

```
alter system dump datafile 15 block min 16215 block max 16215;
alter system dump datafile 9 block min 52872 block max 52872;
alter system dump datafile 8 block min 45128 block max 45128;
```

Reading trace files, we see three different segments and one line per block is printed below from the trace file.

```
seg/obj: 0xd756 csc: 0x00.17b5fe4f itc: 2 flg: E typ: 1 - DATA
seg/obj: 0x1801a csc: 0x00.1cb9f0ab itc: 2 flg: E typ: 1 - DATA
seg/obj: 0x181c5 csc: 0x00.1bef7a59 itc: 169 flg: E typ: 2 - INDEX
```

seg/obj field is the object_id printed in hex. Converting these hex numbers d756, 1801a and 181c5 to decimal equivalents results in 55126, 98330,98757.

Now,we could query, dba_objects to find object_names.

```
select owner, object_id, object_name, data_object_id from dba_objects
where object_id in (55126, 98330,98757) or
       data_object_id in (55126, 98330,98757)
SQL>/
```

OWNER	OBJECT_ID	OBJECT_NAME	DATA_OBJECT_ID
SOME_USER	55126	ORDER	55126
SOME_USER	98330	GBLOCK	98330
SOME_USER	98757	ALLOCATION_OID	98757

Comparing explain plans and object_names printed above, we can see that ORDER table is a common object between these two techniques.

7	NESTED LOOPS		195	14040	18
8	TABLE ACCESS FULL	ORG	1	34	2
* 9	TABLE ACCESS FULL	ORDER	195	7410	16

14	NESTED LOOPS		1	106	6
15	NESTED LOOPS		1	68	4
* 16	TABLE ACCESS FULL	ORG	1	34	2
* 17	TABLE ACCESS FULL	ORG	1	34	2
* 18	TABLE ACCESS FULL	ORDER	3	114	2

```
9 - filter(UPPER(UPPER("GO"."ORD_ID"))) LIKE :1 AND "GO"."IS_MOD"='F')
18 - filter(UPPER(UPPER("GO"."ORD_ID"))) LIKE :1 AND "GO"."IS_MOD"='F')
```

Quick summary

Let's summarize what we have done so far.

1. We found one latch children address, located all buffers protected by that latch, found buffers with high tch, queried to find object names for those buffers
2. Through v\$session_wait we found sql hash value, found sql suffering from latch contention, generated explain plan.

From these two different techniques, we can find objects common to both steps 1 and 2 and those objects are probable candidates to focus on. We see that ORDER table is common to both techniques. From the plan above, ORDER table is accessed in a tight nested loops join. This will increase buffer access to ORDER table, in turn, resulting in higher latching activity.

SQL tuning: That was easy

From here onwards, solution is straightforward, we need to avoid tight nested loops join. Specifically, if inner tables in the nested loops join is accessed with Full Table Scan access then that can cause increased latching activity. Hash join might be a preferred access method. For every row from outer row source, inner row source is queried, in a nested loops join. But, in hash join, tables are scanned once and hashed reducing latching activity.

In this specific case, ORDER is a small table. Further analysis revealed that, CBO chose nested loops join since rownum triggers first_rows optimizer_mode. As a test, let's remove rownum clause to see what plan we get.

```
explain plan for
select * from v1 WHERE (
  col1 IN (
    3, 20, 21, 44, 45, 47, 48, 49, 50, 51, 57, 58, 59, 67,
    68, 69, 76, 78, 79, 80, 81, 82, 84, 85, 106, 450, 451,
    452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462,
    463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473,
    474, 476, 478, 500, 501, 502) OR
  col2 IN (3, 20, 21, 44, 45, 47, 48, 49, 50, 51, 57, 58,
    59, 67, 68, 69, 76, 78, 79, 80, 81, 82, 84, 85, 106, 450,
    451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463,
    464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 476, 478, 500, 501, 502))
AND UPPER(col3) LIKE :1 --and rownum
```

```
select * from table(dbms_xplan.display);
```

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT		4902	818K	4014
* 1	HASH JOIN		4902	818K	4014
2	TABLE ACCESS FULL	ORG	370	12950	5

* 3	HASH JOIN		17513	2325K	4008
4	TABLE ACCESS FULL	ORG	370	12950	5
* 5	HASH JOIN		17479	1724K	4002
* 6	TABLE ACCESS FULL	ORDER	17331	643K	1320
7	TABLE ACCESS FULL	TRADE	353K	21M	2680

After commenting rownum clause, CBO chose hash join to join that table. Comparing execution times between these two versions, running two of these SQLs in parallel, original query consumed almost 70 CPU seconds per execution and query with commented rownum clause consumed a cpu time of just 0.5 seconds. Essentially, optimizer_mode of all_rows should be used for this SQL, even if rownum predicate is used. Fortunately, this query is accessing a view (and only similar queries are accessing that view) and so adding all_rows hint to that view resolved latch contention.

We used two pronged approach: Find objects causing latch contention and match those objects with execution plan of SQL suffering from latch contention. Then, resolved the issue with a minor change to the view.