

OBJECT REMASTERING IN RAC

Introduction

In RAC, every data block is mastered by an instance. Mastering a block simply means that master instance keeps track of the state of the block until the next reconfiguration event (due to instance restart or otherwise).

Hash to the master

These data blocks are mastered in block ranges. For example, range of blocks starting from file 10, block 1 through block 128 may be mastered by instance 1, blocks from file 10, block 129 through 256 are mastered by instance 2 etc. Of course, there are differences between various versions 10g, 11g etc, but Idea here is that block ranges are uniformly mastered between various instances so that Global cache grants are evenly distributed among the instances. Interestingly, length of the block range is 128 from 10g onwards (Julian Dyke mentioned that is 1089 in 9i, but I have not personally tested it). Of course, Support recommends you to unset `db_file_multiblock_read_count` which will be auto adjusted to 128 which means that Full block range can be read with fewer GC messages, I suppose. I digress.

Further, Michael Möller (“M2”) pointed out that this hash-algorithm is further optimized: The hash-algorithm used when initially computing the master node from the DBA, results in a "virtual master", which is then translated to a real (online&open) master by a lookup table (the length of which is the maximum number of possible nodes (128 ?)). This means that when one node goes off/on-line, RAC does NOT have to recalculate the hash for all blocks, but only distribute the new Hash-to-node table. (One can later visualize dynamic remastering as an additional lookup table between the hash value and node. This table also needs redistributing on node changes.)

Following SQL is helpful in showing masters and owners of the block. This SQL joins the fixed tables, `x$kjbl` with `x$le` to retrieve resource name. If you are familiar with Oracle locking strategy, you would probably recognize the format of these cache fusion (aka old PCM) locks. Lock type in this case is BL, id1 is block# and id2 is file_id in this case. Column `kjblname2` provides a decimal format lock resource.

Please observe the output below:

1. Block range: File 1, block 6360-6376 is mastered by node 3 and also owned by node 3.
2. Block range: File 1, blocks upto 10709 is mastered by instance 0 and owned by instance 3.
3. Next block range from 10770 is mastered by instance 2 and owned by 3.

Note that this output is coming from a database with no remastering done yet.

```
REM In kjblname2 first entry before ',' is block and seond entry file_id*65536 for BL locks
```

```
select kj.*, le.le_Addr from (
select kjblname, kjblowner, kjblmaster, kjbllockp,
substr ( kjblname2, instr(kjblname2,',')+1, instr(kjblname2,',',1,2)-instr(kjblname2,',',1,1)-1)/65536 fl,
substr ( kjblname2, 1, instr(kjblname2,',')-1) blk
from x$kjbl
```

) kj, x\$le le
 where le.le_kjbl = kj.kjbllockp
 order by le.le_addr
 /

KJBLNAME	KJBLNAME2	KJBLOWNER	KJBLMASTER	FL	BLK	LE_ADDR
[0x18d8] [0x10000], [BL]	6360, 65536, BL	3	3	1	6360	000000038DF9AB08
...						
[0x18e7] [0x10000], [BL]	6375, 65536, BL	3	3	1	6375	00000038DFBF818 #2
[0x18e8] [0x10000], [BL]	6376, 65536, BL	3	3	1	6376	000000038DFD3BA0
...						
[0x29d1] [0x10000], [BL]	10705, 65536, BL	3	0	1	10705	00000005D6FE9230
...						
[0x29d5] [0x10000], [BL]	10709, 65536, BL	3	0	1	10709	000000038EFBB990
[0x2a12] [0x10000], [BL]	10770, 65536, BL	3	2	1	10770	000000075FFE3C18
...						
[0x2a18] [0x10000], [BL]	10776, 65536, BL	2	2	1	10776	000000038EFA8488 #1
[0x2a19] [0x10000], [BL]	10777, 65536, BL	3	2	1	10777	000000038EFB7F90
[0x2a1a] [0x10000], [BL]	10778, 65536, BL	1	2	1	10778	000000038EFC318

Let's consider three simple cases of SELECT sql statement running instance 3:

