Why a 99%+ Database Buffer Cache Hit Ratio is *Not* Ok

Cary Millsap/*Hotsos Enterprises, Ltd.*

Introduction

Many tuning professionals and textbook authors sell advice encouraging their customers to enjoy the performance virtues of Oracle database buffer cache hit ratios that approach 100%. However, database buffer cache hit ratio is *not* a reliable system performance metric. Buffer cache hit ratios above 99% usually indicate particularly serious SQL inefficiencies. Repairs of these inefficiencies often yield 100× or greater performance improvements, yet the optimizations result in *reduced* database buffer cache hit ratios. This paper shows why a 99%+ buffer cache hit ratio almost always signals performance inefficiencies, including real-life examples. It shows how to detect the inefficiencies and how to repair them. It concludes with recommendations about what metrics the performance analyst should use instead of the venerable but unreliable database buffer cache hit ratio.

The State of Oracle System Performance Analysis

The state of Oracle system performance analysis is not good enough. For most people, Oracle optimization is an art that requires intuition, luck, and a lot of time. Most people regard the art as so simultaneously difficult, frustrating, and important that they pay thousands of dollars per week to have consultants fix performance problems for them. In many cases I've seen, performance problems linger for months without proper diagnosis. For a few people I've met, Oracle optimization is a legitimate science that is based on a simple, repeatable, step-by-step method. These few people diagnose system performance problems literally in minutes. In their diagnoses, they forecast the performance impact of the solutions they propose with startling accuracy.

The world where Oracle optimization is an art is a scary place where minor victories are met with celebration. It is a place where things sometimes happen for no apparent reason. It is a place where nobody local quite seems to know what is going on, but there are an anointed few whose advice is popularly regarded as trustworthy. Living in Art World[1] is frustrating and expensive. If the things you know how to fix don't fix your problem, then you seek new lists of things to check from books, newsgroups, and conference presentations. People in Art World learn through trial and error, but it's so difficult to attribute effects to causes that no two companies seem to be able to use exactly the same techniques to improve performance. As a result, the state of the art advances very slowly, if at all.

Living in Science World is fun, fulfilling, and economically efficient. It is rarely mysterious, and when it is, the mystery only lasts for a little while. This world really does exist. However, you won't get there by reading the Oracle textbooks that are in bookstores today.[2] Today's textbooks prescribe only the artistic approach. They teach that by using the right checklists and checking the right ratios, you can find and fix performance problems. This is the heart of the difference between Art World and Science World. In Science World, we use a simple, high-precision problem diagnosis technique to determine exactly where the response time for a session is being consumed. The technique reveals not just the bottleneck, but by *exactly how much time* the bottleneck has degraded the response time of a problem session.

In order to take residence in the pleasant place I'm calling Science World, it is necessary to give up certain dearly held beliefs about key performance metrics. This paper is devoted to detailing the dangers of relying on one of the oldest and perhaps most dearly held beliefs of all: the belief that the database buffer cache hit ratio is a reliable performance indicator. In this paper, we will show why the Oracle database buffer cache hit ratio is an unreliable

[1] For my potentially unfortunate choice of metaphor, I apologize in advance to artists, appreciators of art, and people named Arthur.

[2] Since the time this statement was originally written, several excellent books have been published that do actually prescribe the scientific approach. Excellent recent works include [Lewis 2001; Morle 2000; Vaidyanatha et al. 2001].

system performance diagnostic tool, and we will provide a glimpse into the advanced technology that you can use to fill in the hole we will leave.

Database Buffer Cache Hit Ratio

One of the first ratios that Oracle database administrators learn is the *database buffer cache hit ratio*. The database buffer cache hit ratio is the proportion of Oracle logical reads that do not require operating system (OS) read calls. When an Oracle session accesses a block, Oracle executes an instruction path that determines whether or not that block is already in the *database buffer cache*. We refer to this instruction path as an Oracle *logical I/O call: LIO* for short. If the block is not in the buffer cache, then Oracle will execute an OS read call. We refer to this as a *physical I/O call*, or *PIO* for short. LIO calls and PIO calls relate to each other in the following manner [Millsap, Holt 2001]:

```
procedure LIO(data block address, …)
    if the block is not in the database buffer cache, then
        PIO(data block address, …)  /* execute an OS read call (potentially a multi-block pre-fetch)[3] */
        ‘physical reads’ += number of Oracle blocks read
    update the LRU chain if necessary  /* necessary less often in 8.1.6+ than in prior Oracle releases */
    if mode eq ‘consistent’, then
        clone(data block address, …)
        ‘consistent gets’++
    else  /* mode eq ‘current’ */
        ‘db block gets’++
```

Note that every PIO counts as an LIO. Some LIOs, however, never require a PIO.

The database buffer cache hit ratio is simple to compute. It is commonly defined as follows, using statistics you can obtain by querying *v$sysstat*:[4]

$$\begin{aligned}\text{database buffer cache hit ratio} &= \frac{\text{LIO} - \text{PIO}}{\text{LIO}}\\ &= 1 - \frac{\text{PIO}}{\text{LIO}}\\ &= 1 - \frac{\text{physical reads}}{\text{db block gets} + \text{consistent gets}}.\end{aligned}$$

Intuitively, it makes sense to maximize this ratio for a system. For Oracle PIO calls that actually read from disk,[5] the PIO call duration is a few hundred times greater than the duration of an Oracle LIO call that is fulfilled from the database buffer cache. So the argument goes that a higher buffer cache hit ratio indicates better response times and happier users.

