

Dynamic plan table, x\$skqlfxpl and extreme library cache latch contention

We had an interesting latch contention issue with a client database worth sharing. Client complained that they are seeing increased library cache latch contention in production with no logical explanation (yet). So, client wanted us to find the root cause. This performance issue seems to have started recently and occurs intermittently and also resolves by itself. Client is planning to upgrade to 10g in 4 weeks time frame and this issue need to be resolved to avoid any delays in 10g upgrade.

From statspack analysis and various statistics analysis, it is visible that there is an increase in library cache latch contention, but nothing jumps up as a problem. Of course, this is a very active application [5 node RAC application] with very high number of literal SQLs but that is nothing new, it is always been this way. No new dynamic SQLs, no DDLs invalidating cursors or anything of that sort. I started watching production instance, in real time, waiting for the re-occurrence of this issue. It occurred!

latchprofx for rescue

At this very juncture, instance was suffering from latch contention and It was visible that sessions were starting to queue up for latches.

I was ready with necessary tools. Tanel Poder has written a great tool latchprofx to sample v\$latch_holder with high frequency [Actually, it uses x\$ tables but you get the idea]. In this case, high frequency sampling is a must since this performance problem disappears in few minutes or so. Executed latcprofx script for library cache latches and 10000 samples as printed below.

```
SQL> @/tmp/latchprofx sid,name % library 10000
-- LatchProfx 1.06 by Tanel Poder ( http://www.tanelpoder.com )

  SID NAME                                Held  Held %    Held ms
-----
  140 library cache                       9555   95.55   25167.870
  185 library cache                        198    1.98    521.532
  245 library cache                         97     .97    255.498
  185 library cache pin                     69     .69    181.746
  245 library cache pin                      39     .39    102.726
  240 library cache                         33     .33    86.922
  240 library cache pin                      6     .06    15.804
  227 library cache                          2     .02     5.268
  227 library cache pin                      1     .01     2.634
```

Yikes. We can see that one session holding library cache latch in 95.5% of samples. That seems bit odd.

Of course, there are many latch children for library cache latches. Is that session repeatedly acquiring just one latch children? Or is it acquiring many different latch children? More importantly what is that session executing ?

Latchprofx to the rescue again. Latchprofx can also print latch children address. Re-executing latcprofx script with modified second argument to the call. Only difference between these two calls is that first argument to latchprofx includes laddr now.

```
@/tmp/latchprofx sid,name,laddr % library 10000
```

SID	NAME	LADDR	Held	Held %	Held ms
140	library cache	0000000990FEFA18	298	2.98	783.442
140	library cache	0000000990FED2B8	270	2.70	709.830
140	library cache	0000000990FEF298	254	2.54	667.766
140	library cache	0000000990FEF658	254	2.54	667.766
140	library cache	0000000990FEECF8	240	2.40	630.960
140	library cache	0000000990FF0198	236	2.36	620.444
140	library cache	0000000990FEE488	225	2.25	591.525
140	library cache	0000000990FEFCE8	223	2.23	586.267
140	library cache	0000000990FEEED8	222	2.22	583.638
140	library cache	0000000990FEF388	222	2.22	583.638
140	library cache	0000000990FEDEE8	220	2.20	578.380
140	library cache	0000000990FF00A8	216	2.16	567.864
140	library cache	0000000990FEE758	215	2.15	565.235
140	library cache	0000000990FEEFC8	212	2.12	557.348
140	library cache	0000000990FEE938	211	2.11	554.719
140	library cache	0000000990FEE668	205	2.05	538.945
140	library cache	0000000990FEDB28	201	2.01	528.429

.....

Whoa! This session was acquiring nearly all latch children. Notice that latch addresses are different in every row output. Meaning, various samples of latchprofx saw that this session is holding a different latch children. In a nutshell, this session was acquiring and releasing ALL library cache latches (aka children) in quasi sequential fashion and causing massive performance issue. There are 53 latch children for library cache latch (in this database). Column held% is approximately 2%, indicating all library cache latch children were acquired and released in a quasi-uniform distributed(2% times 50), quasi-cyclical fashion! Same pattern repeats if I reduce or increase samples.

