

Cache Buffer Chains Demystified

Arup Nanda, Longtime Oracle DBA

ABSTRACT

You must have seen sessions waiting on the event “latch: cache buffers chains” from time to time. If you ever wondered what this means and how you can reduce time spent on it, this paper is for you. Here you will learn how buffer cache works, how Oracle multi-versioning works, how buffers are allocated and deallocated, what hash chain is and how buffers are linked to it, what the role of cache buffer chain latch is and why sessions wait on it, how to find the objects causing the contention and how to reduce the time spent on that event.

TARGET AUDIENCE

This paper is intended for DBAs who want to know the internals of cache management and solve the cache buffer chain latch waits. An intermediate level understanding of the Oracle database administration is required.

EXECUTIVE SUMMARY

Readers will learn:

- How buffer cache works
- How buffers are populated
- About buffer states and versioning
- How buffers are flushed
- About the role of Cache Buffer Chain latch
- How to reduce CBC Latches
- About other kinds of latches

BACKGROUND

While exploring the reasons for the slowness in database sessions, you check the wait interface and see the following output:

```
SQL> select state, event from v$session where sid = 123;
```

```
STATE    EVENT
```

```
-----  
WAITING latch: cache buffers chains
```

This event is more common, especially in applications that perform a scan of a few blocks of data. To resolve it, you should understand what the cache buffers chains latch is and why sessions have to wait on it. To understand that you must understand how the Oracle buffer cache works. We will explore these one by one, and close with the solution to reducing the cache buffers chains latch waits.

HOW BUFFER CACHE WORKS

The buffer cache (BC), resident inside the SGA of an Oracle instance is designed to hold blocks that come from the database. When a user issues a statement such as the following:

```
update EMP
```

```
set NAME = 'ROB'
where EMPNO = 1
```

the Oracle server process assigned to the session performs the following actions:

- 1) locates the block that contains the record with EMPNO = 1
- 2) loads the block from the database file to an empty buffer in the buffer cache
- 3) if an empty buffer is not immediately found, locates an empty buffer or forces the DBWn process to write some dirty buffers to make room
- 4) updates the NAME column to ROB in the buffer

In step 1, we assume an index is present and hence the server process can locate the single block immediately. If the index is not present, Oracle will need to load all the blocks of the table EMP into the buffer cache and check for matching records one by one.

The above description has two very important concepts:

- 1) a block, that is the smallest unit of storage in the database
- 2) a buffer, that is a placeholder in the buffer cache used to hold a block.

Buffers are just placeholders, which may or may not be occupied. They can hold exactly one block at a time. Therefore for a typical database where the block size is set to 8KB, the buffers are also of size 8KB. If you use multiple block sizes, e.g. 4KB or 16KB, you would have to define multiple buffer caches corresponding to the other block sizes. In that case the buffer sizes will match the block sizes corresponding to those blocks.

When buffers come to the cache, the server process must scan through them to get the value it wants. In the example shown above, the server process must find the record where EMPNO=1. To do so, it has to know the location of the blocks in the buffers. The process scans the buffers in a sequence. So, buffers should ideally be placed in a sequence, e.g. 10 followed by 20, then 30, etc. However this creates a problem. What happens when, after this careful placement buffers, buffer #25 comes in? Since it falls between 20 and 30, it must be inserted inbetween, i.e. Oracle must move all the buffers after 20 one step towards the right to make room for the new buffer #25. Moving of memory areas in the memory is not a good idea. It costs expensive CPU cycles, requires all actions on the buffers (even reading) to stop for the duration and is prone to errors.

Therefore, instead of moving the buffers around, a better approach is to put them in something like a linked list. Fig 1 shows how that is done. Each of the buffers has two pointers: which one is behind and which one is right ahead. In this figure, buffer 20 shows that 10 is in front and 30 is the one behind. This would be the case regardless of the actual position of the buffers. When 25 comes in, all we have to do is to update the “behind pointer” of 20 and “ahead pointer” of 30 to point to 25. Similarly the “ahead pointer” and “behind pointer” of 25 are updated to point to 30 and 20 respectively. This simple update is much quicker, does not need activity to stop on all the buffers except the ones being updated and less error-prone.

