All About Oracle’s Touch Count Algorithm

Understanding Oracle’s Touch Count Buffer Cache Algorithm

Latest version available at http://www.orapub.com

All About Oracle's Touch Count Data Block Buffer Cache Algorithm

Craig A. Shallahamer (craig@orapub.com)

Original 2001

Version 4a, January 5, 2004

INTRODUCTION

Oracle introduced a *touch-count* based data buffer management algorithm to address the growing size, performance requirements, and complexities of current day systems. Data block buffers are no longer managed using a modified least recently used (LRU) algorithm. The touch-count based algorithm introduced in Oracle 8i significantly reduces latch contention, forces buffers to *earn* their way higher up an LRU, and allows for increased granular buffer performance optimization opportunities. This paper details Oracle's touch-count algorithm, how to monitor its performance, and how to manage for optimal performance.

NEW ALGORITHM NEEDED

Things change. In the Oracle world, things change fast and with significant impact. Because of change and the need to remain competitive, Oracle has essentially been forced to re-evaluate many of its central algorithms. For example, the redo buffer algorithm has changed over the past few years. But why a new data block buffer algorithm? Answer: Bigger caches, increased performance requirements, and increased buffer cache control.

Over the past few years the possible size of an Oracle data block buffer cache has dramatically increased into the mulit-gigabyte range. This can cause a performance problem because Oracle's data block buffer algorithm was designed and built many, many years ago when systems where much less powerful than systems today. DBAs who have experimented with very large data block buffers have personally experienced a performance *decrease*. That is correct, decrease. Why? Oracle was designed for I/O and so when a buffer cache nature shift occurs, a different algorithm must be employed.

Simply being able to manage a very large buffer cache is not enough. In fact, maintaining performance with a large cache is not enough. To remain competitive, Oracle systems must not only support large caches, but provide significant performance gains. Again, this required a new algorithm.

In addition to larger caches and increased performance requirements, Oracle has been moving its marketing position and technical position from distributed databases to a massive central Oracle database position. This requires increased control. Oracle is slowly providing the DBA with ways to segment a single database system into smaller sub-databases. As you will see, Oracle’s touch-count implementation is highly tunable which will provide some of the required control to manage the buffer cache in a highly dynamic and very large cache.

When put together; bigger caches, increased performance requirements, and increased control, it should not be surpassing that Oracle has chosen a new data block buffer cache algorithm. As the next section discusses, what is surprising to many is that Oracle has been changing its database block buffer algorithms for many years.

HISTORY OF ORACLE BUFFER CACHE MANAGEMENT

If you have been working with Oracle for many years, you may never have thought about it, but Oracle has been modifying its data block buffer algorithm. The center of attention is the least recently used list – or LRU for short. Conceptually, an LRU is simply a list of pointers to the cached database blocks. In general, toward one end of this list are the popular buffers and on the other end the not-popular buffers. The popular end of the LRU chain is called the *most recently used* or the MRU end of the list. The not-popular end of an LRU chain is called the least recently used or an LRU end. And yes, it is sometimes confusing talking about an LRU end of an LRU chain. But after a while you get used to it. Regardless of the naming conventions, the general idea is to keep popular blocks cached to minimize physical I/O.

In the beginning, Oracle implemented what is known as a *standard* least recently used algorithm. Whenever a buffer was brought into the cache or touched again while in the cache, the buffer’s pointer (remember, the blocks are never physically moved, just a pointer to the physical memory block) was moved to the MRU end of an LRU chain. The thinking here is that the not-popular blocks would naturally migrate towards the LRU end of an LRU list and perhaps their contents filled with a more popular block.

However, there is a cache killer called the full-table scan where each and every table block is placed into the buffer. If the buffer cache is 500 blocks and the table contains 600 blocks, all the popular blocks would be replaced with this full-table scanned table’s blocks. This is extremely disruptive to consistent database performance because it forces excessive computer system usage and basically destroys a well-developed cache.

To combat this destructive force, the famous *modified least recently used* or more frequently referred to as the *modified LRU* algorithm was implemented. While Oracle has changed this algorithm and our control over it over the years, its basic operation has remained the same. The modified LRU algorithm places full-table scanned blocks read into the buffer cache at the LRU end of the LRU chain and only permits a limited number of these blocks to exist in the cache at once. This prevents a full-table scan from flushing the entire buffer cache.

