Database Writer and Buffer Management

Nitin Vengurlekar
Field Support Specialist
Oracle Corporation

May 18,1998

Database Writer and Buffer Management

I. **Cache Buffer Management**
II. **Multiple DBWR s and Async I/O**
III. **Cache Management Process Flow**
IV. **Buffer Cache Management in O8**
V. **DBWR in Oracle8**
VI. **Appendix**

I. *Cache Buffer Management*

The Database Writer (DBWR) is one of the four minimum background processes required to initialize and run an Oracle instance. An Oracle instance collectively refers to the all background processes and the shared memory that is allocated on behalf of these processes. DBWR is generally Oracle process id 3 (v$process.pid) and starts after PMON[1]. Upon initialization, DBWR will acquire a Media Recovery (MR) lock on each online datafile. Thusly, DBWR is considered to be the maintainer of the database files. DBWR is a server process whose main function is to manage the buffer cache by making buffers available when requested and "clean it" when dirty. This is all done in an effort to reduce physical reads and writes to disks. Note, in an OPS environment (shared disk), cache management becomes slightly complicated as each instance maintains its own cache structures and consequently must provide global cache coherency across nodes. Nevertheless, the mechanics of cache management are generally the same. DBWR (along with foreground processes and LCK processes, if using OPS) utilize the Cache Buffer Management strategy to manage the buffer cache. Cache buffer management is composed of three internal structures : *cache buffer chains* and two chain lists, the *LRUW list* (dirty list) and *LRU list*[2] .

*A.. **Cache buffer chain***

The Cache buffer chain consists of hash tables (or buckets) that maintain doubly-linked hash lists. These hash lists comprise buffer headers (see figure 1 on page 15). Note; hash lists do not contain the actual buffer blocks, but rather the buffer headers[3]. Hash buckets are allocated at instance start up time. The number of hash buckets is determined as *prime(db_block_buffers/4)* . Although, this value can be overridden by defining init.ora parameter _db_block_hash_buckets, it is not a recommended practice. To determine the number of hash buckets allocated to an instance (V7323):

```
SVRMGR> select value from v$parameter where name = 'db_block_buffers';
VALUE
----------------
200

SVRMGR> select kviival , kviidsc from x$kvii where indx = '4';
KVIIVAL    KVIIDSC
----------     ----------------------------------------------------------------
       53      number of hash queue latch structures
```

[1] Dbwr is Oracle session id (v$session.sid) 2 .
[2] Although the buffer cache is depicted as pool of buffers, its really a linear array.
[3] This document will use the term buffer blocks to refer to blocks within the db_block_buffer cache and also to distinguish itself from buffer headers.

Buffers get hashed to a particular hash table based on their DBA (data block address)[4]. There is one latch that manages each hash chain and is called the cache buffer chain latch. Foreground processes must obtain this latch before searching the hash chain list. This is done in order to prevent the chain from being manipulated whilst searching. Note, the hash table is keyed for searches by "<DBA,Block Class>", see Appendix A for a list of block classes.
As stated earlier the hash chain list holds buffer headers. A buffer header is an internal structure that succinctly describes a buffer block's properties. There is a one-to-one relation between a buffer header and a buffer block. Thus the number of buffer headers is always equal to the *db_block_buffer* and is shown in the following (V7323):

```
SVRMGR> select kvitval, kvitdsc from x$kvit where indx = '2';
KVITVAL    KVITDSC
----------      ----------------------------------------------------------------
       200       number buffer headers
```

Highlighted below is some of information contained in these buffer headers. Indicators within the parenthesis reflect the field descriptor that is defined in a buffer dump. Note, contents of the headers will vary depending on action performed against the buffer block. To produce a buffer header dump on a running database, use the following command : *'alter session set events 'immediate trace name buffers level 10'*.

- Buffer status (st) : xcurrent, CR, reading.
- Lock mode (md) : exclusive (excl), shared (shr) , and null (null)
- In cache data block summary : dba, inc & seq, version and block type
- Buffer holder and waiters list
- Reference to the buffer's location on the hash chain, a pointer location of the buffer header on either the LRU or LRUW list and the pointer to the actual buffer block (ba). Note, buffers blocks can be on either the LRU or LRUW lists, but not both.
- Flags (flags) indicating operation to be perform or performed on the buffer block and recovery information.

B. LRU and LRUW lists

Least Recently (LRU) used is generally considered a discipline or policy to manage a set of equally or unequally weighted objects. Oracle implements this LRU policy against the database block buffers within the buffer cache. However, Oracle also uses the term LRU to refer to the two physical lists that makeup the LRU mechanism. These two separate lists that are called the LRU and LRUW linked lists, basically hold the buffer headers. Both the lists have different properties and thus are treated differently by DBWR.

1. *LRU list.*
The head of the LRU list is considered to be the hottest part of list; ie, it contains the MRU (most recently used) buffers. All new block gets are placed on the MRU end (with the exception of sequentially scanned blocks). The tail end of the LRU contains the buffers that have not been referenced recently and thus can be reused. Therefore the tail of LRU is where the foreground processes begin to search for free buffers.

Buffers on the LRU can have one of three statuses; free, pinned, or dirty. Pinned buffers are buffers currently being held by a user and/or have waiters against them. Moreover, the *pinned* status will be sub-categorized as *pinned clean* or *pinned dirty*. Free buffers are unused buffers; i.e., a new block that is to be

[4] A DBA, data block address, uniquely identifies a particular block id within a database file. Thus, the DBA is a 32 byte address that composes of a 22 byte block# and 10 byte file#.

read into the cache (from disk) can use it. The dirty buffers are modified buffers that have not been moved over to the LRUW cache. Dirty buffers are different from pinned dirty buffers, in that pinned dirties have user/waiters against them and hence cannot be written out to disk; whereas, the dirty buffers are freed buffers and can move to the LRUW list and subsequently to disk.