Latch greedy session

Initially, I thought, this is a bug. I have not seen a case where all library cache latch children were needed to execute a SQL or a process causing massive latch contention. [Okay, I confess, I have simulated that kind of issue in my test database holding library cache latch children using oradebug kslgetl stuff, but that is a different topic altogether]. Session 140 was executing a SQL printed below. This SQL is querying execution plan from shared_pool for a specific hash_value in this 9i database(Equivalent to dbms_xplan.display_cursor stuff in 10g.)

```
select plan_table_output from
TABLE( dbms_xplan.display (
      'backup.dynamic_plan_table',
      (select rawtohex(address)||'_'||child_number x from
        v$sql where hash_value=&hv ), 'serial' ) )
/
```

Turned on sqltrace on that session and tkprof showed that recursive SQL generated from dbms_xplan package as a costly SQL.

```

REM Removing few lines to improve readability.
SELECT /* EXEC_FROM_DBMS_XPLAN */ id, position, level , operation,
options, object_name , cardinality, bytes, temp_space,
cost, io_cost, cpu_cost ,decode(partition_start, 'ROW
LOCATION',
....
        from BACKUP.DYNAMIC_PLAN_TABLE start with id = 0
        and timestamp >=
        (select max(timestamp) from BACKUP.DYNAMIC_PLAN_TABLE
where id=0 and
        statement_id = '00000009BB9B6D50_0' and
nvl(statement_id, ' ') not like 'SYS_LE%')
        and nvl(statement_id, ' ') not like 'SYS_LE%' and
statement_id = '00000009BB9B6D50_0' connect by (prior id = parent_id
        and prior nvl(statement_id, ' ') =
        nvl(statement_id, ' ')
        and prior timestamp <= timestamp)
        or (prior nvl(object_name, ' ') like 'SYS_LE%'
        and prior nvl(object_name, ' ') =
        nvl(statement_id, ' ')
        and id = 0 and prior timestamp <= timestamp)
        order siblings by id
/

```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.03	0.02	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	30	53.79	55.90	423	375837	0	29
total	32	53.82	55.93	423	375837	0	29

X\$KQLFXPL

55 seconds to execute this SQL? That seems bit excessive. This SQL is the only time-consuming recursive SQL generated from the select statement. So, somehow, this must be causing latch contention. I will print only relevant lines from the plan output.

```

....
| 4 | NESTED LOOPS OUTER | | 1 | 4479 | 21
(5)|
| 5 | NESTED LOOPS OUTER | | 1 | 4475 | 20
(5)|
|* 6 | FIXED TABLE FULL | X$KQLFXPL | 1 | 4444 |
|
| 7 | TABLE ACCESS BY INDEX ROWID| OBJ$ | 1 | 31 | 3
(34)|
|* 8 | INDEX UNIQUE SCAN | I_OBJ1 | 1 | | 2
(50)|
| 9 | TABLE ACCESS CLUSTER | USER$ | 1 | 4 | 2
(50)|
|* 10 | INDEX UNIQUE SCAN | I_USER# | 1 | |
|
....

```

```
6 - filter("P"."INST_ID"=:B1 AND "P"."KQLFXPL_HADD""P"."KQLFXPL_PHAD")
```

Line #6 is the problem. This table x\$skqlfxpl is accessed using Full table scan. Looks like, x\$skqlfxpl can NOT be accessed without holding library cache latches. Since there is a FTS on this fixed table, almost every library cache object need to be touched which potentially means all library cache objects must be inspected. Further, accessing this x\$

table means library cache buckets (`_kgl_bucket_count` controlled area) need to be walked, latches acquired and released before that memory area is inspected. Makes sense!

Repeated calls to `kqfexp`, `kglic`, `kglic0` also visible in the stack output. These calls suggests that library cache latches were acquired and released. Systemstate dumps confirms that too.

I was also able to reproduce this issue querying `x$kqfexpl` in a cloned database.

```
SQL> select /*+ full(a) */ count(*) from x$kqfexpl a;
```

```
-----
COUNT(*)
-----
48926
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)
0	SELECT STATEMENT		1		18 (6)
1	SORT AGGREGATE		1		
2	FIXED TABLE FULL	X\$KQLFXPL	100		

Running `latchprof` script from another session concurrently shows that accessing this `x$` table will result in excessive library cache latching activity. Row estimates column for this table shows 100 which is a CBO default cardinality estimate if there are no statistics on a table.