1. A session is trying to access the block file 1, block 10776, but that block is mastered by instance 2 and also that block is owned by instance 2 (meaning, it is in instance 2 cache). So, instance 3 will send a PR (Protected Read) mode BL lock request on that block to instance 2. Ignoring any additional complexities, instance 2 will grant PR mode lock to instance 3 and transfer the block to instance 3. Obviously, this involves multiple GC messages, grants and block transfer. Statistics 'gc remote grants' gets incremented too.
2. Let's consider that session is trying to access another block: file 1, block 6375. That block is mastered by instance 3 and also owned by instance 3. At this point, there is no additional GCS/GES processing is needed and the session pin that buffer and continue the work.
3. Let's consider a third case. Session is trying to access file 1 block 6374. That block is not in any buffer cache, but instance 3 is master of the block, so local affinity locks are acquired with minimal GC messages and waits. That block is read from the disk in to the buffer cache

In the case #2 and #3 above, requesting instance also is the master node of a block or block range. In these cases, statistics 'gc local grants' is incremented and cheaper local affinity locks on those block ranges are acquired avoiding many Global cache messages.

So far so good, but what if, say instance 1, is reading one object (table, index etc) aggressively, but other instances are not reading that object at all? [through some sort of application node partitioning or just plain workload]. Does it make sense for the instance accessing that object aggressively request a grant to the remote instance(s) for each OPEN on that object's blocks? Especially, if the blocks are read in to the buffer cache, but disappears soon from the buffer cache? Wouldn't that be better if the instance reading that object aggressively is also the master of that object, as in the cases #2 and #3 above?

In addition to that, if the block is supposed to be thrown away from buffer cache (close of BL lock) or if the block needs to be written, then that will involve additional overhead/messaging between the master instance and owner instance since the ownership needs to be communicated back to the master of the block.

Enter Object remastering.

Object remastering

There are many new features in 10g/11g RAC stack. One of them is Object remastering feature. This feature was implemented in 10gR1 and improved in 10gR2 and further enhanced in 11g. I realize there are parameters in 9i also, but I don't think it worked as intended.

With object remastering feature, if an object is accessed by an instance aggressively, then that instance will become the master of the object reducing gc remote grants improving performance of the application. In the prior sentence, I used the word "accessed", but it is a loose term, and the correct term is if the instance is requesting much BL locks on an object, then that object can be remastered. In an ideal world, even if the application is not partitioned, remastering of the objects that were accessed aggressively from one instance will acquire cheaper local instance affinity locks and effective RAC Tax will be minimal.

Well, I said, in an ideal world :-) There are few issues here:

1. Instances do not remember prior mastership across restarts. This means that instance needs to re-learn the object mastership map after every restart. I can see the complexities of remembering the mastership, but it is possible to implement that.
2. Remastering is not exactly cheap. Instance GRD is frozen during reconfiguration and in a very busy instances, this can take many seconds leading to instance freeze for several seconds. While 10gR2 introduce parallel reconfiguration (`_rcfg_parallel_Replay` parameter controls this behavior) using all LMS processes to complete the reconfiguration, still, several seconds of freeze is not exactly acceptable in many environments.
3. I advice my clients to keep LMS processes to a lower value (3 to 5), at the most, but instance reconfiguration effective parallelism is reduced if we reduce number of LMS processes.
4. Last, but not the least important point is that, default values of few parameters that trigger remastering events are quite low for busy environments causing frequent remastering of objects. In an E-Business World, minor mismanagement in the manager configuration can lead to a massive reconfiguration issues.

Parameters, views and internals

Few parameters are controlling this behavior, not well documented, my test case results are not very accurate either. But, these parameters are giving us a picture of what is going on internally. These parameters are applicable to 10gR2 and below. For 11g, whole set of different parameters comes in to play and I will blog about the differences in another blog entry.

`X$object_affinity_statistics` maintains the statistics about objects and OPENs on those objects. It is important to understand the difference between OPEN and Buffer access. If the block is in the cache already in a suitable mode, there is no need for BL opens on that block. So, if the sessions are accessing the same block repeatedly without requesting any additional BL locks, then the count is not incremented. So, OPEN is simply a number of BL request initiated in an ephemeral time frame.