While this might seem like all our buffer cache problems are solved, think again. How about a large index range scan? Picture hundreds of index leaf blocks flowing into the buffer cache. The modified LRU algorithm only addresses full-table scan issues, not index block issues.

An interim solution and to remain competitive, Oracle implemented multiple buffer pools. While most DBAs do not use multiple buffer pools and the touch-count algorithm makes them less attractive, they can be very effective. The *keep pool* is designed for small objects that, for application response time issues, should *always* be cached. The *recycle pool* is designed for larger objects that can disrupt normal and smooth buffer cache operations. And finally, the *default pool* is for everything else.

To maintain smooth and effective buffer cache operation, a very fast and very flexible algorithm must be implemented that essentially forces *every* buffer to seriously earn the right to remain in the buffer cache. A touch-count algorithm is all about forcing each and every block to earn its way not only into the buffer cache but to remain in the buffer cache. As you will read below, the words *seriously earn* is an understatement. Even the default touch-count related instance parameters make it very difficult for a block to simply remain in the buffer cache. It's almost cruel how many hurdles a buffer must *continually* jump to simply remain in the buffer cache. How this works in general and Oracle's specific implementation is presented in the next section.

TOUCH-COUNT BUFFER MANAGEMENT

Since a touch-count based algorithm, sometimes referred to as a *count frequency scheme*, is more complicated than a simple LRU algorithm, there must be some motivating factors before the benefits of using a touch-count algorithm surpass its cost and also provide increased performance when compared to Oracle's past LRU algorithms. We have already discussed these motivating factors. They are bigger caches, performance requirements, more control, full-table scan access, and index access. These factors create an intense and very strenuous cache environment. It is only then that the touch-count algorithm becomes economical.

At the core of any touch-count based algorithm are the harsh requirements placed on each buffer to not only remain in the cache, but to remain in the MRU end of an LRU list. Conceptually, the touch count algorithm does this by assigning each buffer a *counter*. Each time the block is touched its counter is incremented. The value of this touch counter is used to assign a kind of *value* or *popularity* to each buffered block. This touch count is then referenced and manipulated during buffer cache activity. The reference and manipulation specifics are touch-count algorithm implementation specific. This means, Oracle's implementation may be different than another vendor's implementation.

ORACLE'S TOUCH-COUNT IMPLEMENTATION

Oracle does a number of nasty things to make it very difficult for a buffer to remain in the buffer cache. Only the chosen few can remain in the MRU end of an LRU chain. What does Oracle do that makes the life of a buffer so difficult? The next few sections specifically addressees this topic.

MIDPOINT INSERTION

Each LRU is divided up into the two basic areas or regions; a *hot region* and a *cold region*. All buffers in the hot region are called hot buffers and all buffers in the cold region are called cold buffers. There is a midpoint marker between the hot region and the cold region. The midpoint pointer moves to ensure the appropriate number of buffers are in the hot region and in the cold region. That is, the midpoint pointer is not associated with a specific buffer.

By default Oracle divides the LRU evenly. That is, the hot region is composed of 50% of the buffers and the cold region is composed of 50% of the buffers. This can be controlled by manipulating the instance parameter, **_db_percent_hot_default**. Increasing this instance parameter will increase the percentage of buffers in the hot region, that is, buffers above the midpoint.

When a server process reads a block from disk and places it into the buffer cache, it places the buffer in the middle of an LRU chain, that is, between the hot and cold regions. This is known as *midpoint insertion* and is a fundamental concept in Oracle's touch-count algorithm implementation. It is important to grasp that the buffer is *not* placed at the MRU end of an LRU chain and that the buffer must work or earn its way to the MRU end of the LRU chain.

TOUCH COUNT INCREMENTATION

In concept, each time a buffer is touched, for whatever reason, its touch count is incremented. In practice this is not the case, which makes its more difficult for a buffer’s touch count to increase. In a buffer’s life, there are times when it will be of intense interest to server processes and then drop out of favor very quickly. This “bursty” kind of buffer activity would reek havoc in a touch count based algorithm. So reduce this problem, Oracle only allows a buffer’s touch count to be increased, at most, once every three seconds by default. This does not mean a buffer’s touch count can only be incremented once within any three second period. If you graph out the time sequence, you will see a touch count could be incremented twice within three seconds. This “three second period” can be modified by the instance parameter, **_db_aging_touch_time**.