Many DBAs rely on the database buffer cache ratio as a key measure of system performance health. Several authors, speakers, consultants, and software tools encourage the notion that increasing a system’s database buffer cache hit ratio equates to making that system faster. As we shall soon see, it is a false claim. It is true that a very low database buffer cache hit ratio can indicate a specific type of problem. But what many people do not know is that a very *high* buffer cache hit ratio value is a reasonably reliable indicator of severe system inefficiencies.

[3] If the LIO is a participant in an execution plan that can benefit from a multi-block PIO call, then Oracle might pre-fetch blocks during a physical I/O call. See [Holt 2000b] for more information about read call sizes.

[4] Many books about Oracle performance show you the SQL to compute this ratio. Because my aim is to discourage you from using this ratio to manage your system, I shall refrain from showing you the SQL to compute it.

[5] Note that when Oracle issues an OS read call, it doesn’t necessarily mean that the operating system will actually perform a physical read. On operating systems that buffer I/O calls in OS memory (such as standard UNIX filesystems), it is possible for the OS to fulfill a read call issued by Oracle without ever having to visit a disk. A similar phenomenon can occur with any disk subsystem that uses memory cache as an intermediary between the application and the physical disks.

Hit Ratio is an Unreliable Performance Indicator

The big problem with the database buffer cache hit ratio is that it is an unreliable performance indicator. Having a "good" database buffer cache hit ratio does *not* imply that you have an efficient high-performance system. There are many situations in which there is no correlation between system performance and database buffer cache hit ratio. Many slow systems have high hit ratios, and fast systems can have low hit ratios. The ratio does not tell you whether or not a system is optimized.[6]

As a performance indicator capable of helping *diagnose* a problem, the database buffer cache hit ratio is especially weak. The reason is simple: if you use the metric in the way the authors recommend, then the *only* possible course of action that you can discern directly from the study of a database buffer cache hit ratio is that you should add more memory to the database buffer cache. Rarely in a competently administered database will executing this recommendation actually improve performance.[7]

Many people are surprised the first time they find out that increasing a system's buffer cache hit ratio does not improve system performance. The intrigue increases when they see that the reason for many 95%+ cache hit ratios is the presence of highly inefficient application SQL. The following sections demonstrate how a focus on "improving" the database buffer cache hit ratio stimulates actions that yield negative impact to your system's performance.

High Hit Ratio Conceals Inefficient SQL, Part I

In my classes, I like to guide my students through the following thought experiment to help them understand why the database buffer cache hit ratio is an unreliable performance problem diagnosis indicator:

Example: Two distinct SQL statements, *A* and *B*, return identical row sets, but the two statements have different execution plans. Which statement would you rather have on your system?

SQL statement	Database buffer cache hit ratio
A	99.99%
B	90.00%

The conventional answer is that one should rather have statement *A* on one's system, because it has the higher (and therefore presumably better) database buffer cache hit ratio. Statement *A* returns its result set with a smaller proportion of LIOs requiring OS read calls than statement *B*, so the conventional wisdom about database buffer cache hit ratios says that *A* is better than *B*.

However, the correct response is that, knowing only the information presented here, it is impossible to determine the answer to the question. Let's look one level deeper into the details of what workload the statements generate. Given the following bits of additional information, *now* which statement would you rather have on your system?

SQL statement	Database buffer cache hit ratio	LIOs	PIOs	Execution time
A	99.99%	10,000	1	0.405 sec
B	90.00%	10	1	0.005 sec

This table clearly reveals that statement *B* is superior. Statement *B* will produce identical business results while incurring 1,000 times less LIO workload upon a system. Especially if the business function

[6] Unfortunately, this statement is not unique to discussions about the database buffer cache hit ratio. Most of the ratios that you find in textbooks about Oracle performance are both unreliable performance metrics and unreliable performance problem diagnosis indicators. They are unreliable for the same types of reasons that we'll explore as we discover the manifest weaknesses of the database buffer cache hit ratio.

[7] Specifically, you should only need to add memory to the database buffer cache in only two events: (1) when you initially configure a new Oracle system (because the shipped value of *db_block_buffers* is often too small; more about that later…); and (2) when you significantly increase the number of concurrent sessions to which an existing Oracle system is required to provide service.

implemented by these SQL statements is used several hundred times a day, switching from *A* to *B* represents a *huge* optimization.

In 1989, I witnessed an instructor tell a SQL optimization class that, "LIOs are essentially free; it's the *PIOs* you must eliminate to have a well optimized system." This turned out to be one of the worst bits of technical advice that anyone has ever sold me. Certainly, it is bad for a SQL statement to generate Oracle PIOs that could have been prevented. However, *LIOs* are an Oracle system's primary CPU consumer. It is crucial to optimize LIO call frequencies because CPU is the least scalable and most expensive resource on modern-day computers. A key difference between a fast, high-concurrency application and one that won't scale to its user load is how well its owners have reduced the system's *LIO* call frequency.

The belief that LIOs are "essentially free" appears widespread even today. Niemiec, for example, notes that, "retrieving information from memory is over 10,000 times (depending on the memory you have) faster than retrieving it from disk" [Niemiec 1999 (9)]. Regardless of whether memory is over 10,000 times faster than disks, an Oracle LIO is *not* over 10,000 times faster than an Oracle PIO. The difference in LIO and PIO elapsed times is a factor of only a few hundred.[8] Mr. Niemiec almost says this later in his book when he says, "In tests, the same amount of CPU was used for 89 logical reads as it was to process 11 physical reads" [1999 (146)].[9] Unfortunately, he interprets this evidence only in support of his thesis that PIOs are so expensive (because in addition to consuming elapsed time on the disk, they also consume CPU). His book does not acknowledge the evidence as also pointing out the non-trivial CPU cost of a LIO.