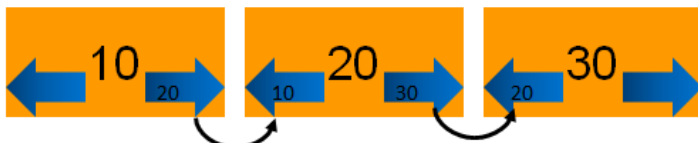


Figure 1 Buffer Linked List

However, there is another problem. This is just one of the lists. Buffers are used for other purposes as well. For instance, the LRU algorithm needs a list of buffers in LRU order, the DBWn process needs a list of buffers for writing to the disk, etc. So, physically moving the buffers to specific lists is not just impractical, it's impossible at the same time. Oracle employs a simpler technique to overcome the obstacle. Rather than placing the actual buffers in the linked list, Oracle creates a simpler, much lighter structure called **buffer header** as a pointer to an actual buffer. This buffer cache is moved around, leaving the actual

buffer in place. This way, the buffer header can be listed in many types of lists at the same time. These buffer headers are located in the *shared pool*, not the buffer cache. This is why you will find the reference to buffers in the shared pool.

BUFFER CHAINS

The buffers are placed in strings. Compare that to rows of spots in a parking lot. Cars come in to an empty spot in a row. If they don't find one, they go to the next row and so on. Similarly buffers are located on the cache as rows. However, unlike the parking spots which are physically located next to each other, the buffers are logically placed as a sequence in the form of a linked list, described in the above section. Each linked list of buffers is known as a **buffer chain**, as shown in Fig 2.

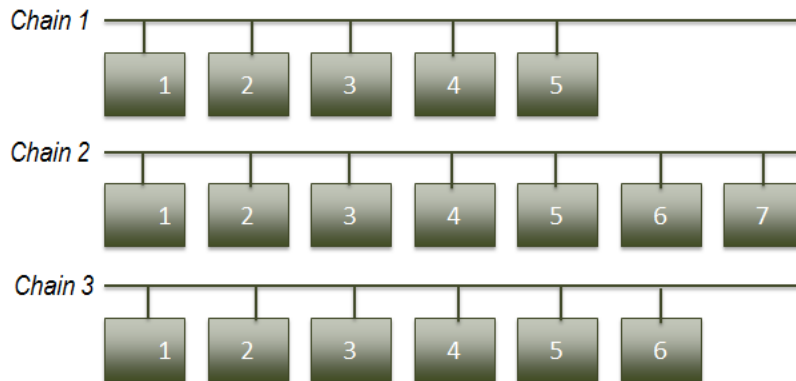


Figure 2 Buffer Chains

Notice how each of the three chains has different numbers of buffers. This is quite normal. Buffers are occupied only when some server process brings them up from the block. Otherwise the buffers are free and not linked to anything. When the buffers are freed up, perhaps because some process such as DBWn writes their contents to the disk, they are removed from the list—a process known as unlinking from the chain. So, in a normal database, buffers will be constantly linked to and unlinked from a chain—making the chain long or small depending on the frequency of either activity. The number of buffer chains is determined by the hidden database parameter `_db_block_hash_buckets`, which is automatically calculated from the size of the buffer cache.

When a server process wants to access a specific buffer in the cache, it starts at the head of the chain and goes on to inspect each buffer in sequence until it finds what it needs. This is called walking the chain. You might be wondering about a nagging question here—when a buffer comes to the cache, who decides which of the three chains it should be linked to and how? A corollary to that is a challenge posed by the server process in trying to find a specific buffer in the cache. How does the process know which chain to walk? If it always starts at the chain 1, it will take an extraordinary amount of time to locate the block. Typical buffer caches are huge, so the number of chains may run into 10's of thousands, if not 100's. So, searching all the chains is not practical. On the other hand, if Oracle were to maintain a memory table showing which blocks are located in which buffers is not practical either, because maintaining that memory table will be time consuming and make the process sequential. Several processes can't read chains in parallel then.

Oracle solves the problem in a neat manner. Consider the parking lot example earlier. What if you forget where you parked your car? Suppose after you come out of the mall, you find that all the cars have been buried under a thick pile of snow making identification of any of the cars impossible. So, you would have to start at the first car at the first row, dust off the snow from the license plate, check for your car, move on to the next, and so on. Sounds like a lot of work, doesn't it? So, to help forgetful drivers, the mall marks the rows with letter codes and asks the drivers to park in the row matching the first letter of their last name. John Smith will need to park in row S, and in row S only, even if row T or row R are completely empty. In that case, when John returns to find his car and forgets where it is, he will know to definitely find it in row S. That will be the domain of his search—much, much better than searching the entire parking lot.