Another important aspect of touch count incrementation is the buffer is not moved, that is, its pointer is not moved. Touch count incrementation and buffer pointer movements are *independent*. Buffer pointer movement will be discussed below.

A latch is simply a token an Oracle process may need to get to enable execution of a specific section of Oracle kernel code. *Anytime* there is an Oracle buffer, cache, list, node, etc. activity, a latch or latches are involved to ensure cache correctness. Well…now there is an exception. Oracle updates a buffer’s touch count *without* a latch. This means a buffer may be manipulated while the touch count is being incremented, but more interesting is that two processes may simultaneously update a buffer’s touch count to the same value. If this occurs, Oracle assures us the worst that could happen is the touch count is not actually incremented every time a buffer is touched and that no cache corruption will result. While shocking at first, this makes a lot of sense….without using a latch, it’s impossible to get latch contention.

BUFFER MOVEMENT

As mentioned previously, when a buffer is brought into the buffer cache, it is placed at the midpoint. Unlike least recently used algorithms, Oracle’s touch-count algorithm will not move a buffer just because is touched. Yes, its touch count will probably be incremented, but it is not moved.

When a server process is looking for a free buffer to place an Oracle block into the buffer cache, it must first find a *free* buffer. A *free* buffer is simply a buffer whose contents matches the copy on disk. That is, there is no difference. If there is a difference, the block is tagged as *dirty*.

When a server process is searching for a free buffer or a database writer (DBWR) is looking for dirty blocks (sometimes called *victims*), and if a buffer’s touch count is observed to be greater than two, it is moved to the MRU end of the LRU chain. The default block movement threshold of two is controlled by the instance parameter, **_db_aging_hot_criteria**.

Oracle’s touch count implementation is tough on buffers! When a buffer is moved to the MRU end of the LRU chain, its touch count is usually reset to zero![1] As I’ll describe below in more detail, the now popular buffer, which just had its touch count chopped to zero will need its touch count sufficiently incremented or it could be paged out to disk.

---

[1] People I usually trust say that when a buffer is moved to the MRU end of an LRU chain, its touch count is reset to 0 unless **_db_aging_stay_count** (default: 0) >= **_db_aging_hot_criteria** (default: 2), in which case the touch count is set to the current touch count divided by two.

HOT TO COLD MOVEMENT

If a buffer is moved from the cold region to the hot region (i.e., to the MRU end of the LRU chain), the midpoint marker must also move one position to ensure the proper number of number of hot and cold buffers are present. When this occurs and with no fault of its own, a buffer will be forced to cross over the threshold from the hot region into the cold region. When this occurs, Oracle once again resets the buffer's touch count to one. Even if the buffer's touch count is 250, if it crosses the threshold, then its touch count will be set to 1. The threshold crossing touch count reset value is controlled by the instance parameter, **_db_aging_cool_count**. This means the buffer must now be touched all over again (to increase it's touch count) before a server process or the DBWR asks, "What's your touch count?" or it could be replaced.

TOUCH-COUNT PERFORMANCE TUNING

Performance tuning can consist of only a few fundamental steps. First the system under investigation must be monitored. Second, if there is contention or a problem, hopefully it will be identified. Next, an analysis must be performed and finally a solution implemented. Discussed below is how to monitor Oracle's buffer cache and ways to utilize our control over the touch-count algorithm by appropriately manipulating the available instance parameters.

TOUCH-COUNT PERFORMANCE MONITORING

Monitoring Oracle buffer cache performance is algorithm independent. However, if a buffer cache related problem is identified, your solution options are algorithm dependent. The best way to accurately identify a buffer cache problem, or any Oracle server based performance problem for that matter, is to monitor response time and its components. While this topic is discussed in detail in my response time paper [9], I will quickly summarize.

Response time analysis considers both service time and queue time. Service time is the CPU time used during a database operation (e.g., buffer cache management) and queue time is time the CPU is waiting on something to continue processing (e.g., latching). Ratio based analysis does not consider service time or queue time, but focuses on Oracle system activity, and therefore *leads* one into where the problem *might* reside. Session wait analysis effectively identifies queue time and its components, but does not consider service time. This can sometimes lead to incorrect problem identification. Response time analysis considers both service time and queue time in its analysis, thereby *always* correctly identifying where the real Oracle problem resides. When queue time components are identified and categorized, the performance bottleneck is quickly and confidently identified.