LIO's are far from free. Notably, *every* Oracle LIO call requires a latch-serialized search of the database buffer cache hash table, and a parse of the contents of the multi-kilobyte Oracle block itself. Many LIOs also require latch-serialized manipulation of a cache buffers LRU chain (this is especially true before Oracle Release 8.1.6). And some LIOs require execution of significant additional codepath motivated by read consistency requirements. Because of the latch serialization induced during LIO processing, too many LIOs prohibit an Oracle system from scaling. Unfortunately, having too many LIOs also makes a database buffer cache hit ratio look really good.

Considering just our simple thought experiment above helps us understand why lots of nines in an Oracle database cache buffer hit ratio is especially likely to indicate a workload inefficiency. It is impossible for an individual SQL statement to have *n* nines in its database buffer cache hit ratio without having at least *n* significant digits in its LIO count. For example, you can't have a 99.999% cache hit ratio unless you have at least 100,000 LIOs in your hit ratio denominator. A LIO-to-PIO ratio as low as even 1,000-to-1 (i.e., a cache hit ratio of 99.9% or greater) is almost inevitably a bad thing.

Having lots of LIOs almost always means trouble. Certainly, LIO processing is the dominant workload component in a competently managed system. Reducing LIO counts thus becomes the performance optimizer's number one job. The job becomes even more important on high-concurrency systems, where inefficient statements with too many LIO calls serialize on latches, increasing the number of 'latch free' events as LIO calls wait for a *cache buffers chains* latch. Latch contention problems are examples of serialization bottlenecks that cannot be cured by addition of hardware capacity to a system.[10]

High Hit Ratio Conceals Inefficient SQL, Part II

The next case is a striking example of how a single statement's LIO inefficiency can devastate the performance of an entire system.

[8] A recent test at a Hotsos customer site running a Compaq Alpha computer showed the LIO capacity of each 731-MHz CPU to be less than 100,000 LIOs/sec. If you assume that a disk today can fulfill a PIO call in 10 ms, then the PIO capacity of a disk is 100 PIOs/sec. So, even on a fast system with slow disks, a LIO is still less than 1,000 times faster than a PIO. On modern disks with 5 ms access times, a LIO on a fast CPU is less than 500 times faster than a PIO. On a system with slower CPUs, the ratio will be even lower.

[9] It is important to note that most of the CPU consumed during a PIO call is CPU consumed during the LIO that has motivated the PIO.

[10] Not even the invention and installation of 10 times as many CPUs that are each 10 times faster than yours can solve a serialization bottleneck if its magnitude is an inefficiency of 100 times or more. Many latch contention problems are caused by SQL statements that issue thousands or millions of times more Oracle LIO calls than they should.

Example: The following statement is an example of SQL that qualifies as perfect if viewed traditionally in terms of its database buffer cache hit ratio. If we had searched for inefficient SQL by using statements' database buffer cache hit ratios as a performance indicator, we'd never have found this one, because its hit ratio is a perfect whopping 100%.[11,12]

```
SELECT
   RCEPI.EC_PO_ID,
   RCEPI.SUPPORT_ITEM_ID,
   RCEPI.SUPPORT_PRICE,
   RCEPI.QUANTITY,
   RCEPI.SYSTEM_ID,
   RCEPI.CUSTOMER_PRODUCT_ID,
   RCECP.CUSTOMER_ID,
   RCECP.BILL_TO_SITE_ID,
   RCECP.SHIP_TO_SITE_ID,
   RCECP.CONTACT_ID,
   RCECP.PO_CREATE_DATE,
   RCECP.OMTX_ORDER_NUM,
   MSI.PRIMARY_UOM_CODE,
   NVL(MSI.SERVICE_DURATION,1) SERVICE_DURATION,
   MSI.SERVICE_DURATION_PERIOD_CODE
FROM
   CS_EC_PO_ITEM RCEPI,
   CS_EC_CUST_PO RCECP,
   MTL_SYSTEM_ITEMS MSI
WHERE
   RCEPI.CP_SERVICE_TRANSACTION_ID IS NULL
AND RCEPI.CUSTOMER_PRODUCT_ID IS NOT NULL
AND RCEPI.SUPPORT_ITEM_ID IS NOT NULL
AND RCEPI.EC_PO_ID = RCECP.EC_PO_ID
AND RCEPI.SUPPORT_ITEM_ID = MSI.INVENTORY_ITEM_ID
AND MSI.ORGANIZATION_ID = 21

SessionTotal 1936  StatementTotal 1936  PercentOfSession 100
FirstParsingUser 0  OptimizerGoal CHOOSE
HprofHash 2052618893  OracleHash 1780117128  RecursiveLevel 0

Statement Total Statistics
a  lcm    count       cpu    elapsed       disk      query    current       rows
- ---- -------- --------- ---------- ---------- ---------- ---------- ----------
P    0        1         0          0          0          0          0          0
E    0        1         0          0          0          0          0          0
F    0        2      1936       2000          0     610413          0          2
- ---- -------- --------- ---------- ---------- ---------- ---------- ----------
T    0        4      1936       2000          0     610413          0          2

Statement Average Statistics (E=PerExec, R=PerRow)
E    0               1936       2000          0     610413          0          2
R    0                968       1000          0     305206          0
```

What in the world could a statement be doing that requires over 300,000 LIO calls for each row it returns?[13] The execution plan shows that the query uses LIOs extremely inefficiently because it filters so late.

