Why You Should Focus on LIOs Instead of PIOs

Cary Millsap · *Method R Corporation*

Executive Summary

Many Oracle educators teach that reducing the number of PIO calls should be the top priority of SQL optimization. However, in our field work, we commonly eliminate 50% or more of the response time from slow Oracle applications, even after they've been tuned to execute *no* PIO calls. The secret is that Oracle LIO calls are more expensive than many people understand. In this paper, I explain the following research results:

- LIO processing is the number-one bottleneck for many business processes today, even on systems with "excellent" database buffer cache hit ratios.
- Excessive LIO call frequency is a major scalability barrier, because LIOs consume two of the system's most expensive resources: CPU and latches.
- Even if you could have an infinite amount of memory and achieve a perfect 100% database buffer cache hit ratio, your system will be inefficient and unscalable if it executes more Oracle LIO calls than it needs to.
- The statistics that database administrators commonly track can lead you to believe that PIO processing is your bottleneck when it's not. *Do not* increase disk or memory capacity until after you determine the impact of PIO latency upon your end-user response times.
- If you will focus on LIO reduction from the very beginning of a SQL optimization task instead of PIO reduction, then you will usually eliminate most of your PIOs by side-effect, because most of your PIOs are motivated by LIO calls in the first place.

Reading from the Buffer Cache is More Expensive than You Might Think

Oracle analysts routinely assume that converting disk reads to memory reads is the key to excellent performance. Memory access is certainly faster than disk access, but how much faster? One popular text claims that "retrieving information from memory is over 10,000 times faster than retrieving it from disk" [Niemiec (1999) 9]. This number makes sense if you compare advertised disk access latencies with advertised memory access latencies:

Typical disk access latency	0.010 000 seconds (10 milliseconds)
Typical memory access latency	0.000 001 seconds (1 microsecond)
Relative performance	Memory is $10^{-2} \div 10^{-6} = 10^{4}$ = 10,000 times faster

Perhaps reading from memory actually is 10,000 times faster than reading from disk. But it is a mistake to infer from this hypothesis that reading an Oracle block from memory (the database buffer cache) is 10,000 times faster than reading an Oracle block from disk. Data from Oracle trace files reveal the truth to be considerably different.

Table 1 (page 12) lists 71 Oracle trace files uploaded to *www.hotsos.com* by participants in the Hotsos Profiler beta program.[1] In total, these files represent about 2.6 billion Oracle block accesses and over 21 million operating system read calls. For this sample, fetching an Oracle block from the database buffer cache is only about 37 times faster than fetching an Oracle block from a database file.

[1] In April 2008, Cary Millsap and his team of software developers who had developed the Hotsos Profiler left Hotsos to create a new company called Method R Corporation, which owns and maintains the software now called the Method R Profiler.

Average latency of reads from disk	0.000 053 seconds + 0.001 913 seconds = 0.001 966 seconds
Average latency of reads from buffer cache	0.000 053 seconds
Relative performance	Retrieving from buffer cache is 0.001 966 ÷ 0.000 053 ≈ 37 times faster than retrieving from disk

Retrieving an Oracle block from the buffer cache is *not* 10,000 times faster than retrieving it from disk. Examination of real Oracle performance data reveals that the true factor is typically far less than 100. This is of course absurdly distant from that factor-of-10,000 figure that you might have expected. As you will see, you can save money and time by understanding how to measure the actual cost of LIOs and PIOs.

Oracle LIO and Oracle PIO

The analysis begins by understanding the definitions of *LIO* and *PIO*.

- An *Oracle LIO* (or *logical read call*) occurs any time the Oracle kernel requests access to an Oracle block in the database buffer cache.[2] If the kernel cannot find a specified Oracle block in the database buffer cache, then the LIO motivates an Oracle PIO call. In accounting for LIO elapsed times, we do not include the time spent executing the PIO portion of the LIO call. The number of LIO calls for a session is the sum of the values in `v$sesstat` for the statistics named “db block gets” and “consistent gets.”