> As stated above all new buffer gets are assigned to the head of the LRU (the MRU). The only exception to this case occurs when new buffer gets come from blocks read through full table scans. Blocks gotten in this mode are placed at the tail end of the LRU list and limited to db_block_multi_read_count worth of buffers. This isolation prevents single data accesses from flushing out "hot" blocks from the cache and over-flooding the buffer pool. Moreover, full table scanned blocks are also considered least likely to be re-accessed; hence, it makes practical sense to place these blocks on the LRU end so they can be readily replaced.
> Having stated that, this default behavior for full table scans can be altered using the "CACHE" clause. Which is specified during table creation or alteration. Full table scan gets from tables with this attribute set on, are stored on the MRU end of the LRU list. Nevertheless, the use of the CACHE segment option does not preclude buffers from being aged out of the buffer cache; i.e., the CACHE option does not guarantee a table will be pinned in the buffer pool, it merely defers the aging out.
> Oracle will also cache full table scanned blocks at the MRU end if they are determined to be small tables. Oracle employs (prior to 7.3) the init.ora parameter small_table_threshold to determine whether a table is considered to be a small table. By default this parameter is initialized to max(4,db_block_buffers/50)
> Thus, a table is considered to be a small table if the total number of blocks does not exceed 2% of the buffer cache size. For example, if the buffer cache has 1000 blocks (db_block_buffers=1000) then a small table must have 20 or less blocks. Full table scanned block gets against this tables smaller than 20 blocks are placed at the MRU end of the list. The CACHE clause must be used with discretion, since it may cause inadvertent physical I/O for blocks that need to be in cache.

2. *LRUW list.*

The LRUW list contains the dirty buffers eligible for disk write-outs by DBWR. The LRUW list is also called the dirty list. How buffers get moved over to the LRUW list and consequently to disk is the foundation of DBWR's function and is illustrated below. DBWR writes buffer blocks to disk when it is signaled to do so. There are three events in which this happens, and the signals that trigger these events are shown by the following (V7323) query:

```
SVRMGR> select * from x$messages where indx in ('9','10','11');
ADDR       INDX    DESCRIPTION
--------   ----------  ------------------------------------------------------------
009F6ED0        9   write dirty buffers when idle - timeout action
009F6EE0       10   write dirty buffers/find clean buffers
009F6EF0       11   write checkpoint-needed buffers/recovery end
```

a. *DBWR write dirty buffers/find clean buffers* (indx value 10).

When a foreground process reads a database block from disk , the block must be read into the buffer cache as well[5]. However, in order to read it into cache, a free buffer must exist. To find a free buffer, the foreground process must lock and search the LRU list for a free buffer, starting from the tail of the LRU. Dirty buffers that are detected along the way are moved over to the LRUW list; in addition, the *dirty buffers inspected* and *free buffers inspected* statistics[6] are incremented. If a free buffer is not found within the threshold limit, referred to as the *foreground scan depth* (influenced by init.ora _db_block_max_scan_count) , then the search is halted, an internal structure within the SGA (variable area) is generated w/ a flag that messages DBWR and the LRU latch is released. This message will signal DBWR to perform a large batch write to make clean buffers available at the tail of LRU, whilst the

---

[5] Note, only foreground processes will read blocks into cache, and only Dbwr will write buffers to disk.

[6] During the LRU search, if a pinned buffer is detected, it is not moved over to the LRUW list, but the *free buffer inspected statistic* is incremented.

foreground process waits on the *free buffer* wait event. DBWR will acquire the LRU latch and scan the LRU list, gathering dirty buffers to write out. This DBWR scan amount is referred to as the DBWR scan depth and will be discussed later.

In addition to the aforementioned scenario, if the foreground process detects a dirty buffer in the LRU and upon moving it to the LRUW list, it might ascertain that the LRUW list is full. This is a upper bound limit, defined as max dirty queue (dictated by 2*_db_block_write_batch or _db_large_dirty_queue). At this point the LRUW list will not accept anymore dirty buffers. DBWR is then signaled to clean out the cache with the same size large batch write. In this situation, DBWR is considered to be in a panic state and will put complete emphasis on cleaning up the LRUW and LRU lists. Foregrounds in this state will be blocked from accessing the LRU list as this will prevent further dirties and scans from occurring. This situation is similar to most operating systems during a demand paging state, where the minimum number of free memory pages falls below the minfree amount.

> DBWR performs batch writes or "IO clumps" to disk. These write sizes are generally up to *_db_block_write_batch (*init.ora parameter); however in a well-tuned system these sizes maybe smaller. After Oracle 7.2, the _db_block_write_batch parameter is automatically set to 0 by default, and Oracle dynamically determines the new value via
>
> min(½ * db_file_ simultaneous_writes * db_files, max_batch_size, buffers / 4).
>
> The _db_block_write_batch parameter, also known as *max write batch*, influences the behavior of many other DBWR's functions, such as *Dbwr buffer scan depth*. Therefore, the recommendation is not to alter this parameter. The following query displays the dynamically determined _db_block_write_batch parameter size (V7323).

```
SVRMGR> select kviival from x$kvii where kviidsc = 'DB writer IO clump';
            KVIIVAL
            ----------
                  40
```

b. *DBWR write dirty buffers when idle* (indx value 9)
DBWR is set to timeout after three seconds of inactivity[7]. Each timeout will awaken DBWR to traverse through the buffer headers (scan size equals 2*_db_block_write_batch) to find and write out any current or dirty blocks (temporary, a.k.a. sort blocks, are skipped) . If there are any buffers in the dirty list, then this is also considered non-idle activity. This prevents DBWR from being too idle.

c. *DBWR write checkpoint -needed buffers/recovery end* (indx value 11)
When a checkpoint occurs (either through a threshold value set in init.ora or an explicit alter system command) LGWR [8] will signal DBWR with a buffer header array (buffers with the checkpoint flag set on) of current, dirty and non-temporary buffers to write out to disk. The write size is dictated by the _db_block_checkpoint_batch parameter. Similar to the *DBWR write dirty buffers when idle* event, DBWR will write out a list of current and dirty buffers. However, checkpoint-ed buffers that are written out to disk are not marked as free; ie, they are retained in the cache. This allows for a better hit ratio, hence less physical I/O. There are two types of checkpoints slow and fast, these checkpoints differ only in the way DBWR paces itself. With slow checkpoints, DBWR will write out db_block_checkpoint_batch , then pauses for a determined amount of time, then waits for "make free requests" then writes another _db_block_checkpoint_batch until the checkpoint is complete. The checkpoint pause allows efficient use of CPU and disk bandwidth and also prevents DBWR from overusing the LRU latch. With fast checkpoints there are no pauses and moreover, DBWR will also write dirty buffers from the LRU list. Fast checkpoints generally occur when check-pointing falls behind LGWR. Check-pointing has changed significantly in Oracle8, please review the Oracle8 Concepts Manual for details.