[11] We came across this example in front of a group of about 20 people in a client site classroom. Our client had asked us to apply our Hotsos SQL optimization method [Millsap, Holt 2001] on a live system that we'd never seen before in front of an audience of application developers. The goals were simultaneously to (1) relate our method to the clients' developers in their own environment, and (2) demonstrate that the method works on examples other than those we had contrived for our notes. This statement stuck out immediately in our diagnostic queries of *v$sql* because of its extraordinary LIO inefficiency (more about that later).

[12] The output you see here is generated by an early version of the Hotsos Profiler, distributed at *hotsos.com* [Holt 2000a]. The Hotsos Profiler is a tool that is akin to Oracle's *tkprof*, but the Hotsos Profiler contains significantly more features for the performance analyst.

[13] Actually, the problem at the client site was even worse, on the order of 6 million LIOs per row returned. What you see here is a facsimile of the client's situation, which we have re-created on our own server with a less demanding set of underlying data.

```
   Rows  Operation
-------  ---------------------------------------------------------------------
      2  SELECT STATEMENT  -  -
      2  NESTED LOOPS  -  -
 75,074   NESTED LOOPS  -  -
137,636    TABLE ACCESS BY INDEX ROWID - MTL_SYSTEM_ITEMS -
137,636     INDEX RANGE SCAN - MTL_SYSTEM_ITEMS_N1 -
212,708    TABLE ACCESS BY INDEX ROWID - CS_EC_PO_ITEM -
437,595     INDEX RANGE SCAN - CS_EC_PO_ITEM_N2 -
      2   TABLE ACCESS BY INDEX ROWID - CS_EC_CUST_PO -
 75,075    INDEX UNIQUE SCAN - SYS_C0013303 –
```

After optimization, the query's response time is reduced from 20.00 seconds to 0.04 seconds (a 500 times improvement), and its LIO count drops from 610,413 to 35 (a 17,440 times improvement),[14] yet of course it would have been impossible to improve the original statement's database buffer cache hit ratio.

```
SELECT /*+ full(rcecp) use_nl(msi) index(rcepi CS_EC_PO_ITEM_N1) */
   RCEPI.EC_PO_ID,
   RCEPI.SUPPORT_ITEM_ID,
   RCEPI.SUPPORT_PRICE,
   RCEPI.QUANTITY,
   RCEPI.SYSTEM_ID,
   RCEPI.CUSTOMER_PRODUCT_ID,
   RCECP.CUSTOMER_ID,
   RCECP.BILL_TO_SITE_ID,
   RCECP.SHIP_TO_SITE_ID,
   RCECP.CONTACT_ID,
   RCECP.PO_CREATE_DATE,
   RCECP.OMTX_ORDER_NUM,
   MSI.PRIMARY_UOM_CODE,
   NVL(MSI.SERVICE_DURATION,1) SERVICE_DURATION,
   MSI.SERVICE_DURATION_PERIOD_CODE
FROM
   CS_EC_PO_ITEM RCEPI,
   CS_EC_CUST_PO RCECP,
   MTL_SYSTEM_ITEMS MSI
WHERE
   RCEPI.CP_SERVICE_TRANSACTION_ID IS NULL
AND RCEPI.CUSTOMER_PRODUCT_ID IS NOT NULL
AND RCEPI.SUPPORT_ITEM_ID IS NOT NULL
AND RCEPI.EC_PO_ID = RCECP.EC_PO_ID
AND RCEPI.SUPPORT_ITEM_ID = MSI.INVENTORY_ITEM_ID
AND MSI.ORGANIZATION_ID = 21

SessionTotal 2  StatementTotal 2  PercentOfSession 100
FirstParsingUser 0  OptimizerGoal CHOOSE
HprofHash 913009341  OracleHash 42726173  RecursiveLevel 0

Statement Total Statistics
a  lcm    count       cpu    elapsed       disk      query    current       rows
- ---- -------- --------- ---------- ---------- ---------- ---------- ----------
P    1        1         2          2          0          6          0          0
E    0        1         0          0          0          0          0          0
F    0        2         0          2          2         29          4          2
- ---- -------- --------- ---------- ---------- ---------- ---------- ----------
T    1        4         2          4          2         35          4          2

Statement Average Statistics (E=PerExec, R=PerRow)
E    1                  2          4          2         35          4          2
R    0                  1          2          1         17          2
```

Here is the new execution plan that we forced into existence with the appropriate hints.

[14] At the client site, the LIO count dropped from 6,314,980 LIOs per execution to 42. Statement elapsed time dropped from more than two minutes to less than 0.04 seconds.

```
   Rows  Operation
-------  -----------------------------------------------------------------------
      2  SELECT STATEMENT  -  - 179
      2   NESTED LOOPS  -  - 179
      3    NESTED LOOPS  -  - 165
      6     TABLE ACCESS FULL - CS_EC_CUST_PO - 1
      7     TABLE ACCESS BY INDEX ROWID - CS_EC_PO_ITEM - 2
     10      INDEX RANGE SCAN - CS_EC_PO_ITEM_N1 - 1
      2    TABLE ACCESS BY INDEX ROWID - MTL_SYSTEM_ITEMS - 1
      4     INDEX UNIQUE SCAN - SYS_C0013295 -
```

This single query was executed so many times each day that it represented almost 50% of the total workload on the system. After the optimization, the query's total daily workload was inconsequential. Repairing the statement while at the same time diagramming the problem and explaining a generalized method for finding and repairing such problems took about two hours. Two hours of labor rendered a server upgrade unnecessary,[15] but the event would never have occurred if we had judged the original statement by its database buffer cache hit ratio.