LCK0 process maintains these object affinity statistics. If an instance opens 50 more opens on an object then the other instance (controlled by `_gc_affinity_limit` parameter), then that object is a candidate for remastering. That object is queued and LMD0 reads the queue and initiates GRD freeze. LMON performs reconfiguration of buffer cache locks working with LMS processes. All these are visible in LMD0/LMON trace files. Parameter `_gc_affinity_time` controls how often the

queue is checked to see if the remastering must be triggered or not with a default value of 10 minutes.

Now, you don't want just any object as a candidate for remastering, meaning, if instance 1 opened 101 BL locks on that object and instance 2 opened 50 BL locks on that object, you don't want to trigger object remastering. Only objects with higher amount of BL lock requests must be queued for remastering. Well, that threshold seems to be controlled by another parameter `_gc_affinity_minimum`: This parameter is defined as "minimum amount of dynamic affinity activity per minute" to be a candidate for remastering. Defaults to 2500 and I think, it is lower in a busy environment.

Few lines from LMD0 trace files showing that LMD0 is reading a request queue:

```
* kjdrchkdrm: found an RM request in the request queue
  Transfer pkey 6589904 to node 3
*** 2009-10-12 11:41:20.257
```

How bad can it get?

Performance can suffer if there are remastering issues. Following AWR report shows that few instances froze due to DRM reconfiguration issue. Same type of freeze is visible in all other nodes too. gc buffer busy is a side effect of DRM freeze (not always, but in this case).

Event	waits	Time (s)	Avg wait (ms)	%Total Call Time	wait Class
gc buffer busy	1,826,073	152,415	83	62.0	cluster
CPU time		30,192		12.3	
enq: TX - index contention	34,332	15,535	453	6.3	Concurrenc
gcs drm freeze in enter server	22,789	11,279	495	4.6	Other
enq: TX - row lock contention	46,926	4,493	96	1.8	Applicatio

At the same time, a storm of DRM were started. This lead to repetitive DRM freeze and instance configuration leading to massive performance issues.

```
* received DRM start msg from 2 (cnt 5, last 1, rmno 404)
Rcvd DRM(404) Transfer pkey 1598477 from 3 to 2 oscan 1.1
Rcvd DRM(404) Dissolve pkey 6100030 from 2 oscan 0.1
Rcvd DRM(404) Dissolve pkey 5679943 from 2 oscan 0.1
Rcvd DRM(404) Transfer pkey 6561036 from 0 to 2 oscan 1.1
Rcvd DRM(404) Transfer pkey 5095243 to 2 oscan 0.1
...
```

A small test case

Let's walk through a test case that shows DRM in play. Query used index access path to read nearly all blocks from a big index.

Session #1:

```
select data_object_id from dba_objects where object_name='WMS_N1';
DATA_OBJECT_ID
-----
6099472
```

REM No affinity statistics yet.

```
select * from x$object_affinity_statistics where object=6099472;
no rows selected.
```

```
REM executing a costly select statement
select /*+ index(a WMS_N1 )*/ count(*) from big_table1 a;
```