---

[7] The Dbwr timeouts can be disabled via init.ora _db_block_no_idle_writes; however, this not recommended

[8] The CKPT process will signal DBWR if it is enabled

As foreground processes begin to use and dirty the buffer cache, the number of free buffers slowly decreases. This may translate into unnecessary physical I/O, since re-accessed blocks must now be gotten from disk instead of cache. To prevent the number of dirty buffers from growing too large, a variable is defined to evaluate the number of clean (not dirty or pinned) buffers on the LRU. This value, which is referred to as the "known clean buffers count", is basically indicative of how DBWR is managing the tail-end of LRU.[9] The known clean buffers count is incremented (by either foreground or background processes) each time a dirty buffer is moved to the LRUW and thereafter written to disk. Conversely, known clean buffer count decrements whenever a foreground process uses a free buffer. When the known clean buffer count value begins to diminish down to a threshold value, DBWR will be signaled to start clean-out of the LRU list, by scanning DBWR scan depth full of buffers. The DBWR scan depth is a self-adjusting variable that changes depending on how DBWR is keeping up and maintaining clean buffers on the tail of LRU. For example, if foreground processes detect and move dirty buffers to the LRUW list, then DBWR knows its not keeping up, since foregrounds are having to move buffers on its behalf. Also, if the known clean buffer count is below ½ the DBWR scan depth, then DBWR is not maintaining enough clean buffers. In both cases , the DBWR scan depth will increment to an amount based on scan depth increment size and not to exceed the max scan depth size. Conversely, the DBWR scan depth decrements when the known clean buffer count is greater than ¾ the DBWR scan depth and the LRUW list is empty. Here, the scan depth decrement is subtracted from the DBWR scan depth.
The following query shows the scan increment and decrements (V7323):

```
SVRMGR> select kvitval, kvitdsc from x$kvit where indx in (7,8);
KVITVAL    KVITDSC
----------       ----------------------------------------------------------------
         5        DBWR scan depth increment
         1        DBWR scan depth decrement
```

In general, when DBWR is signaled to clean-out the dirty list, it will always gather _db_block_write_batch full of buffers from the LRUW list, pinning[10] buffers its going to write along the way. Thereupon, if it has not filled up _db_block_write_batch full of buffers, DBWR will then scan the LRU list for more dirty buffers (note; these are dirty buffers that have not been moved over to LRUW). As soon as DBWR has completed its write, it will post a *write complete* acknowledgment and un-pin all the cache buffers that were written out. If a foreground process has to wait to re-read a particular block (that was on the LRUW) because it is being written out, then the *write complete wait* event statistic is incremented. Moreover, when a foreground process has to wait for a free buffer because the dirty list is full, and subsequently because DBWR had not completed its write request, then the *free buffer waits* statistic is incremented. However, these hardships are somewhat alleviated in Oracle 7.2 as buffers get unpinned when that particular buffer has been written out; whereas, in pre-Oracle7.2, the entire write request had to be complete before any buffers were un-pinned. Once the dirty buffers are written out to disk, these buffers are now considered clean and placed on the tail end of LRU, replenishing the LRU clean buffers.

DBWR performs writes equaling _db_block_write_batch. However, after performing this write, DBWR will discover that there are new dirty buffers on the queue again. The number of new dirty buffers on this list are collectively referred to as the *summed dirty queue length*. Therefore the summed dirty queue length is defined as the size of dirty queue after the successful completion of a write batch request. If this queue is larger than the write batch, then the dirty list is getting dirty too fast and DBWR can not keep up. The average queue length is equal to *summed dirty queue length/write requests.* This value indicates the average size of the dirty buffer write queue after a write request by DBWR . This queue length should be

[9] The known clean buffers value is an merely an approximation, since this value is constantly changing.
[10] The term pinning is loosely used here, since Dbwr does not actually pin the buffer, but rather sets a flag in the header that indicates its about to be written out.

compared to the init.ora parameter _db_block_write_batch, if the queue length is 2 times larger [11], then there may be a need to examine DBWR effectiveness to write. If the system is not already CPU or IO bound, then DBWR can be influenced to write in large batches, to alleviate the queue length issue. The parameter _db_block_write_batch (discussed in Section I) dictates the write size (write batch size) for DBWR and can be induced by altering the value for init.ora, db_files_simultaneous_writes. A larger size will mean a larger write and thus less signals for DBWR to perform writes; however, making this parameter too large will cause DBWR to be I/O bound and skewed between datafiles.

II. *Cache Management Process flow*

A.

Having discussed how DBWR manages the LRU/LRUW lists, the next logical step is to review the foreground processes view of the Cache Buffer Management.

In general, when a user wants to read a particular block, the foreground process will flow through the following scenarios:

1. Obtain a cache buffer chain latch and the search the cache buffer chain list for the DBA with an associated SCN.
2. If the DBA is found, then it pins the buffer block and reads it. However, if the buffer block is dirty and thus the buffer scn is > requesting scn, then a CR operation must be performed. The statistics *db block gets* or *consistent gets* are incremented (depending on type of Sql call). This is considered a logical hit.
3. If the DBA does not exist in the cache buffer chain, then it does not exist on the LRU chain. Therefore it must be read in from disk. In this case, the LRU latch is first obtained and the LRU list is searched to find a free buffer. If a free buffer is not found within the search limit, then several events are triggered, as shown by item (a) of Section B above. If a free buffer is found, then the buffer is pinned and moved to the head of LRU, the block is read in from disk and cache buffer chain latch is acquired to update the hash list with the new buffer header information (corresponding to the new buffer block). If a buffer is found to be unpinned and non dirty, then it is a prime candidate ("victim") to be replaced with the block from disk[12].

It can be easily seen that reducing buffer operations will be a direct benefit to DBWR and also help overall database performance. Buffer operations can be reduced by (1) using dedicated temporary tablespaces, (2) direct sort reads, (3) direct Sqlloads and (4) performing direct exports. In addition, keeping a high buffer hit ratio will be extremely beneficial not only to the response time of application, but the DBWR as well. This is evident when realizing how much of the code path is bypassed with a logical hit of a buffer.