- An *Oracle PIO* (or *physical read call*) occurs when the Oracle kernel executes an operating system (OS) read call for an Oracle database block. In accounting for PIO elapsed times, we include only the time that the Oracle kernel spends waiting for the execution of the OS read call. A PIO may or may not engage the service of an actual disk drive; if the OS can fulfill the read call from memory (e.g., disk controller cache or OS buffer cache), then the PIO will be satisfied without actually incurring the latency of a read from disk. For a single OS call that reads multiple blocks, each block counts as one PIO. In accounting for PIO elapsed times, we do not include the time spent executing the LIO that motivated the PIO. The most common type of PIO is counted in the `v$sesstat` value for the statistic named “physical reads.”

Shortly, this paper provides a Perl program that will help you measure actual LIO and PIO latencies on your system.

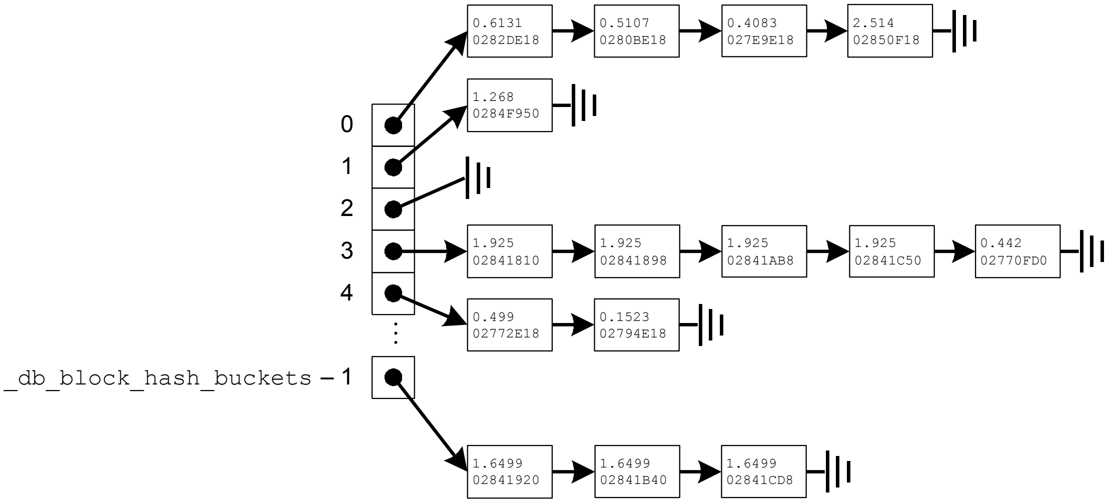
An Oracle LIO Is Not Just “a Memory Access”

An Oracle LIO is far slower than the speed of a single memory access. A typical LIO requires a few dozen to a few hundred microseconds to execute. The reason is that an Oracle LIO call executes many CPU register and memory operations, several of which are subject to system-wide serialization (single-threading). Here is a segment of pseudocode giving an idea of how much work the Oracle kernel has to do in order to fulfill a LIO call:

[2] There are several occasions during which Oracle manipulates database blocks *without* using the database buffer cache. See [Adams 2000a].

```
function LIO(dba, mode, ...)
    # dba is the data block address (file#, block#) of the desired block
    # mode is either 'consistent' or 'current'
    address = buffer_cache_address(dba, ...);
    if no address was found
        address = PIO(dba, …);                # potentially a multi-block pre-fetch[3]
    update the LRU chain if necessary;        # necessary less often in 8.1.6
    if mode is 'consistent'
        construct read-consistent image if necessary, by cloning the block and calling LIO for the appropriate undo blocks;
        increment 'cr' statistic in trace data and 'consistent gets' statistic in v$ data;
    else (mode is 'current')
        increment 'cu' statistic in trace data and 'db block gets' statistic in v$ data;
    parse the content of the block;
    return the relevant row source;
end
```