Similarly, Oracle determines which specific chain a buffer should be linked to. Every block is uniquely identified by a **data block address** (DBA). When the block comes to the buffer cache, Oracle applies a hash function to determine the buffer chain number and places the block in a buffer in that chain alone. Similarly, while looking up a specific buffer, Oracle applies the same hash function to the DBA, instantly knows the chain the buffer will be found and walks that specific buffer only. This makes accessing a buffer much easier compared to searching the entire cache.

To find out the data block address, you need to first get the relative file# and block#. Here is an example where I want to find out the blocks of the table named CBCTEST.

```
SQL> select
2     col1,
3     dbms_rowid.rowid_relative_fno(rowid) rfile#,
4     dbms_rowid.rowid_block_number(rowid) block#
5 from cbctest;
```

COL1	RFILE#	BLOCK#
1	6	220
2	6	220
3	6	220
4	6	221
5	6	221
6	6	221

6 rows selected.

From the output we see that there are 6 rows in this table and they are all located in two blocks in a file with relative file# 6. The blocks are 220 and 221. Using this, we can get the data block address. To get the DBA of the block 220:

```
SQL> select dbms_utility.make_data_block_address(6,220) from dual;
```

```
DBMS_UTILITY.MAKE_DATA_BLOCK_ADDRESS(6,220)
-----
25166044
```

The output shows the DBA of that block is 25166044. If there are three chains, we could apply a *modulo* function that returns the remainder from an input after dividing it by 3:

```
SQL> select mod(25166044,3) from dual;
```

```
MOD(25166044,3)
-----
1
```

So, we will put it in chain #1 (assuming there are three chains and the first chain starts with 0). The other block of that table, block# 221 will end up in chain #2:

```
SQL> select dbms_utility.make_data_block_address(6,221) from dual;
```

```
DBMS_UTILITY.MAKE_DATA_BLOCK_ADDRESS(6,221)
-----
25166045
```

```
SQL> select mod(25166045,3) from dual;
```

```
MOD(25166045,3)
-----
2
```

And so on. Conversely, Oracle if we get a DBA, we can apply the mod() function and the output shows the chain it can be found on. Oracle does not use the exact mod() function as shown here; but a more sophisticated hash function. The exact mechanics of the function is not important; the concept is similar.

MULTI-VERSIONING OF BUFFERS

Consider the update SQL statement shown in the beginning of the paper. When Oracle updates the buffer that already exists in the buffer cache, it does not directly update it. Instead, it creates a copy of the buffer and updates that copy. When a query selects data from the block as of a certain SCN number, Oracle creates a copy of the buffer as of the point in time of interest and returns the data from that copy. As you can see, there might be more than a single copy of the same block in the buffer cache. While searching for a buffer the server process needs to search for the versions of the buffer as well. This makes the buffer chain even longer.

To find out the specific buffer of a block, you can check the view V\$BH (the buffer headers). The column OBJD is the object_id. (Actually it's the DATA_OBJECT_ID. In this case both are the same; but may not be in all cases). Here are the columns of interest to us:

- FILE# - the file_id
- BLOCK# - the block number
- CLASS# - the type of the block, e.g. data block, segment header, etc. Shown as a code
- STATUS - the status of the buffer, Exclusive Current, Current, etc.

To make it simpler to understand, we will use a decode() on the class# field to show the type of the block. With that, here is our query:

```
select file#, block#,
       decode(class#,1,'data block',2,'sort block',3,'save undo block', 4,
'segment header',5,'save undo header',6,'free list',7,'extent map',
8,'1st level bmb',9,'2nd level bmb',10,'3rd level bmb', 11,'bitmap block',
12,'bitmap index block',13,'file header block',14,'unused',
15,'system undo header',16,'system undo block', 17,'undo header',
18,'undo block') class_type, status
from v$bh
where objd = 99360
order by 1,2,3
/
```

FILE#	BLOCK#	CLASS_TYPE	STATUS
6	219	segment header	cr
6	221	segment header	xcur
6	222	data block	xcur
6	220	data block	xcur

4 rows selected.