Any economically efficient approach to optimizing Oracle systems almost always includes a search-and-destroy mission that targets high-LIO SQL. Approaching optimization this way often accomplishes significant gains in conserved hardware capacity, reclaimed at a rate far faster than you could possibly afford to buy additional hardware.

High Hit Ratio Conceals Inefficient Program Structure

Consider the following example in which a "good" database buffer cache hit ratio conceals a severe program inefficiency. In this case, a high database buffer cache hit ratio is caused by an inefficiency that drives up the LIO count in the numerator of the ratio. If you were to use the database buffer cache hit ratio for this session as a key performance indicator, then you would probably determine falsely that the session was well optimized.

Example: On a test system, we traced a full-table scan of a 524,288-row table executed from a SQL*Plus session. The following profiler output shows that the database buffer cache hit ratio to be what most people would probably consider very high, and therefore very good. It is 1 – 714/(262,858 + 7) = 99.728%.

```
select * from abc

SessionTotal 262873  StatementTotal 262865  PercentOfSession 99.997
FirstParsingUser 19  OptimizerGoal CHOOSE
HprofHash -2076482706  OracleHash 1864957478  RecursiveLevel 0

Statement Total Statistics
a  lcm    count       cpu    elapsed       disk      query    current       rows
- ---- -------- --------- ---------- ---------- ---------- ---------- ----------
P    0        1         0          0          0          0          0          0
E    0        1         0          0          0          0          0          0
F    0   262145      3048       3073        714     262858          7     524288
- ---- -------- --------- ---------- ---------- ---------- ---------- ----------
T    0   262147      3048       3073        714     262858          7     524288

Statement Average Statistics (E=PerExec, R=PerRow)
E    0               3048       3073        714     262858          7     524288
R    0                  0          0          0          0          0
```

This statement executes a full-table scan of a table called *abc*. The full-table scan obtained 524,288 rows by executing 262,145 database fetch calls. Each fetch call returned an average of only two rows (*rows*/*count* ≈ 2). Any fetch call to the database must execute at least one LIO to pin the Oracle block from which it will read row data. Here is what each fetch call indicated above did to the database:

LIO the appropriate Oracle block, which pins the block in the database buffer cache.
Obtain a row from the block.

[15] Think about how much wasted CPU and memory capacity were conserved by dropping the LIO count from 6,314,980 to 42.

```
If the block contains an additional row, then
    Obtain a (second) row from the block.
Otherwise,
    Release the pin on the prior block.
    LIO the next Oracle block in the table, pinning it in the database buffer cache.
    Obtain a row from the block.
Return the two rows to the caller of the fetch() function
```

The table contains about 730 rows per block (524,288/714), so we should expect most fetch calls to execute only one LIO call (the fact that there are so many rows in each block means that most two-row fetches will be fulfilled without needing to access a second block). The statistics bear this out (262,858/262,145 ≈ 1). Furthermore, only 714 of the 262,145 fetch calls executed ever needed to visit an Oracle block that wasn't already pre-loaded by a fetch call that came before it. Consequently, this session will produce an extremely high database buffer cache hit ratio.

The burning question is this: Why did someone execute a full table scan of *abc* by doing fetches of only two rows at a time? In this example, we contrived the effect of inadequate use of array fetches with the explicit setting of *arraysize* = 1 in SQL*Plus.[16,17] If the session's database buffer cache hit ratio were our principal performance metric, we would assume falsely that this session had executed efficiently.

The next profiler output shows the same statement executed from within SQL*Plus with *arraysize* = 500. The result is a much more efficient session that runs five times more quickly and executes about 150 times fewer LIO calls for fetch actions. However, the session's buffer cache hit ratio drops to only 1 – 710/(1,763 + 7) = 59.887% for the fetch actions of the session.

```
select * from abc

SessionTotal 1770  StatementTotal 1770  PercentOfSession 100
FirstParsingUser 19  OptimizerGoal CHOOSE
HprofHash -2076482706  OracleHash 1864957478  RecursiveLevel 0

Statement Total Statistics
a  lcm    count       cpu    elapsed       disk      query    current       rows
- ---- -------- --------- ---------- ---------- ---------- ---------- ----------
P    0        1         0          0          0          0          0          0
E    0        1         0          0          0          0          0          0
F    0     1050       228        509        710       1763          7     524288
- ---- -------- --------- ---------- ---------- ---------- ---------- ----------
T    0     1052       228        509        710       1763          7     524288

Statement Average Statistics (E=PerExec, R=PerRow)
E    0                228        509        710       1763          7     524288
R    0                  0          0          0          0          0
```

In this case, the higher buffer cache hit ratio of the inefficient SQL statement again demonstrates the unreliability of the ratio as a performance indicator. The ratio has concealed a problem that makes a system incrementally less capable of providing good response times to the business that owns it.

The problem with the database buffer cache hit ratio is a type of fallacy that plagues all ratios. Ratios are designed to conceal the magnitudes of their numerators and denominators. However, we cannot sense several types of workload inefficiencies unless we can see the very information that the ratio conceals. The "good" hit ratio produced by the inefficient session shown above yields a false positive indication. Had we listened to conventional knowledge about what the high database buffer cache hit ratio told us, we would perhaps never have investigated the inefficiency introduced by using the small fetch array size.

[16] It is a SQL*Plus curiosity that using *arraysize* = 1 actually causes fetches of two rows at a time. In this test, only the first fetch returned exactly one row. All other fetches except for the final one returned two rows apiece. The final fetch returned zero rows.

[17] The author thanks and acknowledges Jonathan Lewis for pointing out this additional "distortion factor" recently over a sequence of dinner conversations in Copenhagen.