B. Multiple LRU latches

As shown in the example above, the LRU latch is acquired, by foreground and DBWR, for every buffer movement and scanning. In some cases, due to heavy loads and excessive LRU scanning, the LRU latch may become a bottleneck. For example, enabling the db_lru_extended_statistics (via the init.ora parameter) can be one of causes of excessive contention against the LRU latch. Thus, it is not prudent

---

[11] An alternative derivation is to determine if the average queue length is larger than min ((db_files * db_file_simultaneous_writes)/2), (¼ * db_block_buffers)). If so, then increase either db_files or db_file_simultaneous_writes. As noted earlier, it is not recommended to alter the value of _db_block_write_batch.

[12] Note, buffers that are unpinned and non dirty do not have users/waiters against them and do not have to be moved to LRUW, thus these buffers are simply replaced on the LRU chain.

to specify this in a production environment. Moreover, the LRU latch does not scale very well on SMP systems, since multiple CPUs may try to acquire the single LRU latch. To assuage the serialization against the LRU latch , Oracle V7.3 introduced multiple LRU latches via the init.ora parameter db_block_lru_latches. This parameter is generally set to the number of CPUs. Each LRU latch is referred to as a system set. Collectively, the sets can be thought of as small assemblage of the single-whole LRU/LRUW lists. Each system set will have its own LRU and LRUW lists, and thus will manage its own set of buffer blocks. Buffers are assigned to the system sets in a round-robin fashion to balance the buffers-to-LRU. Thus, a buffer block will only be associated with one system set. Before a foreground searches for free buffers it is assigned a to set, if a latch is unable to be acquired, then the next set is pursued. This is performed until a successful latch is gotten[13]. In the event a latch could not be acquired, the statistic *chain buffer LRU chain* under latch misses is incremented. When a foreground process acquires a LRU latch, it will only scan the blocks on its assigned system set LRU list, and only move dirty buffers to its system set's LRUW list.
Although the advent of multiple LRU latches/sets provides a buffer-LRU isolation, DBWR will still monitor all the system sets; scanning the sets for dirty buffers. The scan depths and write batch sizes are localized, rolled and collected as a whole. For example, if DBWR is to search for dirty buffers, it still acquires a latch and scans the LRU. If the scan still has not fulfilled its write batch size, then it will acquire the next latch and gather buffers on that LRU list. This is done until all the sets are scanned or the batch size is filled.

III. *Multiple DBWR s and Async I/O.*

As discussed earlier DBWR's main function is to keep the buffer cache clean. This function is partially dependent upon how fast DBWR can write out dirty buffer blocks. There are two mechanisms that will aid DBWR in this arena: DBWR slaves processes and asynchronous I/O operations.

a. Multiple DBWR s

The number of DBWR slave processes is dictated by the init.ora parameter *db_writers.* The general rule of thumb in setting an appropriate value for the db_writers, is *mean (average # of disks that span a typical file, 2* #CPUs).* However, if the system is already I/O bound then it may be appropriate to set this value as

min(average # of disks that span a typical file, 2 #CPUs).*

The maximum value is 50 (Platform dependent). Slave DBWR processes (also called detached processes) provide parallel write processing and I/O distribution. The DBWR slave processes startup after SMON has been initialized and each slave process started will consume approximately 3400 bytes in the variable portion of the SGA. Once the slave processes are initialized they usually wait on the 'Slave DBWR timer' event (v$session_event.event) when waiting for data to be handed-down from the master DBWR. Conversely, the master DBWR will also wait on the 'DBWR I/O to slave' event when awaiting acknowledgment for the I/O completion from the slave processes.[14]

DBWR slave processing allows the main DBWR process to offload the I/O writes to each of the slave processes. When the master DBWR gets posted with a batch write, the I/O request is managed by the master DBWR process, which then hands off blocks to write out to each slave process, in a round-robin fashion. This is done until the entire I/O request is handed off. The master DBWR process does not participate in the physical I/O, its function is merely as an overseer. Moreover, each slave may not

[13] The LRU latch is gotten in nowait mode, if unsuccessful after trying all the sets, then latch is gotten in wait mode (possibly against the original set).
[14] The two DBWR events were in effect after 7.2.x, however, several ports (such as HP) had not implemented this change to ssfao.c thus still used the old events, which where 'Null event'.

participate in each I/O request, this is dependent upon the size of the write and the current activity of the slaves.
The master DBWR, upon initialization, will allocate a structure within the variable portion of the SGA area. This structure maintains the # of db_writers, a file identification structure, which describes the datafile info), and a file status structure .
When multiple DBWR s are started, each slave process will allocate a file status structure within their PGA.. The contents of this structure is populated by copying the file status data from the master DBWR's SGA structure. The master DBWR will also create another structure within the SGA structure to communicate all write/close operations to the slave processes. There is one structure for each slave process. Note, this structure is allocated in the SGA and the slave's PGA.. Therefore when DBWR receives a write request, it searches the array for idle slaves to process the I/O request. Once idle slaves are found, the master DBWR copies the I/O request into the slaves' (PGA) structure. The I/O request comprises the file#, block# and the file identification structure. After the request is initialized, the master DBWR sets the write pending I/O flag and posts the slave(s). The slave process will validate the passed file identification structure information with its own copy in the PGA. This validation is performed to make sure the state of datafile has not been changed; such as dropped or offlined[15].
If validation is successful then the I/O is processed Upon completion of the I/O, the slave process will post the completion by turning off the write pending flag in the SGA and mark itself . Note, if the db_writers parameter is set to 1, then slave processing is automatically disabled and thus db_writers is reset to 0. Therefore db_writers must be set greater than 1 to enable this feature. In versions 7.2.x and earlier, multiple DBWR s feature was automatically disabled if the number of open datafiles exceeded 100. This is a hard limit within Oracle. If this datafile limit is exceeded than master DBWR will assume responsibility of all I/O operations and bypass the slave processes[16]. Multiple db_writers are implemented differently in Oracle8, review Section IV for the new changes.

b. Asynchronous I/O.