If the buffer cache is the main problem issue, one of the following Oracle wait events will consume the majority of the queue time. The buffer cache associated wait events are *LRU chain latch*, *free buffer wait, cache buffer chains latch*, *buffer busy wait, db file sequential read* (single block IO), and *db file scattered read* (multi-block IO). Once buffer cache contention is discovered and if the touch-count algorithm is being used, then one of your options is to manipulate the touch-count related instance parameters. This is discussed in a following sections.

CLASSIC BUFFER CACHE PERFORMANCE TUNING

While manipulating the touch-count related instance parameters, which is discussed in the next section, can be effective in resolving buffer cache performance issues, directly addressing the wait events is probably your best solution. Below is a brief list of what to do to address each of the key buffer cache related wait events. I am sorry, but this paper is not the appropriate place to explain *why* I make the below suggestions or *how* to actually perform what I suggest. If you have specific issues or questions, please feel free to email me. Or better yet, considering attending my *Advanced Reactive Performance Management* [1]course which covers this topic

LRU chain latch wait. Reduce the number cache blocks being touched, reduce the number of processes requiring an LRU chain latch, increase the number of LRU latches, reduce latch hold duration, add more CPUs, and use faster CPUs.

Free buffer wait. Increase I/O subsystem throughput capacity, alter the application to produce less dirty blocks, give the DBWRs more power, and increase the number of data block buffers to better absorb activity bursts.

Buffer busy wait. Determine if the same buffer is busy or if there are many, many buffers that have been busy. If the same buffer is busy, then fix the application or reduce the block's popularity. If many buffers are busy, then increase I/O subsystem throughput capacity, or reduce the number of busy buffers.

Cache buffer chains latch wait. Determine if there are many copies of the same block in the buffer cache. If there are many copies, then fix the application or reduce the block's popularity. Otherwise, increase the number of block buffers, increase the number of hash buckets, use faster CPUs, or user more CPUs.

Db file sequential or scattered read waits. Change the application to reduce the physical I/O requirements (e.g., tune the SQL), increase I/O subsystem throughput capacity, increase the buffer cache, or increase physical memory.

TOUCH-COUNT PARAMETER PERFORMANCE TUNING

There are five relevant touch-count related instance parameters that we can use to our advantage. Each of these parameters has been discussed above, but for reference purposes, I listed each below.

First off and of utmost importance, if response time analysis does not point to a CPU shortage combined with a queue time problem predominately composed of buffer cache contention (the wait events have previously been discussed), then messing with the touch-count parameters is of no real value. If response time analysis does point to a queue time problem predominately composed of buffer cache contention along with a CPU shortage, then messing with the touch-count parameters *can* be of real value. Remember our objectives are to minimize buffer movement and keep the cache full of the most popular blocks.

_db_percent_hot_default. The percentage of block buffers that reside in the hot region. Default 50 (percent). If you want more blocks to be considered *hot*, than increase this parameter. If you want to push more blocks out into the cold, decrease this parameter. By increasing this parameter, you give a buffer less time to get its touch count increased before a server process or the DBWR asks its touch count. This could result in buffer cache thrashing.

_db_aging_touch_time. The window of time where a buffer's touch count can only be incremented by one. Default 3 (seconds). If there is too much block movement or popular block burst activity, increasing this parameter value will reduce blocks reaching the movement threshold, therefore reducing buffer movement. Severe cache buffer chains latch or LRU latch contention is an indication of too much block movement. Increase this parameter to reduce sudden activity disruptions and to reduce buffer movement. However, be careful. If you increase this parameter too much, you are essentially devaluing popular blocks.

_db_aging_hot_creiteria. The threshold when a buffer is being considered to be moved to the MRU end of the LRU chain. Default 2 (touch count). If you need to slow down block movement or increase buffer replacement then increase this parameter's value. If you want to make it more difficult for a buffer to be moved (to the MRU end of the list), then increase this parameter. The result will be only the really popular blocks will remain in the cache.

_db_againg_stay_count. Involved with resetting the touch count when a buffer is moved to the MRU end of the LRU chain. Default 0 (touch count). Since a buffer's touch count is reset when it becomes a cold buffer, changing this value should have no affect on buffer replacement.