Hit Ratio Fails to Indicate Proper Situation Response

The following example taken from a customer situation in Europe shows how much time and money you can waste by chasing the false promise of a better database buffer cache hit ratio.

> *Example:* A popular company in Copenhagen runs an order management application atop an Oracle database. For several weeks, the company experienced severe system performance problems. The worst problems occurred each Monday morning, when the company's telesales department always took the majority of its orders for the week. Order entry clerks would apologize to their customers for the slow system and wait uncomfortably for it to respond to their orders.
>
> An outside firm responsible for proactive system performance monitoring had observed each week for several weeks that the database buffer cache hit ratio seemed "too low." Each week for several weeks, the performance engineers had recommended increasing the value of *db_block_buffers*, and each week the manufacturer had responded by doing so. Each week, the database buffer cache hit ratio would rise, but the system performance did not improve.
>
> By the time the company phoned my dear friend Mogens Nørgaard, at that time the leader of Oracle Denmark's Premium Services group, the company had fed a gigabyte of memory to its database buffer cache, yet the performance problem was just as bad as when the problem began. It was a happy visit for everyone involved. Within a few minutes of Mr. Nørgaard's arrival, he was able to prove that the slow sessions on the system were waiting for the Oracle kernel event called 'free buffer waits'. Shortly thereafter, the company's system management staff were able to cut the bottleneck at its root cause (the DBWR was writing too slowly to keep up with the order workload), and they returned the wasted memory from the database buffer cache to someplace where it was needed.

Increasing the size of the database buffer cache hadn't helped at all, nor was it ever going to help. Measurement of the database buffer cache hit ratio was a completely unreliable diagnostic tool for this type of performance problem.[18]

Why Database Buffer Cache Hit Ratio is Popular

The database buffer cache hit ratio is a reliable performance diagnostic indicator *only if* you actually have less than a suitable amount of memory allocated to the database buffer cache. The problem with statements about indicators that say, "this indicator is reliable *only if* the problem is *x*," is that we must have *a priori* knowledge of what the problem is in order to know whether the indicator is reliable. Of course, if we knew what the problem was in the first place, then we wouldn't need the indicator.

The database buffer cache hit ratio is popular probably because so many people have had performance problems induced by undersized database buffer caches. Unfortunately, the default *init.ora* parameter values distributed with Oracle are far too small for many companies' systems. Therefore, a lot of companies using Oracle for the first time do experience problems with their database buffer cache size. Companies who are large enough to hire Oracle system performance consultants are especially likely to have busy enough systems to require modification of the default *db_block_buffers* value.

In Oracle Release 8.1.6, the recommended values for *db_block_buffers* are:

```
db_block_buffers = 100                                             # SMALL
# db_block_buffers = 550                                           # MEDIUM
# db_block_buffers = 3200                                          # LARGE
```

The Oracle instances my colleagues and I have visited in the past half-decade have almost exclusively required tens of thousands of database block buffers to provide efficient database services to their applications.[19] However, it is probably best for Oracle for their recommended numbers to be so small, because most Oracle customers have very

[18] There was probably some other ratio that could have been measured on the system that would have pointed to the root cause. However, my aim is to discourage you from thinking about which ratio it might have been. As we'll discuss soon, there's a far better way to diagnose system performance problems than to seek for meaningful diagnostic ratios.

[19] The types of sites we visit typically manage several hundred gigabytes of data for hundreds to tens of thousands of concurrent users.

small systems. If the Oracle *db_block_buffers* recommendations were significantly larger, then Oracle Support would probably receive a much larger volume of calls from frustrated customers who had been unable to execute their very first instance startup.[20]

Certainly, having an inadequate number of buffers in the database buffer cache can cripple a system's response times. But, customers need to carefully scrutinize their own situation using some of the ideas presented in this paper before they accept the implications of statements like the following at face value: "Increasing the buffer hit ratio from 95 to 99 percent can yield performance gains of over 400 percent" [Niemiec 1999 (7)]. Assuming that all is well because you see a high hit ratio is a mistake. Similarly, assuming that your system requires an expensive consultant for a few 70-hour weeks every time you see your hit ratio fall below 99% is also a mistake.

The database buffer cache hit ratio is probably most useful early in your system's life cycle when you are trying to determine adequate values for your key *init.ora* parameters. Checklists of ratios are excellent tools for *configuring* a database. However, checklists become highly inefficient when it comes time to *diagnose* a specific performance problem. What you need when you're called upon to diagnose and repair a system performance problem is a method with a single entry point that focuses your attention immediately upon the source of the problem. That is what the remainder of this paper is about.

Diagnosing Performance Problems Efficiently

Before you picked up this paper, you may have relied upon the database buffer cache hit ratio as a key performance indicator. If this is the case, then following my impassioned instructions to disregard the metric of course creates a void. If not the database buffer cache hit ratio, then what should you use instead? Welcome to Science World, the place where my Hotsos staff and I spend every moment of our professional lives preparing for your arrival.

Diagnosing a Specific Session

If you know which sessions are performing poorly, then the course of action is simple. The answer is to focus on session response times and determine what causes delays. The information to do this is found in a session's 10046 level-8 or level-12 trace file [Nørgaard 2000], which is tallied in summary form within the pseudotable *v$session_event*. The information produced by these two sources, if gathered at intelligently selected times, shows you exactly what you need to know about a session to optimize it *in the most economically efficient manner possible*. Furthermore, the timing information gives you the ability to *measure the performance improvement opportunity* in a way that is impossible to do with ratios.