Asynchronous I/O operations allow DBWR to perform parallel write processing as well as non-blocking I/O[17]. However, this is only possible if async I/O is available on the dependent operating system. Generally operating systems will restrict async I/O to only raw devices; whereas others will support raw and filesystems (cooked). On AIX, async I/O is automatically turned on; whereas other operating systems require a device driver configuration, followed by a kernel rebuild. To enable Oracle to use async I/O , the init.ora parameter (parameter varies between platforms) must be set to true[18]. Certain operating systems have implemented independent async write and async read structures. For example, Oracle for Sun Solaris requires two separate configurable init.ora parameters for async I/O processing : async_read and async_write. In general the two dependent should be set to the same values. On AIX there is a unified async I/O configuration, therefore there is a single async I/O init.ora parameter; use_async.
If Oracle has async I/O enabled, then aioread and aiowrite (via libaio functions) will be issued as opposed to read (pread) or write (pwrite) calls. Note, AIO calls result in calls to KAIO (kernel asynchronous I/O). KAIO supports raw (non-buffered) devices only. For example, on Oracle-Solaris based systems, if the AIO call is for a filesystem, the KAIO call will fail; however, the I/O is still serviced. Once DBWR has queued and started (submitted) an AIO I/O request, DBWR is free to do other work, but may not issue any more AIO calls. The outstanding I/O will be polled (using poll()) to determine status and evaluate any errors.

---

[15] However, if a datafile resize operation occurs then this "update does not reach the slaves, who are looking at an stale version of this structure". This will cause the slave process to issue a write call against a file-block# that is out of its known range, and thus incur a KCF write/open error (bug 311905)
[16] The db_writer code is PL specific, thus different ports might implement it differently. This document was written based on the Sun Solaris port.
[17] There have been several documented cases where problems arose due to implementing async I/O and multiple DBWR s.
[18] Oracle 8 use the parameter init.ora *disk_async_io* .

To determine if async I/O is enabled (note, this is OS level check):

On AIX
lsdev -Ccaio <--------------- Aio is enabled If output of this command indicates that device "AIO" is "available"
On Pyramid:
strings /unix | grep specaio
specaio Y 1 0 0 0 0 0 0 0 --> "Y" - AIO enabled or "N" - AIO not enabled
ddmp_cspecaio

IV. ***Buffer Cache Management in Oracle8.***

Oracle 8 has introduced a new feature to the buffer cache area, called multiple (or partitioned) buffer pools. This feature provides the capability to configure three separate buffer pools. These new entities include "keep", "recycle" , and default buffer pools. The three buffer pools are carved out of the db_block_buffers cache. The keep buffer pool is a pool of buffers that will be used to hold hot blocks (such as indexes and data) that need to kept in the cache as long as possible. Whereas, the recycle cache will house the transient data such as temporary table blocks, full table scanned blocks, or blocks that will not be re-accessed again. The default pool is the remaining portion left from the db_block_buffer cache after the keep and recycle have been allocated. The main advantage of the keep and recycle buffer pools, also known as the *subcaches,* is the segregation that is provided by isolating the buffer blocks by its usage; ie, reducing the data access interference. As a result, this will allow hot blocks to remain in the cache longer, providing extremely high ratios.
In versions prior to Oracle8, there was a single unified buffer cache, which housed all the segment blocks; hence it was subject to data access interference. However, Oracle7 simulated some characteristics of Oracle8 cache management. For example :

- data access interference was provided by preventing large data gets (full table scans) from over-flooding the buffer cache by placing block gets on the tail of the LRU and the number of blocks from this access were limited to db_block_multi_read_count.
- In V7.3, a table segment could be cached entirely in the pool, using the CACHE table attribute.
- Also in V7.3, multiple LRU latches were introduced to simulate "mini" subpools.

Nevertheless, these V7 mechanisms, still did not provide complete data isolation as furnished by the V8 multiple buffer pools.

With the advent of this buffer cache change comes the addition of a new init.ora parameter; buffer_pool_*name,* where name is either keep or recycle. The buffer_pool_*name* parameter will include the size of pool (unit is # of buffers) and the number of latches. Listed below is a sample of the init.ora parameters that illustrate new buffer pool entities.

```
db_block_buffers = 1000
db_block_lru_latches = 6
buffer_pool_keep=("buffers:400","lru_latches:3")
buffer_pool_recycle=("buffers:50","lru_latches:1")
```

To display how cache management has orchestrated the setup of multiple buffer pools, use the following query:

```
select name, set_count, lo_bnum, hi_bnum, buffers from v$buffer_pool;
NAME          SET_COUNT   LO_BNUM   HI_BNUM      BUFFERS
------------  -------- ----------  ---------------  --------------  --------------
                      0          0         0            0
KEEP                  3          0       399          400
```

RECYCLE	1	400	449	50
DEFAULT	2	450	3999	3550

The set_count is the number of system sets (LRU latches) assigned to each pool. The lo_bnum and hi_bnum are buffer number ranges within the cache buffer chain. Thus, the difference between the lo_bnum and hi_bnum is the number of buffers assigned to the pool. In the above example, the keep pool will contain three system sets each comprising of (approximately) 133 buffers, the recycle has 1 system set which has 50 buffers and the default pool has the remaining 2 sets (db_lru_latchs - (keep latches + keep latches)) which manages 1775 buffers. Note, the maximum number of system sets is dictated by *min(n*2*3, db_block_buffers/50)* where n is the number of CPUs[19]. Therefore, if the db_block_lru_latches parm is not set accordingly, it will be reset to this value. Moreover, each system set requires a minimum of 50 buffers.
Segments are assigned to either buffer pool at segment creation (or modified via the alter command). Moreover, each V8 partition table partition can be uniquely assigned. Described below is a the sample create table syntax. After table creation all subsequent block access to this segment will be cached and managed in the appropriate buffer pool. Note, each of the buffer pools are still subject to the same LRU algorithm as in a single unified buffer pool. Catalog view *dba_segments* will indicate the subcache setting via the *buffer_pool* column (or seg$.cachehint).

```
create table emp
   (empno  number(5)
     deptno              number(5)
     name                varchar(50) )
 tablespace test storage (initial 1M next 1M minextents 2 buffer_pool keep) ;
```

```
create table dept
   (empno number(5)
     deptno              number(5)
     name                varchar(50) )
 tablespace test storage (initial 1M next 1M minextents 2 buffer_pool recycle) ;
```

Therefore, all data accessed from the emp table will be held in the keep buffer pool; whereas, dept table will be housed in recycle pool.
Once segments have been assigned to the appropriate pools, various statistics such as logical hit ratio or free buffer waits, can be produced. The view that contains these statistics are in v$buffer_pool_statistics. This view must be created via $ORACLE_HOME/rdbms/admin/catperf.sql. The expected logical hit ratio should be higher than a single-large buffer pool. Tuning the keep and recycle buffer pools should be performed similarly to a single cache pool; ie, monitor free buffer waits, write complete waits, and buffer busy waits. Currently utlbstat/utlestat does not reflect the new subcaches.