There are 4 buffers. In this example we have not restarted the cache. So there are two buffers for the segment header. There is one buffer for each data block – 220 and 221. The status is "xcur", which stands for Exclusive Current. It means that the buffer was acquired (or filled by a block) with the intention of being modified. If the intention is merely to select, then the

status would have shown CR (Consistent Read). In this case since the rows were inserted modifying the buffer, the blocks were gotten in xcur mode. From a different session update a single row. For easier identification I have used Sess2> as the prompt:

```
Sess2> update cbctest set col2 = 'Y' where col1 = 1;
```

1 row updated.

From the original session, check the buffers:

FILE#	BLOCK#	CLASS_TYPE	STATUS
6	219	segment header	cr
6	220	segment header	xcur
6	220	data block	xcur
6	220	data block	cr
6	221	data block	xcur

5 rows selected.

There are 5 buffers now, up one from the previous seven. Note there are two buffers for block ID 220. One CR and one xcur. Why two?

It's because when the update statement was issued, it would have modified the block. Instead of modifying the existing buffer, Oracle creates a "copy" of the buffer and modifies that. This copy is now XCUR status because it was acquired for the purpose of being modified. The previous buffer of this block, which used to be xcur, is converted to "CR". There can't be more than one XCUR buffer for a specific block, that's why it is exclusive. If someone wants to find out the most recently updated buffer, it will just have to look for the copy with the XCUR status. All others are marked CR.

Suppose from a third session, update a different row in the same block.

```
Sess3> update cbctest set col2 = 'Y' where col1 = 2;
```

1 row updated.

From the original session, find out the buffers.

FILE#	BLOCK#	CLASS_TYPE	STATUS
6	219	segment header	xcur
6	219	segment header	cr
6	221	data block	xcur
6	220	data block	xcur
6	220	data block	cr
6	220	data block	cr
6	220	data block	cr

There are 4 buffers for block 220 now - up from 2. What happened? Since the buffer was required to be modified once more, Oracle created yet another "copy", marked it "xcur" and relegated the older one to "cr". What about the extra CR copy? That was done because Oracle had to perform something called CR processing to create a CR copy from another CR copy or an XCUR copy.

You can notice how the number of buffers proliferates. Let's change the experiment a little bit. From a 4th session, select from the table, instead of updating a row:

```
Sess4> select * from cbctest;
```

From the original session, check for the buffers.

FILE#	BLOCK#	CLASS_TYPE	STATUS
6	219	segment header	xcur
6	219	segment header	cr
6	221	data block	xcur
6	220	data block	xcur
6	220	data block	cr
6	220	data block	cr
6	220	data block	cr
6	220	data block	cr
6	220	data block	cr

9 rows selected.

Whoa! There are 9 buffers now. Block 220 now has 6 buffers - up from 4 earlier. This was merely a select statement, which, by definition does not change data. Why did Oracle create a buffer for that?

Again, the answer is CR processing. The CR processing creates copies of the buffer and rolls them back or forward to create the CR copy as of the correct SCN number. This created 2 additional CR copies. From one block, now you have 6 buffers and some buffers were created as a result of select statement. This how Oracle creates multiple versions of the same block in the buffer cache.

LATCHES

Now that you know how many buffers can be created and how they are located on the chains in the buffer cache, consider examine another problem. What happens when two sessions want to access the buffer cache? There could be several possibilities:

- 1) Both processes could be after the same buffer
- 2) The processes are after different buffers but the buffers are on the same chain
- 3) The buffers are on different chains

Possibility #3 is not an issue; but #2 will be. We don't allow two processes to walk the chain at the same time. So there needs to be some sort of a mechanism that prevents other processes to perform an action when another process is doing it. This is enabled by a mechanism called a **latch**. A latch is a memory structure that processes compete to acquire. Whoever gets is said to "hold the latch"; all others must wait until the latch is available. In many respects it sounds like a lock. The purpose is the same—to provide exclusive access to a resource—but locks have queues. Several processes waiting for a lock will get it when the lock is released in the same sequence they started waiting. Latches, on the other hand, are not sequential. Whenever latches are available, every interested process jumps into the fray to capture it. Again, only one gets it; the others must wait. A process first performs a loop, for 2000 times to actively look for the availability of a latch. This is called *spinning*. After that the process sleeps for 1 ms and then retries. If not successful, it tries for 1 ms, 2 ms, 2 ms, 4 ms, 4 ms, etc. until the latch is obtained. The process is said to be *sleep* state in between.