Example: Look at the two resource profiles shown below. Among laypeople I've queried, I've found that it is practically impossible to misdiagnose the performance problems revealed by these two tables.

```
Resource Profile (Sparky v1.3β)
Interval duration, scope: 69.00s, sid=11 sqlplus@cm27785-a (TNS V1-V2)
t0, file: Sun Feb  4 11:44:28 2001 + 0.000s, 11-981308668.d
t1, file: Sun Feb  4 11:45:37 2001 + 0.000s, 11-981308737.d

Oracle Kernel Event                                 Event Duration   Pct
--------------------------------------------------  --------------  ----
enqueue                                                     64.34s   93%
SQL*Net message from client                                  2.25s    3%
session cpu                                                  0.02s    0%
--------------------------------------------------  --------------  ----
Total                                                       66.61s   97%
```

In the first case, a session has spent 93% of its total elapsed time waiting for a lock. If we eliminate the enqueue contention from this session's execution, we will reduce its 69-second duration to less than 5 seconds (69.00 – 64.34). Which enqueue? The session's 10046 level-8 trace shows you exactly which type of lock was queued for and in what mode the lock was held. Of course, you could also have discovered the enqueue contention had you looked at *v$lock* at the appropriate time, but how would you

[20] Systems with small memory configurations would fail to allocate a large enough shared memory segment for a big-cache Oracle instance to start.

have known to look there? Admittedly, this situation could have been detected by proper examination of some ratio on the system, but which ratio? Actually, since it can be shown that any ratio suffers from one or more conditions that render it unreliable, you would have needed to examine at least *two* ratios to detect this problem. What are the chances that you would have examined the right two or more ratios at exactly the right times?

```
Resource Profile (Sparky v1.3β)
Interval duration, scope: 44.00s, sid=7 sqlplus@cm27785-a (TNS V1-V2)
t0, file: Sun Feb  4 10:54:00 2001 + 0.000s, 7-981305640.d
t1, file: Sun Feb  4 10:54:44 2001 + 0.000s, 7-981305684.d

Oracle Kernel Event                                 Event Duration   Pct
--------------------------------------------------  --------------  ----
free buffer waits                                           26.31s   60%
db file scattered read                                       3.98s    9%
db file sequential read                                      2.00s    5%
--------------------------------------------------  --------------  ----
Total                                                       32.29s   73%
```

In the second case, a session has spent at least 60% of its execution time awaiting 'free buffer waits' waits [*sic*]. A similar report is what enabled Mr. Nørgaard to diagnose our friends' problem in Copenhagen (described earlier) in a matter of just a few minutes. Again, there does exist some combination of ratios that would also have revealed this problem, but the fact is that the outside firm's performance engineers never found it over the course of a several weeks-long performance problem.

A properly computed resource profile is a reliable, single data source that pinpoints exactly what an Oracle session is doing. The jobs of computing accurate Oracle resource profiles and teaching people what to do with them are what I do for a living.

Finding the Right Session to Diagnose

I have found that users are delighted to tell you exactly what you need to know in order to identify a specific session that requires performance optimization. However, if you don't have efficient access to information about which sessions are responsible for the largest system inefficiencies, then it's reasonably simple to find them.

Several Oracle monitoring tools offer the capability to view "top SQL"-style reports, based on the pseudotable *v$sql*. Few tools sort the list by *statement efficiency*.[21] Statement efficiency is defined as the number of Oracle LIO calls executed by a given statement, divided by the number of rows returned by the statement. Statements that execute thousands or millions of LIOs per row returned are candidates for closer examination.[22]

Like any ratio, statement efficiency is—in the strict technical sense of the word—an unreliable performance metric. For example, perhaps a statement returns no rows (yielding a division by zero instead of a statement efficiency). Or perhaps the statement returns only one row, but that row legitimately requires that every single block in a large table be visited (perhaps the result of a *count* or *sum* function call). However, this simple ratio, coupled with a common-sense understanding of what small denominator values might mean, can help the Oracle performance analyst quickly determine which session offers the greatest opportunity for performance optimization.

Conclusion

I've characterized the database buffer cache hit ratio several times now as "unreliable." I have selected this term carefully from a scientific vocabulary. I use the word *reliable* in this context to mean that using the specified metric either (1) *always* indicates a valid course of action, or (2) *never* indicates a valid course of action [Millsap 1987 (33)]. Unreliable metrics have the annoying attribute that using them sometimes produces valid results, but not always; and furthermore, you can't tell by looking at the metric whether it's producing a valid result this time or not.

[21] The author extends his gratitude to Jeff Holt, the Hotsos chief scientist, for the development and refinement of ideas concerning statement efficiency as a valuable performance metric.

[22] The *hotsos.com* tool *htopsql.sql* [Holt, Millsap 2000] uses statement efficiency as a sorting criterion to guide the Oracle performance analyst's attention first to the SQL statements with the greatest opportunity for high return on investment.

The best you can hope for from your database buffer cache hit ratio calculations is that you will find values that are so incredibly high (say, 99%) that it is almost inevitable that you have grossly inefficient SQL gobbling up your system's LIO capacity. To a handful of the world's most efficient Oracle system optimizers, hearing about performance problems on systems with database buffer cache hit ratios in excess of 95% starts them looking for workload inefficiencies. However, even this correlation is not reliable, because even extraordinary high hit ratios can be generated by databases with legitimately optimized SQL workload.

The following points summarize my story about the database buffer cache hit ratio.