[19] The formula, n*2*3, was chosen because there should be (theoretically) two LRU latches per CPU and there were 3 subcaches.

Illustrated below is a describe of the *v$buffer_pool_statistics*

```
desc v$buffer_pool_statistics
 Name                           Null?    Type
 ------------------------------ -------- ----
 ID                                      NUMBER
 NAME                                    VARCHAR2(20)
 SET_MSIZE                               NUMBER
 CNUM_REPL                               NUMBER
 CNUM_WRITE                              NUMBER
 CNUM_SET                                NUMBER
 BUF_GOT                                 NUMBER
 SUM_WRITE                               NUMBER
 SUM_SCAN                                NUMBER
 FREE_BUFFER_WAIT                        NUMBER
 WRITE_COMPLETE_WAIT                     NUMBER
 BUFFER_BUSY_WAIT                        NUMBER
 FREE_BUFFER_INSPECTED                   NUMBER
 DIRTY_BUFFERS_INSPECTED                 NUMBER
 DB_BLOCK_CHANGE                         NUMBER
 DB_BLOCK_GETS                           NUMBER
 CONSISTENT_GETS                         NUMBER
 PHYSICAL_READS                          NUMBER
 PHYSICAL_WRITES                         NUMBER
```

V. *DBWR in Oracle8*

In Oracle7 DBWR could only perform asynchronous I/O if the platform supported the function calls. If the platform did not support this feature, then the alternative was to use multiple database writers (db_writers). As discussed in the earlier section, multiple db_writers was used to simulate async I/O by way of master-slave processing. In Oracle8, two new approaches have been implemented to allow greater I/O throughput for DBWR write processing[20]. Note, these two implementations are mutually exclusive.

a. DBWR IO slaves
   In Oracle7, the multiple DBWR processes were simple slave processes; i.e., unable to perform async I/O calls. In Oracle803, the slave database writer code has now been kernalized, and true asynchronous I/O is provided to the slave processes, if available. This feature is implemented via the init.ora parameter *dbwr_io_slaves*. With dbwr_io_slaves, there is still a master DBWR process and its slave processes. . This feature is very similar to the db_writers in Oracle7, except the IO slaves are now capable of asynchronous I/O on systems that provide native async I/O, thus allows for much better throughput as slaves are not blocked after the I/O call. Slave processes are started at the database open stage (not instance creation), and thus will probably be assigned process id 9 through x, where x is the number of slave processes. The names of the DBWR slave processes are different than the slaves of Oracle7. For example a typical DBWR slave background process maybe : *ora_**i103**_testdb*.
   Where **i** indicates that this process is a slave IO process.
   1 indicates the IO adapter number
   3 specifies the slave number

   Therefore if dbwr_io_slaves was set to 3 then the following slave processes will be created: ora_i101.testdb, ora_i102_testdb and ora_i103_testdb.

```
p97050 15304     1  0 08:37:00 ?        0:00 ora_i102_pmig1
```

[20] In Oracle8 disk asynchronous I/O can be enabled using the parameter disk_async_io.

```
p97050 15298     1  0 08:36:56 ?        0:00 ora_smon_pmig1
p97050 15296     1  0 08:36:56 ?        0:00 ora_ckpt_pmig1
p97050 15302     1  0 08:37:00 ?        0:00 ora_i101_pmig1
p97050 15292     1  0 08:36:55 ?        0:00 ora_dbw0_pmig1
p97050 15290     1  0 08:36:55 ?        0:00 ora_pmon_pmig1
p97050 15294     1  0 08:36:56 ?        0:00 ora_lgwr_pmig1
p97050 15306     1  0 08:37:01 ?        0:00 ora_i103_pmig1
```

b. Multiple DBWRs.
   Multiple database writers is implemented via the init.ora parameter *db_writer_processes*. This feature was enabled in Oracle8.0.4, and allows true database writers; i.e., no master-slave relationship. If *db_writer_processes* is implemented, then the writer processes will be started after PMON has initialized. The writer processes can be identified (OS level) by viewing ps command output . In this example db_writer_processes was set to 3. The sample ps output shows the following. Note, the DBWR processes are named starting from 0 and there is no master DBWR process; all are equally weighted.

```
p97050  1472     1  0 10:48:18 ?        0:00 ora_dbw2_pmig1
p97050  1474     1  0 10:48:18 ?        0:00 ora_dbw3_pmig1
p97050  1466     1  0 10:48:17 ?        0:00 ora_pmon_pmig1
p97050  1478     1  0 10:48:18 ?        0:00 ora_ckpt_pmig1
p97050  1470     1  0 10:48:18 ?        0:00 ora_dbw1_pmig1
p97050  1480     1  0 10:48:18 ?        0:00 ora_smon_pmig1
p97050  1468     1  0 10:48:18 ?        0:00 ora_dbw0_pmig1
p97050  1476     1  0 10:48:18 ?        0:00 ora_lgwr_pmig1
```

   With Oracle804 db_writer_processes , each writer process is assigned to a LRU latch set (discussed in Section III). Thus, it is recommended to set db_writer_processes equal the number of LRU latches (db_lru_latches) and not exceed the number of CPUs on the system. Thus if db_writer_processes was set to four and db_lru_latches=4, then each writer process will manage its corresponding set; i.e., each writer will write buffers from its appropriate LRUW list and asynchronously, if available,. Allowing each writer to manage at least one LRU latch provides a very autonomous and segregated approach to Cache management.