Session #2: I was monitoring the DRM tables:

```
REM DRM operations completed so far is 409. we will keep track this
count to monitor remastering event. There are few
REM other interesting fields in this view.
```

```
select drms from X$KJDRMAFNSTATS;
```

```
DRM
----
409
```

```
REM I see that 23442 opens on that index already since the session #1
started running.
```

```
select * from x$object_affinity_statistics where object=6099472;
ADDR                INDX    INST_ID    OBJECT    NODE    OPENS
-----
FFFFFFFF7C05BFA8    14      1    6099472    1      23442
```

```
REM No mastering has kicked in for that object.
select * from v$gcspfmaster_info where object_id=6099472;
```

```
no rows selected
```

```
REM few seconds later, open count increased from 23442 -> 33344.
```

```
select * from x$object_affinity_statistics where object=6099472;
ADDR                INDX    INST_ID    OBJECT    NODE    OPENS
-----
FFFFFFFF7C05BFA8    16      1    6099472    1      33344
```

```
REM No remastering has kicked in for that object.
```

```
select * from v$gcspfmaster_info where object_id=6099472;
```

```
no rows selected
```

```
REM Surprisingly, while session #1 is still executing, the counter for
OPENS was zeroed out even when though DRM has not been triggered yet.
```

```
REM OPENS Increased to 1229 from 0 since the session #1 is still
REM executing.
```

```
ADDR                INDX    INST_ID    OBJECT    NODE    OPENS
-----
FFFFFFFF7C05BFA8    0      1    6099472    1      1229
```

```
REM Approximately 10 minutes or so later, Remastering kicked in..
```

```
REM # of DRMS increased from 409 to 411
```

```
select drms from X$KJDRMAFNSTATS;
```

```
DRM
----
411
```

```
REM Remastering of that index happened. Now the master is 0 which is
REM instance 1.
```

```
select * from v$gcspfmaster_info where object_id=6099472;
```

FILE_ID	OBJECT_ID	CURRENT_MASTER	PREVIOUS_MASTER	REMASTER_CNT
0	6099472	0	32767	1

REM Opens are still increasing but remastering has already occurred.

```
select * from x$object_affinity_statistics where object=6099472;
```

ADDR	INDX	INST_ID	OBJECT	NODE	OPENS
FFFFFFFF7C05AF78	2	1	6099472	1	42335
FFFFFFFF7C05AF78	3	1	6099472	2	1

REM LMON trace files are also indicating transfer of that pkey. Notice
REM that pkey is same as object_id

```
*** 2010-03-23 10:41:57.355
```

```
Begin DRM(411) - transfer pkey 6099472 to 0 oscan 0.0  
ftd received from node 1 (4/0.30.0)  
all ftds received
```

REM few seconds later, opens have been reset again.

```
select * from x$object_affinity_statistics where object=6099472;
```

ADDR	INDX	INST_ID	OBJECT	NODE	OPENS
FFFFFFFF7C05BFA8	0	1	6099472	1	7437

REM Still the master is instance 1.

```
select * from v$gcspfmaster_info where object_id=6099472;
```

FILE_ID	OBJECT_ID	CURRENT_MASTER	PREVIOUS_MASTER	REMASTER_CNT
0	6099472	0	32767	1

Essentially, an object was remastered after excessive BL locking requests (in a loose term accesses)
on that index.

UNDO and affinity

Mastering of Undo segments differ from non-undo segment mastering. With non-undo segments, all the blocks are mastered by a hash technique spreading mastership among instances for a segment. Only after an instance opens BL locks aggressively on a segment that segment is mastered. But, for undo segments, Instance that activates an undo segment masters the segment immediately. This makes sense, since that undo segment will be used by the instance opening the segment in most cases. Parameter `_gc_undo_affinity` controls whether this dynamic undo remastering is enabled or not.

Since undo segments do not have real `object_ids`, a dummy `object_ids` over a value of 4294950912 is used. For example, undo segment 1 (with `usn=1`) will have an `object_id` of 4294950913, `usn=2` will have `object_id` of 4294950914 etc. As you can see below, undo segments are mastered immediately to the instance opening that undo segment.

```
select object_id, object_id-4294950912 usn, current_master,  
previous_master,  
remaster_cnt from v$gcspfmaster_info where object_id>4294950912
```

OBJECT_ID	USN	CURRENT_MASTER	PREVIOUS_MASTER	REMASTER_CNT
-----------	-----	----------------	-----------------	--------------

```

-----
4294950913      1      0      32767      1
4294950914      2      0      32767      1
4294950915      3      0      32767      1
4294950916      4      0      32767      1
4294950917      5      0      32767      1
4294950918      6      0      32767      1
4294950919      7      0      32767      1
4294950920      8      0      32767      1
4294950921      9      0      32767      1
4294950922     10      0      32767      1
4294950923     11      1      32767      1
4294950924     12      1      32767      1
4294950925     13      1      32767      1
...