1. A "really great" database buffer cache hit ratio is not necessarily good news. *A hit ratio in excess of 99% often indicates the existence of extremely inefficient SQL that robs your system's LIO capacity.*
2. Don't look to your database buffer cache hit ratio to determine whether your system is running efficiently or not. *The ratio will not tell you the answer.*
3. If you find out that you have a system performance problem, don't expect to diagnose the problem by examining your database buffer cache hit ratio. *The ratio will not tell you the answer.*
4. To find out whether your Oracle system is performing well or not, find a way to measure response times. *The first thing you should do is ask your users.*
5. Using checklists and ratios is an inefficient and often ineffective way to optimize a system. *To optimize a system more efficiently, profile sessions that are experiencing performance problems; the profile will tell you exactly what the problems are and by exactly how much you can expect to improve the situation.*
6. Sometimes you won't have efficient access to user information about which sessions are slow. *In that case, analyze your system's SQL statements in descending order of LIO calls generated divided by rows processed. Choose the session responsible for the most inefficient statement as your first target.*

Acknowledgments

My sincere thanks go to Jeff Holt, my Hotsos chief scientist, for his contribution to the scientific value that I believe *hotsos.com* is beginning to make in the Oracle world. Jeff has patiently and faithfully executed experiment after experiment in response to my relentless questioning. His experimental proofs and disproofs constitute the hearty spine of a new economically efficient method for Oracle system performance optimization.

Science World has been made possible by Juan Loaiza at Oracle Corporation, who created the idea to instrument the Oracle kernel with event-based performance diagnostic data. Anjo Kolk, with his famous "YAPP paper" [Kolk et al. 1999], was the first to write about how to apply those diagnostic data to the task of optimizing an Oracle system. We also owe a debt of thanks to Sameer Patkar and Roderick Mañalac, who encouraged Anjo to write about the subject; and Shari Yamaguchi and Jim Viscusi, who helped him do it. Thanks go to my dear friend Mogens Nørgaard of Miracle A/S, who to the best of my knowledge was the first man on Earth to force his entire staff to use *only* the Oracle wait interface to optimize Oracle systems. Without the insight of these gentlemen, we'd probably all still be spending 70 hours a week using ratio-based checklists to tune our Oracle systems.

I would also like to thank Jonathan Lewis, David Ensor, Willis Ranney, Virag Saksena, Bjørn Engsig, Johannes Djernæs, Dan Tow, Dominic Delmolino, Graham Wood, and Craig Shallahamer for discussions spanning backward in some cases for a decade that have contributed to the formation of the ideas I have attempted to convey in this document.

Finally, I appreciate the enormous commitment in time and effort that Rich Niemiec of TUSC has put forward to create his popular and useful 894-page tuning reference. I hope that Rich will accept my criticisms of parts of his work in the spirit of professional objectivity in which they are intended.

References

J. Holt. 2000a. *Hotsos Profiler*, an Oracle event 10046-aware trace file analyzer. Available at http://www.hotsos.com.

J. Holt. 2000b. *Predicting Multi-Block Read Call Size*. Hotsos. Available at http://www.hotsos.com.

J. Holt, C. Millsap. 2000. *Hotsos Clinic Tools Pack*. Hotsos. Available at http://www.hotsos.com.

A. Kolk, S. Yamaguchi, J. Viscusi. 1999. *Yet Another Performance Profiling Method (or YAPP-Method)*. Redwood Shores CA: Oracle Corp. Available at http://www.dbatoolbox.com/WP2001/tuning/O8I_tuning_method.htm.

J. Lewis. 2001 *Practical Oracle8i: Building Efficient Databases.* Upper Saddle River NJ: Addison-Wesley.

C. Millsap. 1987. *Software Testing Strategies for Automatic Test Data Generators*. Boulder CO: University of Colorado. Master of science thesis.

C. Millsap, J. Holt. 2001. *Hotsos Clinic on Oracle® System Performance Problem Diagnosis and Repair*. 2½-day educational event: Hotsos. Information available at http://www.hotsos.com.

J. Morle. 2000. *Scaling Oracle8i: Building Highly Scalable OLTP System Architectures.* Reading MA: Addison-Wesley.

R. Niemiec. 1999. *Oracle Performance Tuning Tips & Techniques.* Berkeley CA: Osborne/McGraw Hill.

M. Nørgaard. 2000. *Introducing Oracle's Wait Interface*. Hotsos. Available at http://www.hotsos.com.

G. K. Vaidyanatha, K. Deshpande, J. A. Kostelac. 2001. *Oracle Performance Tuning 101*. Berkeley CA: Osborne/McGraw Hill.

About the Author

Cary Millsap is a manager of Hotsos Enterprises, Ltd., a company dedicated to improving the self-reliance of Oracle system performance managers worldwide through classroom education; information, software, and services delivered via www.hotsos.com; and consulting services. He is the inventor of the Optimal Flexible Architecture, creator of the original APS toolkit, a *hotsos.com* software designer and developer, editor of *hotsos.com*, and the creator and principal instructor of the *Hotsos Clinic on Oracle® System Performance Problem Diagnosis and Repair*.

Mr. Millsap served for ten years at Oracle Corporation as a leading system performance expert, where he founded and served as vice president of the System Performance Group. He has educated thousands of Oracle consultants, support analysts, developers, and customers in the optimal use of Oracle technology through commitment to writing, teaching, and speaking at public events. While at Oracle, Mr. Millsap improved system performance at over 100 customer sites, including several escalated situations at the direct request of the president of Oracle. He served on the Oracle Consulting global steering committee, where he was responsible for service deployment decisions worldwide.

Revision History

February 2001: Original revision produced to accompany live presentation at *IOUG-A Live! 2001*.

October, December 2001: Minor revisions.