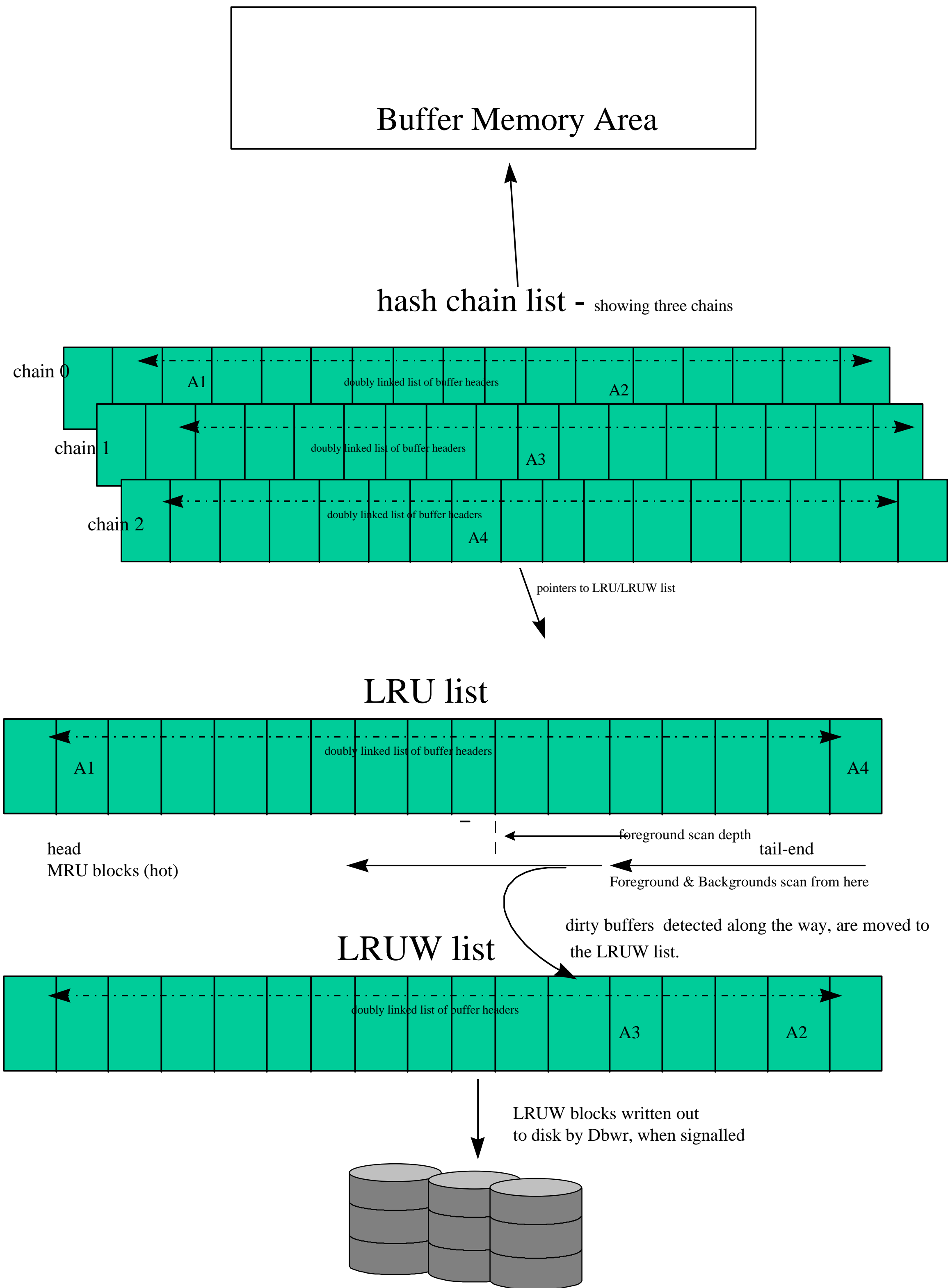
The main advantage to implementing Oracle8 DBWR I/O slaves (and DBWR processes), is the programmatic simplicity that is afforded. The DBWR slave IO code has now been kernalized, and thus is more generic. In the past the slave IO code was in the OSD layer, thus making it very port specific. Although both implementations of DBWR processes will be beneficial, the general indicator rule, on which option to use, depends on the availability of asynchronous I/O (from the OS) and the number of CPUs. Note, the number of CPUs is also indirectly related to the number LRU latch sets. The following is top down checklist approach to determine which option , if any, to implement.

- If async I/O is available, then use db_writer_processes ; Note, the number of writer processes should not exceed the number of CPUs.
- If async I/O is available; however, the system is a uniprocessor then implement dbwr_io_slaves. A uniprocessor will most likely have db_lru_latches set to 1.
- If async I/O not available and the system is a multiprocessor (then db_lru_latches can be set to the number of CPUs), then use db_writer_processes.

However, implementing db_io_slaves comes with some overhead cost. Enabling slave IO processes, requires that extra shared memory be allocated for IO buffers and request queues. Multiple writer

processes (and IO slaves) are advanced features, meant for heavy OLTP processing.   Implement this feature only if the database environment requires such IO throughput. For example, if async I/O is available, it may be prudent to disable I/O slaves and run with a single DBWR in async I/O mode. Review the current throughput and examine possible bottlenecks to determine if it is feasible to implement these features.

Figure 1



V. Appendix

A. *Block classes*

KCBCSVSH (Save Undo Segment Headers (block class 5))
KCBCSVUN (Save Undo Blocks (block class 3))
KCBCSORT (Sort Blocks (block class 2))
KCBCDATA (Data Blocks (block class 1))
KCBCSEGH (Segment Headers (block class 4))
KCBCFRLS (Free List Blocks (block class 6))
KCBCUSH (Undo Segment Header (block class 7 + (n*2)))
KCBCUSB (Undo Segment Block (block class 7 + (n*2) + 1))

Note: For undo (rollback) segments, 'n' is the undo segment number.
Block Class '0' is reserved for error detection in KCB.

B. *Listed below are derived, tunable, and informational statistics from a bstat/estat report that affect cache buffer management and/or DBWR performance.*

- **consistent gets** + **db_block_gets** = **logical reads**
  (logical_reads / **(logical_reads** + **physical reads**)) * 100 = **logical hitratio**
  This value should (generally) be 85-90% on a cooked filesystem and 90-95% for a raw devices[21];

- **free buffers inspected** - is the number of buffers foreground processes skipped in order to find a free buffer (this includes pinned and dirtied buffers) **.**
  dirty buffers inspected is equal to the numbers of buffers that foregrounds found dirty.
  free buffers inspected - dirty buffers inspected = the number of pinned buffers; ie, buffers that users/waiters against them.
  free buffers requested - is the number of times a free buffer was requested to create or load a block.
  For informational use only
  Free Buffer Scan Ratio = free buffer inspected / free buffers requested, and should be no more than 4%. If this is higher, then there are too many unusable buffers in the cache and thus DBWR is not keeping up. There may be a need to increase db_block_buffers or increase the number of db_writers.

