

gc cr disk read

Posted by [Riyaj Shamsudeen](#) on January 12, 2012

You might encounter RAC wait event 'gc cr disk read' in 11.2 while tuning your applications in RAC environment. Let's probe this wait event to understand why a session would wait for this wait event.

Understanding the wait event

Let's say that a foreground process running in node 1, is trying to access a block using a SELECT statement and that block is not in the local cache. To maintain the read consistency, foreground process will request the block with a query SCN. Then the sequence of operation is(simplified):

1. Foreground session calculates the master node of the block; Requests a LMS process running in the master node for the block.
2. Let's assume that block is resident in the master node's buffer cache. If the block is in a consistent state (meaning block version SCN is lower (or equal?) to query SCN), then LMS process can send the block to the foreground process immediately. Life is not that simple, so, let's assume that requested block has an uncommitted transaction.
3. Since the block has uncommitted changes, LMS process can not send the block immediately. LMS process must create a CR (Consistent Read) version of the block: clones the buffer, applies undo records to the cloned buffer rolling back the block to the SCN consistent with the requested query SCN.
4. Then the CR block is sent to the foreground process.

LMS is a light weight process

Global cache operations must complete quickly, in the order of milli-seconds, to maintain the overall performance of RAC database. LMS is a critical process and does not do heavy lifting tasks such as disk I/O etc. If LMS process has to initiate I/O, instead of initiating I/O, LMS will downgrade the block mode and send the block to the requesting foreground process (this is known as Light Works rule). Foreground process will apply undo records to the block to construct CR version of the block.

Foreground process might not find the undo blocks in the local cache as the transactions happened in the remote cache. A request is sent to remote LMS process to access undo block. If the undo block is not in the remote cache either, remote LMS process will send a grant to the foreground process to read the undo block from the disk. Foreground process accounts time to the 'gc cr disk read' wait event for this specific wait for undo grant.

There are other reasons as to why FG process might have to read undo block. One of them is that Fairness downconvert by LMS. Essentially, if a block is requested too many times for CR fabrication, then instead of LMS doing all that hard work, LMS process will simply down convert the block and send the block and grant to the requester. gv\$cr_block_server can be used to review the number of down converts, Light works etc. But, it is probably not possible to identify the reason for a block down convert ex post facto.

Why do we need this new event?

There is a very good reason why this event was introduced. Prior to 11g, waits for single block CR grants are accounted to wait event such as 'gc cr block 2-way', 'gc cr block 3-way' etc. Waits for grants on remote-instance-undo-blocks for CR fabrication is special, in the sense that, this is an additional unnecessary work from the application point of view. We need to be able to differentiate the amount of time spent waiting for undo block grants for CR fabrication vs other types of grants (such as data blocks etc). So, it looks like, Oracle has introduced this new event and I do think that this will be very useful for debugging performance issues.

Prior to 11g, you could still differentiate waits for single block grants for undo using ASH data or trace files. But, you will have to use the obj# field for this differentiation and obj# is set to 0 or -1 in the case of undo blocks/undo header blocks.

Test case

Of course, a test case would be nice, Just any regular table will do and my table structure have just two columns number, varchar2(255) with 1000 rows or so.

```
create table rs.t_1000 (n1 number, v1 varchar2(255));
insert into rs.t_1000 select n1, lpad(n1, 255, 'DEADBEAF') from (select
level n1 from dual connect by level commit;
```

1. node 1: update rs.t_1000 set v1=v1 where n1=100
2. node 2; select * from rs.t_1000 where n1=100 – just to get parsing details away.
3. node 1: alter system flush buffer cache; –flushed buffer cache.
4. node 2: select * from rs.t_1000 where n1=100 — This SELECT statement suffers from gc cr disk read.