```
@/tmp/latchprof sid,name,laddr % library 10000
```

SID	NAME	LADDR	Held	Held %	Held ms
127	library cache	0000000990FED2B8	26	.26	68.120
127	library cache	0000000990FEFEC8	24	.24	62.880
127	library cache	0000000990FEF658	23	.23	60.260
127	library cache	0000000990FEE668	22	.22	57.640
127	library cache	0000000990FEFA18	22	.22	57.640
...					

Background

Now, a little bit background is necessary. What started all this? As a part of 10g upgrade planning, Client DBAs had setup a job to capture costly SQLs and store execution plans from `shared_pool` using `dynamic_plan_table` script, borrowed from Tom kyte [dynamic plan table discussion](#). Excellent idea and wonderfully written sql, Tom.

If you don't know already, Tom's method creates following view. Client script passes the `hash_value` of costly SQLs from `v$sql` to this view.

```
create or replace view dynamic_plan_table
as
select
  rawtohex(address) || '_' || child_number statement_id,
  sysdate timestamp, operation, options, object_node,
  object_owner, object_name, 0 object_instance,
```

```

optimizer, search_columns, id, parent_id, position,
cost, cardinality, bytes, other_tag, partition_start,
partition_stop, partition_id, other, distribution,
cpu_cost, io_cost, temp_space, access_predicates,
filter_predicates
from v$sql_plan
;

REM there is a wrapper script calling following SQL, in a loop, for top
20 SQL statements at a stretch.
REM Of course, there is more logic in that script.

```

```

select plan_table_output from TABLE( dbms_xplan.display (
'backup.dynamic_plan_table',
      (select rawtohex(address)||'_'||child_number x from
v$sql where hash_value=&hv ), 'serial' ) )
/

```

Unfortunately, such an awesome method has side effects in version 9.2.0.8, as explained above.

At this point, I am almost sure that we need to collect statistics on fixed tables to resolve this issue. There is a fixed index on x\$sqlfxpl (KQLFXPL_HADD, KQLFXPL_PHAD, KQLFXPL_HASH) and if that index is used, effect of this problem should disappear. Since there is a plan to upgrade to 10g soon, client didn't want to collect fixed statistics.

So, I worked around the issue. Not an elegant solution, but it avoids problem by recreating the view with hardcoded hash value for every loop.

```

create or replace view backup.dynamic_plan_table
as
select
rawtohex(address) || '_' || child_number statement_id,
sysdate timestamp, operation, options, object_node,
object_owner, object_name, 0 object_instance,
optimizer, search_columns, id, parent_id, position,
cost, cardinality, bytes, other_tag, partition_start,
partition_stop, partition_id, other, distribution,
cpu_cost, io_cost, temp_space, access_predicates,
filter_predicates
from v$sql_plan
where hash_value=&&hv      -- Added this line..
;

select plan_table_output from TABLE( dbms_xplan.display (
'backup.dynamic_plan_table',
      (select rawtohex(address)||'_'||child_number x from
v$sql where hash_value=&&hv ), 'serial' ) )
/

```

Now, it is evident that query runs in 0.03 seconds and also used fixed index.

```
51      NESTED LOOPS OUTER (cr=261 r=0 w=0 time=1327 us)
51      NESTED LOOPS OUTER (cr=131 r=0 w=0 time=892 us)
51      FIXED TABLE FIXED INDEX X$KQLFXPL (ind:3) (cr=0 r=0 w=0 time=496 us)
27      TABLE ACCESS BY INDEX ROWID OBJ$ (cr=131 r=0 w=0 time=327 us)
27      INDEX UNIQUE SCAN I_OBJ1 (cr=104 r=0 w=0 time=212 us)(object id 33)
27      TABLE ACCESS CLUSTER USER$ (cr=130 r=0 w=0 time=359 us)
27      INDEX UNIQUE SCAN I_USER# (cr=28 r=0 w=0 time=78 us)(object id 11)
```

Finally

10g upgrade guide recommends collecting fixed table statistics. So, this issue should completely disappear in 10g.