- **Dbwrs free buffers found** - is the number free (available) buffers found by Dbwr when requested to search the LRU list.
  Dbwr make free requests - is the number of times Dbwr was invoked to free up db_block_buffers,
  Dbwr free buffers found / **Dbwr make free requests** = average number of reusable (available buffers for use) buffers per LRU scan request. This value should be at least *db_block_buffers/16*
  also,
  review **dirty buffers inspected**, which is the number of unusable (dirtied) buffers on the LRU. If this value is small and the average number of reusable buffers is high, then this indicates that Dbwr is performing efficiently and not falling behind.

- **summed dirty queue length** - numbers of buffers pending for writing; i.e., the length of LRUW after the write request finished.
  summed dirty queue length/write requests = avg. length of dirty list. See Section D for tuning.

- **Dbwr buffers scanned** - total number of buffers scanned in the LRU chain to make clean.

---

[21] The logical hit ratio for heavy OLTP systems may be artificially higher due to the influence of index range scans on logical I/Os.

(this includes dirty, pinned and clean buffers)

- **Dbwr checkpoints** - is the number of times Dbwr was signaled to perform a checkpoint. This includes mini-checkpoints (caused by DDL) , datafile checkpoints and log_interval_* induced checkpoints. Note, however, this not indicate the total checkpoints completed, since two or more checkpoints can union-ed into one checkpoint. Nevertheless, the values for *background checkpoints started* and *background checkpoints completed* should only vary slightly..
  For informational use only

- **cluster Key Scan Block Gets** - number of cluster blocks accessed.
  cluster Key Scans - number of scans processed on cluster blocks.
  cluster key scan block gets/cluster scans = degree of cluster key chaining. If this value is > 1, then you have some chaining.

- **sorts(disks)** - the total number of disks sorts; i.e., this is the number of times sorts could not be done in memory.
  sorts(memory) - the total number of sorts in memory, as specified by amount **sort_area_size.**
  sorts(disks)/[sorts (disk) + sorts (memory)] * 100 = percentage of sorts done on disk, should be less than 10%, if high possibly increase *sort_area_size*.

- **Chained row ratio -** describes the ratio of chained or migrated rows.
  table fetch continued row - the number of chained rows detected.
  Chained row ratio = table fetch continued row / (table fetch by rowid + table scan rows gotten).
  This value should be close to 0 as possible. If this value seems high possibly increase PCTFREE or if plausible, increase db_block_size. If the database contains long datatypes, then the chained row ratio may be misleading.

- **Index usage ratio** - indicates the degree of index usage with respect to full table scans and determined using the following :
  table fetch by rowid / (table fetch by rowid + table scan rows gotten)
  table fetch by rowid - the number of logical rows fetched from a table by rowid (either using indexes or 'rowid=').

- **table scan blocks gotten** and **table scan rows gotten** are respectively, the total number of blocks and rows fetched during full table scans to determine the average number of rows gotten per block for full table scans is:
  table scan rows gotten / table scan blocks gotten **-** this should be a high number for DSS systems.
  Table Scans (short) * 5 blocks = Blocks Scanned (short)
  Table Scan Blocks Gotten - Blocks Scanned (short) = Blocks Scanned (long)
  Blocks Scanned (long) / Table Scans (long tables) = Average number of blocks scanned per long table. This value should be high.

- **table scans (long tables)** - the number of full tablescan performed on tables with 5 or more blocks. This number under the "per trans" column should not be greater than 0. If this is, than you need to review the application to use indexes.

- **table scans (short tables)** - the number of full tablescan performed on tables with less than 5 blocks[22].

[22] Tables created with the CACHE option are considered candidates for short tables.

Table Scans (long tables) plus Table Scans (short tables) is equal to the number of full table scans performed during this interval.

- **free buffer waits** - is the number of times processes had to wait because a buffer was not available. This can occur when Dbwr reaches the *free buffers inspected* limit. A high number indicates that Dbwr is not writing enough dirty buffers to disk.
  If **free buffer waits** / **free buffers scanned** * 100 > 5% than possibly increase *db_block_write_batch*. Also check for disk contention.

- **buffer busy waits** - is the number of times processes had to wait for buffer. This happens because either the buffer is being read in the cache by another user or the status of buffer is incompatible with requested mode.
  (**buffer busy waits**)*100 / (**db blocks gets** + **consistent gets**) should not be > 5%. Perform a "select * from v$waitstat " to determine what type of objects have waits against them and tune accordingly.

- **db file sequential read, db file scattered read, db file single write and db file parallel write** are all events corresponding to I/Os performed against the data files headers, control files, or data files. If any of these wait events correspond to high Average Time, then investigate the I/O contention via sar or iostat. Look for busy waits on the device.

- **cache buffer handles** - This latch protects the State Objects that are needed by a process to make a change to a block/buffer. Before the change a buffer handle is acquired either from the process' private pool or the global pool if none are present. The access to the global pool is protected by this latch.

- **cache buffer chains** - Before modifying a buffer the foregrounds need to get a latch to protect the buffer from being updated by multiple users. Several buffers will get hashed to the same latch. The buffers get hashed to a cache buffer chain latch depending on the DBA of the block that is in the buffer.
  Contention against this latch may indicate that a particular hash list grown too large or there are several CR copies of same block. Use the following query to determine if this is the case:

```
select dbafil "File #", dbablk "Block #",count(*)
  from x$bh
 group by dbafil, dbablk
having count(*) > 1 ;
```

- **cache buffer lru chain** - Protects the Least Recently Used list of cache buffers. For moving buffers around on this list the Oracle7 kernel needs to get this latch. If the contention for this latch is high, it may require the _db_block_write_batch to be increased or _db_writer_max_scan_cnt to be decreased. Note, for SMP systems, the LRU latch can be controlled via the init.ora parameter db_block_lru_latches. Oracle by default will set this value to ½ * #CPUs. This parameter dictates the number of LRU latches per instance. This will alleviate the LRU from being the bottleneck, since each LRU will maintain a list of block buffers (buffers are assigned to a LRU via hashing). Generally db_block_lru_latches is set to no more than 2*#CPUs.

- **cache protection latch** - During the cloning of a buffer the buffer header is atomically updated to reflect the new position in the buffer cache. Only one buffer at the time may be cloned, as it is protected by this latch. A lot of contention on this latch would indicate that there is a lot of concurrent cloning going on.