At step 3 in our test case, I flushed the buffer cache in node 2. When I reread the block again in node 1, here is an approximate sequence of operations that occurred:

1. For SELECT statement in step 4, foreground sent a block request to LMS process in node2; LMS process in node 2 did not find the block in the buffer cache (since we flushed the buffer cache).
2. So, LMS process in node2 sent a grant to the foreground process to read the data block from disk.
3. Foreground process read the block from the disk, found that block version is higher than the query environment SCN and that there is a pending transaction in an ITL entry of the block.
4. Foreground process clones the buffer and tries to apply undo records in order to reconstruct CR version of the block to match the query environment SCN.
5. But, that pending transaction was initiated in node 2 and that undo segment is mastered by node 2. So, FG process sends a request for the block to node 2. LMS process does not find the block in the node 2 cache and sends back a grant to the FG process. Meanwhile, the FG process is still waiting and the amount of time that FG process was waiting to receive the grant is accounted towards 'gc cr disk read'.

(I formatted trace lines to improve readability)

</p.

```
PARSING IN CURSOR #18446741324873694136 len=162 dep=0 uid=0 oct=3 lid=0 tim=1103269602
hv=361365006 ad='73fba918' sqlid='7cmy6msasmzhf'
select dbms_rowid.rowid_relative_fno (rowid) fno,
dbms_rowid.rowid_block_number(rowid) block,
dbms_rowid.rowid_object(rowid) obj, v1 from rs.t_1000 where n1=100
END OF STMT
c=0,e=17012,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,plh=479790187,tim=1103269601
c=0,e=12393,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,plh=479790187,tim=1103282129
nam='SQL*Net message to client' ela= 4 driver id=1650815232 #bytes=1 p3=0 obj#=603
tim=1103288000
nam='Disk file operations I/O' ela= 7 FileOperation=2 fileno=4 filetype=2 obj#=603
tim=1103288120
nam='db file sequential read' ela= 786 file#=4 block#=2891 blocks=1 obj#=85844
tim=1103289701
nam='db file sequential read' ela= 560 file#=4 block#=2892 blocks=1 obj#=85844
tim=1103290450
nam='Disk file operations I/O' ela= 4 FileOperation=2 fileno=7 filetype=2 obj#=85844
tim=1103290546
nam='db file sequential read' ela= 542 file#=7 block#=2758 blocks=1 obj#=75154
tim=1103291149

-- RS: Following is for undo header block to find the transaction.
nam='gc cr disk read' ela= 633 p1=6 p2=176 p3=43 obj#=0 tim=1103292048
nam='db file sequential read' ela= 662 file#=6 block#=176 blocks=1 obj#=0 tim=1103292843

-- RS: Following read is for undo block itself to rollback the transaction changes.
nam='gc cr disk read' ela= 483 p1=6 p2=955 p3=44 obj#=0 tim=1103293699
nam='db file sequential read' ela= 569 file#=6 block#=955 blocks=1 obj#=0 tim=1103294355

nam='library cache pin' ela= 1045 handle address=2043988208 pin address=1969440144
100*mode+namespace=48820893384706 obj#=0 tim=1103295827
FETCH :c=0,e=8033,p=5,cr=6,cu=0,mis=0,r=1,dep=0,og=1,plh=479790187,tim=1103296065
WAIT : nam='SQL*Net message from client' ela= 323 driver id=1650815232 #bytes=1 p3=0
obj#=0 tim=1103296522
FETCH :c=0,e=24,p=0,cr=1,cu=0,mis=0,r=0,dep=0,og=1,plh=479790187,tim=1103296586
STAT id=1 cnt=1 pid=0 pos=1 obj=75154 op='TABLE ACCESS BY INDEX ROWID T_1000 (cr=7 pr=5
pw=0 time=6505 us cost=2 size=261 card=1)'
STAT id=2 cnt=1 pid=1 pos=1 obj=85844 op='INDEX RANGE SCAN T_1000_N1 (cr=3 pr=2 pw=0
time=2494 us cost=1 size=0 card=1)'
WAIT : nam='SQL*Net message to client' ela= 1 driver id=1650815232 #bytes=1 p3=0 obj#=0
tim=1103296728
```

Trace file analysis

Let's review few lines from the trace file. Block 2758 holds the row physically. After reading that block, a 'gc cr disk read' wait event is encountered. Essentially, the FG identified that there is a pending transaction, so sent a request to LMS accounting the wait time to 'gc cr disk read' event. For this 'gc cr disk read' event, p1 is file_id, p2 is block_id, p3 seems to be a counter increasing by 1 for each of these waits. For example, for the next gc cr disk read, p3 is set to 44. obj#=0 indicates that this is an undo block or undo header block.