So, latches are the mechanism for making sure no two processes are accessing the same chain. This latch is known as **cache buffers chains latch**. There is one parent CBC latch and several child CBC latches. However, latches consume memory and

CPU; so Oracle does not create as many child latches as there are chains. Instead a single latch may be used for two or more chains, as shown in Fig 3. The number of child latches is determined by the hidden parameter `_db_block_hash_latches`.

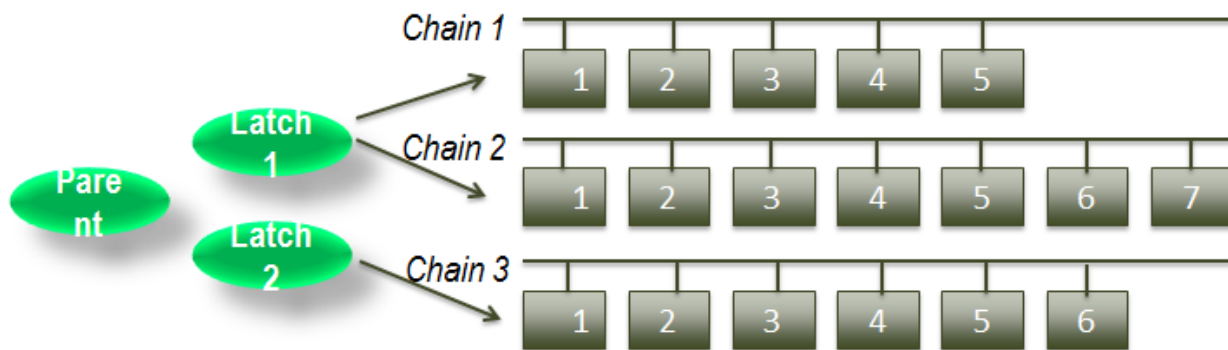


Figure 3 CBC Parent and Child Latches

Latches are identified by `latch#` and `child#` (in case of child latches). A specific instance of latch that is used is identified by its address in memory (latch address). To find out the latch that protects a specific buffer, get the `file#` and `block#` as shown earlier and issue this SQL:

```
select hladdr
from x$bh
where dbarfil = 6
and dbablk = 220;
```

Going back to CBC latches, let's see how you can find out the correlation between chains and latches. First, find the `Latch#` of the CBC latch. `Latvh#` may change from version to version or across platforms; so it's a good idea to check for it.

```
select latch# from v$latch
where name = 'cache buffers chains';
```

```
LATCH#
-----
      203
```

This is the parent latch. To find out the child latches (the ones that protect the chains), you should look into another view—`V$LATCH_CHILDREN`. To find out how many child latches are there:

```
SQL> select count(1) cnt from v$latch_children
where latch# = 203;
```

```
CNT
-----
 16384
```

If you check the values of the two hidden parameters explained earlier, you will see:

```
_db_block_hash_buckets      524288
_db_block_hash_latches      16384
```

The parameter `_db_block_hash_buckets` decides how many buffer chains are there and the parameter `_db_block_hash_latches` decides the number of CBC latches. Did you notice the value, 16384? It determines the number of CBC latches and we confirmed that it is in fact the number of CBC latches.

DIAGNOSIS OF CBC LATCH WAITS

Let's now jump into resolving the CBC latch issues. The sessions suffering from CBC latch waits will show up in V\$SESSION. Suppose one such session is SID 366. To find out the CBC latch, check the P1, P1RAW and P1TEXT values in V\$SESSION, as shown below:

```
select p1, p1raw, p1text
from v$session where sid = 366;
```

P1	P1RAW	P1TEXT
5553027696	000000014AFC7A70	address

P1TEXT clearly shows the description of the P1 column, i.e. the address of the latch. In this case the address is 000000014AFC7A70. We can check the name of the latch and examine how many times this latch has been requested by sessions but has been missed.

```
SQL> select gets, misses, sleeps, name
       2 from v$latch where addr = '000000014AFC7A70';
```

GETS	MISSES	SLEEPS	NAME
49081	14	10	cache buffers chains

From the output we confirm that this is a CBC latch. It has been acquired 49,081 times, 14 times missed and 10 times processes have gone to sleep waiting for it.