_db_aging_cool_count. The re-assigned touch count value when a block moves from the hot region into the cold region. Default 1 (touch count). Increasing the value will make it easier for blocks to stay in the cache, will also slow down and smooth the *cooling* affect, and increase the likelihood of blocks promotion. If popular blocks are being prematurely pushed out of the cache (evidenced by a low buffer cache hit ratio), increasing the parameter value will help these blocks remain in the cache longer. However, if this value is too high, then unpopular blocks will also remain in the cache resulting in block thrashing evidenced by significant cache buffer chains latch or LRU latch contention.

CONCLUDING THOUGHTS

Oracle has refined and will continue to refine its most fundamental cache management algorithms. But it is an exceptionally risky venture fraught the real possibility of introducing bugs and slowing down systems. So there must be a very, very solid argument for making such a fundamental change. Oracle must have felt that the surrounding changes (larger caches, increased performance requirements, increased cache control, future product enhancements) warranted a radically new buffer cache algorithm. And it must have worked because you don't hear people yelling at Oracle about the touch count algorithm or LRU latch contention like we used to.

ACKNOWLEDGEMENTS

A special thanks to my email acquaintance, clients, and students who have brought forth a plethora of stimulating discussions and challenging dilemmas. These situations coupled with my unusual enthusiasm to Oracle performance analysis has evolved into this technical paper. A special thanks is in order for Mr. John Beresniewicz. His thorough review, expertise in the Oracle 8i buffer cache, and his willingness to share his knowledge has always been deeply appreciated.

REFERENCES

The most updated version of this paper can always be found at www.orapub.com.

1. "Advanced *Reactive* Performance Management For Oracle Based Systems" Class Notes (1998-continual update). OraPub, Inc., http://www.orapub.com/training
2. " Advanced *Proactive* Performance Management For Oracle Based Systems" Class Notes (1998-continual update). OraPub, Inc., http://www.orapub.com/training
3. Jain, R. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991. ISBN 0-471-50336-3
4. Michalko, M. *Thinkertoys*. Ten Speed Press, 1991. ISBN 0-89815-408-1
5. *"OraPub System Monitor (OSM)"* tool kit (1998-continual update). OraPub, Inc., http://www.orapub.com/tools
6. Shallahamer, C. *Avoiding A Database Reorganization.* Oracle Corporation White Paper, 1995. http://www.orapub.com/papers
7. Shallahamer, C. *Optimizing Oracle Server Performance In A Web/Three-Tier Environment.* OraPub White Paper, 1999. http://www.orapub.com/papers
8. Shallahamer, C. *Oracle Performance Triage: Stop The Bleeding!* OraPub White Paper, 2001. http://www.orapub.com/papers
9. Shallahamer, C. *Oracle Response Time Analysis: The Next Performance Management Frontier*. OraPub White Paper, 2001-continual update. http://www.orapub.com/papers
10. Shallahamer, C. *The Effectiveness of Global Temporary Tables* OraPub White Paper, 2001. http://www.orapub.com
11. Shallahamer, Craig A. (1995). *Total Performance Management*. Published and presented at various Oracle related conferences world-wide. http://www.orapub.com/papers

ABOUT THE AUTHOR

Quoted a being "An Oracle performance philosopher who has a special place in history of Oracle performance management," Mr. Shallahamer brings his unique experiences to many as a keynote speaker, a sought after teacher, a researcher and publisher for ever improving Oracle performance management, and the founder of the grid computing company, BigBlueRiver. He is a recognized authority in the Oracle server technology community and is making waves in the grid community the result of founding a company which provides "Massive grid processing power—for the rest of us."

Mr. Shallahamer spent nine years at Oracle Corporation personally impacting literally hundreds of consultants, companies, database administrators, performance specialists, and capacity planners throughout the world. He left Oracle in 1998 to start OraPub, Inc. a company focusing on "Doing and helping others Do" both reactive and proactive Oracle performance management. He continues to push performance management forward with his research, writing, consulting, highly valued teaching, and speaking engagements.

Combining his understanding of Oracle technology, the internet, and self organizing systems, Mr. Shallahamer founded BigBlueRiver in 2002 to help meet the needs of people throughout the world living in developing countries. People with limited technical and business skills can now start their own businesses which supply computing power into BigBlueRiver's computing grid. In a small way, this is making a difference in potentially thousands of people's lives.

Whether speaking at an Oracle, a grid computing, or a spiritual gathering, Mr. Shallahamer combines his experiences and his purpose toward communicating his unique insight into the technologies, the challenges, and the controversies of both Oracle and grid computing.