Notice that next line indicates a physical read for that undo block occurred and the obj# is set to 0.

```
nam='db file sequential read' ela= 542 file#=7 block#=2758 blocks=1 obj#=75154
tim=1103291149
nam='gc cr disk read' ela= 633 p1=6 p2=176 p3=43 obj#=0 tim=1103292048
nam='db file sequential read' ela= 662 file#=6 block#=176 blocks=1 obj#=0 tim=1103292843
```

Commit Cleanout

Commit cleanouts are another reason why you would encounter this wait event. When a session commits, that session will revisit the modified blocks to clean out the ITL entries in the modified blocks. But, this cleanout does not happen in all scenarios. For example, if the number of block changes exceeds list of blocks that session maintains in SGA or if the block is not in the buffer cache anymore then the session will mark the transaction table as committed, without cleaning out ITL entries in the actual blocks.

Next session reading that block will close out the ITL entry in the block with an upper bound SCN. Let's tweak our test case little bit to simulate commit cleanouts.

One notable difference in the test case below is that, after flushing the buffer cache, we go to the session updating the block and commit the transaction and flush the buffer cache again. SELECT statement is executed after the commit. So, update session will not clean out ITL entry and the undo blocks are flushed to the disk. When we read the same block from a different node, then the reader session will cleanout the ITL entry. To identify whether the transaction is committed or not, session in node2 must read the undo block. Grants for that read are accounted towards gc cr disk read wait event.

1. node 1: update rs.t_1000 set v1=v1 where n1=100
2. node 2; select * from rs.t_1000 where n1=100 – just to get parsing details away.
3. node 1: alter system flush buffer cache; –flushed buffer cache.
4. commit in node 1 from the other session.
5. node 1: alter system flush buffer cache; –flushed buffer cache to remove undo blocks from cache.
6. node 2: select * from rs.t_1000 where n1=100 .

Trace file

My comments are inline

```
...
-- block with the row is read below
WAIT : nam='db file sequential read' ela= 417 file#=7 block#=2758 blocks=1 obj#=75154
tim=783492616
WAIT : nam='Disk file operations I/O' ela= 2 FileOperation=2 fileno=6 filetype=2
obj#=75154 tim=783492751
-- Undo header block is read with time accounted towards gc cr disk read. Note that,
there are no other reads after this.
-- In the prior test case, there was an additional read to read undo block;
-- In this test case, no additional read for undo block as only commits need to be
cleaned out.
WAIT : nam='gc cr disk read' ela= 12779 p1=6 p2=160 p3=41 obj#=0 tim=783507346
WAIT : nam='db file sequential read' ela= 778 file#=6 block#=160 blocks=1 obj#=0
tim=783508447
```

So, what can we do?

Is there any thing you can do to improve the performance of the application and reduce 'gc cr disk read' wait events? There are few options that you can consider, some are long-term options though.

1. Consider Application affinity. If the application is running in the same node that transactions are aggressively modifying the objects, you can reduce/eliminate the grants. Application affinity is important even though cache fusion is in play.
2. Of course, tune the statement such a way that number of block visits are reduced.
3. Reduce commit cleanouts. For example, if a program is excessively modifying the block in node 1. Subsequent program is reading the same blocks in node 2, then you may have to cleanout the block manually from node 1 by reading the blocks. BTW, parallel queries do not perform commit cleanouts, only serial queries will perform commit cleanouts. Another operation that can get stuck is index creation. If the table has numerous blocks without commit cleanouts, then index build might be slower and can suffer from gc or disk read events. In this case, Reading through all blocks of the table through SELECT statements might be good enough.
4. Don't keep long pending transactions on highly active tables in RAC environment. This typically happens for applications that uses tables as queue to pick up jobs i.e. fnd_concurrent_requests table in EBusiness suite. Proper configuration of concurrent manager, optimal values for sleep and cache might help here.

Summary

In summary, this is a useful event to differentiate CR fabrication performance issues from other performance issues. Using few techniques mentioned here, you can reduce the impact of this event.