Next, identify the object whose buffer is so popular. Get the File# and Block# from the buffer cache where the CBC latch is the latch address we identified to be the problem:

```
select dbarfil, dbablk, tch
from x$bh
where hladdr = '000000014AFC7A70';
```

DBARFIL	DBABLK	TCH
6	220	34523

The TCH column shows the touch count, i.e. how many times the buffer has been accessed—a measure of its popularity and hence how much likely that it will be subject to CBC latch waits. From the file# and block# we can get the object ID. The easiest way is to dump the block and get the object ID from the dump file. Here is how you dump the above mentioned block.

```
alter system dump datafile 6 block min 220 block max 220;
```

This produces a tracefile, a part of which is shown below.

```
Start dump data blocks tsn: 4 file#:6 minblk 220 maxblk 220
Block dump from cache:
Dump of buffer cache at level 4 for pdb=0 tsn=4 rdba=25166044
BH (0x7ff72f6b918) file#: 6 rdba: 0x018000dc (6/220) class: 1 ba: 0x7ff7212a000
  set: 12 pool: 3 bsz: 8192 bsi: 0 sflg: 0 pwc: 39,28
  dbwrld: 0 obj: 93587 objn: 93587 tsn: [0/4] afn: 6 hint: f
```

Get the object ID (the value after “objn”). Using that value you can get the object name:

```
SQL> select object_name
      2 from dba_objects
      3 where object_id = 93587;
```

OBJECT_NAME

CBCTEST

Now you know the table whose blocks are so highly popular resulting in CBC latches.

RESOLVING CBC LATCH WAITS

From the above discussion you would have made one important observation—CBC latch waits are caused by popularity of the blocks by different processes. If you reduce the popularity, you reduce the chances that two processes will wait for the same buffer. Note: you can’t completely eliminate the waits; you can only reduce it. To reduce is, reduce logical I/O. For instance, Nested Loops revisit the same object several times causing the buffers to be accessed multiple times. If you rewrite the query to avoid NLs, you will significantly reduce the chance that one process will wait for the CBC latch.

Similarly if you write a query that accesses the blocks from a table several times, you will see the blocks getting too popular as well. Here is an example of such a code:

```
for i in 1..100000 loop
  select ...
  into l_var
  from tablea
  where ...;
  exit when sql%notfound;
end loop;
```

You can rewrite the code by selecting the data from the table into a collection using bulk collect and then selecting from that collection rather than from the table. The SQL_ID column of the V\$SESSION will show you which SQLs are causing the CBC latch wait and getting to Object shows you which specific object in that query is causing the problem, allowing you to devise a better solution.

You can also proactively look for objects contributing to the CBC latch wait in the Active Session History, as shown below:

```
select p1raw, count(*)
from v$active_session_history
where sample_time < sysdate - 1/24
and event = 'latch: cache buffers chain'
group by event, p1
order by 3 desc;
```

The P1RAW value shows the latch address, using which you can easily find the file# and block#:

```
select o.name, bh.dbarfil, bh.dbablk, bh.tch
from x$bh bh, sys.obj$ o
where tch > 0
and hladdr='<p1raw value>'
and o.obj#=bh.obj
order by tch;
```

With the approach shown earlier, you can now get the object information from the file# and block#. Once you know the objects contributing to the CBC latch waits, you can reduce the waits by reducing the number of times the latch is requested. That is something you can do by making the blocks of the table less popular. The less the number of rows in a block, the less popular the block will be. You can reduce the number of rows in a block by increasing PCTFREE or using ALTER TABLE ... MINIMIZE RECORDS_PER_BLOCK. If that does not help, you can partition a table. That forces the data block address to be recomputed for each partition, making it more likely that the buffers will end up in different buffer chains and hence the competition for the same chain will be less.

CONCLUSION

In this paper you learned how Oracle manages the buffer cache and how latches are used to ensure only one process can walk the chain to access a buffer. This latch is known as Cache Buffer Chain latch. You learned why this latch is obtained and how to reduce the possibility that two processes will want the latch at the same time.