```

REM Notice that usno 0 is in both instances. That is system undo REM segment. As you can see below first 10 undo segments are mastered REM by node 1 is next 3 are mastered by instance 2.

```

select inst_id, usn, gets from gv$rollstat where usn <=13
order by inst_id, usn

```

INST_ID	USN	GETS
1	0	3130
1	1	108407
1	2	42640
1	3	43985
1	4	41743
1	5	165166
1	6	43485
1	7	109044
1	8	23982
1	9	39279
1	10	48552
2	0	4433
2	11	231147
2	12	99785
2	13	1883

I was not successful in triggering another undo segment remastering event. I created one active transaction generating 200K undo blocks in one node, another node was reading that table and I can see enormous waits for those undo blocks. But, I didn't see any remastering events related to that undo segment. Not sure why it did not work, may be the conditions for the undo segment remastering is different.

[PS: I am able to manually remaster the undo segment using lkdebug command discussed below: So, code must be remastering the undo segments automatically too, but may be some other conditions must be met.

REM Initially

```

1* select * from v$gcspfmaster_info where object_id=431+4294950912
SQL> /

```

FILE_ID	OBJECT_ID	CURRENT_MASTER	PREVIOUS_MASTER	REMASTER_CNT
0	4294951343	0	32767	1

```

Oradebug lkdebug -m pkey 4294951343

```

```
* kjdrchkdrm: found an RM request in the request queue
  Transfer pkey 4294951343 to node 1
*** 2010-03-24 12:47:29.011
Begin DRM(520) - transfer pkey 4294951343 to 1 oscan 0.1
ftd received from node 0 (4/0.31.0)
all ftds received
```

```
1* select * from v$gcspfmaster_info where object_id=431+4294950912
SQL> /
```

FILE_ID	OBJECT_ID	CURRENT_MASTER	PREVIOUS_MASTER	REMASTER_CNT
0	4294951343	1	0	2

```
]
```

I am not preaching that you should modify these undocumented parameters. Far from it. Understand the parameters, if you run in to wait events such as 'gc remaster', 'gcs freeze for instance reconfiguration', understand whether the default values are quite low. Work with support and see if this can be tuned.

Manual remastering

You can manually remaster an object with oradebug command

```
oradebug lkdebug -m pkey <object_id>
```

This enqueues an object remaster request. LMD0 and LMON completes this request

```
*** 2010-01-08 23:25:54.948
* received DRM start msg from 1 (cnt 1, last 1, rmno 191)
Rcvd DRM(191) Transfer pkey 6984154 from 0 to 1 oscan 0.0
ftd received from node 1 (8/0.30.0)
ftd received from node 0 (8/0.30.0)
ftd received from node 3 (8/0.30.0)
```

Current_master starts from 0.

```
1* select * from v$gcspfmaster_info where object_id=6984154
SQL> /
```

FILE_ID	OBJECT_ID	CURRENT_MASTER	PREVIOUS_MASTER	REMASTER_CNT
0	6984154	1	0	2

```
SQL> oradebug lkdebug -m pkey 6984154
Statement processed.
```

```
SQL> select * from v$gcspfmaster_info where object_id=6984154
2 /
```

FILE_ID	OBJECT_ID	CURRENT_MASTER	PREVIOUS_MASTER	REMASTER_CNT
0	6984154	2	1	3

Summary

In summary, remastering is a great feature. It is a pity that some times, we fall victims of the side effects. So, if you run in to issues with remastering, don't disable it, but see if you can tune those parameter upwards so as to control the remastering events. If you stil want to disable DRM completely, I would recommend setting `_gc_affinity_limit` and `_gc_affinity_minimum` to much higher value, say 10Million. Setting the parameter `_gc_affinity_time` to 0 will completely disable DRM, but that also means that you can not manually remaster objects. Further, Arup mentioned that `x$object_affinity_statistics` is not maintained if DRM is disabled.

Again, these are undocumented parameters. Before you change these parameters make sure that Support agrees with you.

[Many Thanks **Arup Nanda** and **Michael Möller** (aka “M2”) for reviewing this blog entry, they contributed heavily to some of my discussion. But, any mistakes in this document is solely of mine.]