Evaluating what we're calling the *buffer_cache_address* function is the key piece of missing information about Oracle internals that allows so many analysts to underestimate the true cost of an Oracle LIO. This function uses a hash table in Oracle shared memory to determine whether a specified Oracle block is resident in the database buffer cache. If it's not, then of course a user's Oracle process will be required to retrieve the block via an OS read call. The hash table contains elements called *buffer headers* arranged in chains called *cache buffers chains*, each of which is addressable by a hash key. The hash table looks something like this:



To determine whether a given block is in the database buffer cache, the Oracle kernel computes a hashing function using the desired data block's address (or *dba*, which consists of the file number and block number).[4] The resulting index uniquely identifies a chain in the hash table. Oracle knows that a block resides in the database buffer cache if and only if the block's buffer header exists in that specific chain. So by searching this cache buffers chain, Oracle can determine whether the desired block is in the buffer cache. If the kernel finds the desired dba in the chain, then Oracle accesses the block via the address that is also stored in the buffer header. If the kernel does not find the desired dba in the chain, then it knows it must first read the block (via a PIO call) to put it into the buffer cache before continuing the LIO.

Once a kernel process has computed a block's address in the buffer cache, the cost of parsing and manipulating that block's content is nontrivial. In fact, the actual *use* of Oracle block content is the number one consumer of CPU capacity on any reasonably well optimized Oracle system. Your average LIO latency will vary, depending primarily upon your CPU speed and the total number of machine instructions required to parse the content of an Oracle block.

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

[4] For more information about hashing, see [Jenkins 1997] and [Knuth 1973 (506–549)].

For example, an LIO for a full 8KB index leaf block will take longer than an LIO for a 4KB data block that has two small six-column rows in it. The data in Table 1 show that the LIO portion of an Oracle block manipulation is more expensive than the PIO portion for several of the files (the PIO portion is, of course, almost universally accepted as "intolerably expensive").

Latch Serialization Impacts LIO Latency

The cache buffers chains reside in shared memory, where potentially thousands of Oracle processes expect the ability to read and write them concurrently. Of course, to prevent corruptions of the hash table, the Oracle kernel designers had to implement a serialization mechanism to prevent two or more Oracle kernel processes from making conflicting modifications to one hash chain at the same time. As a result, the Oracle kernel serializes its own accesses to the chain by using a *latching protocol*. Before a kernel process can modify or even search a cache buffers chain, the process must first acquire the *cache buffers chains latch* for that chain. In Oracle releases prior to 8.1, Oracle uses one cache buffers chains latch per chain. Beginning with release 8.1, one cache buffers chains latch can cover multiple chains.[5]

Throughout all Oracle8 releases, only one Oracle process can hold a cache buffers chains latch at a time. If a process attempts to acquire a latch that is not available (because it is currently held by another process), then the requesting process must wait until the latch becomes available. Thus, by holding a latch, a writer or even a reader of a cache buffers chain will block all other prospective readers and writers of every chain protected by that latch.[6] We understand that in release 9 a latch holder who only reads a cache buffers chain can share access to its latch with other readers. This kernel design optimization should provide relief from cache buffers chains latch contention induced by concurrent readers. However, a writer holding a latch will still block all other prospective readers and writers of chains protected by that latch. And, of course, readers will continue to block writers.

The way an Oracle kernel process waits for a latch is widely misunderstood. For single-CPU systems, the latch acquisition algorithm is very simple. On a single-CPU system, a process will voluntarily surrender its time slice if it tests the availability of a latch and determines that it is being held. If another process holds the desired latch, there is nothing productive that the requesting process can do except stand aside in hopes to allow the latch holder enough CPU time to complete its serialized work and release the latch.

For multi-CPU systems, the latch acquisition algorithm is more complicated:

```
function get_latch(latch)
    # multi-CPU implementation
    increment gets;
    if fastget_latch(latch) return true;
    increment misses;
    for try = 0 to +infinity
        for spin = 1 to _spin_count
            if fastget_latch(latch) return true;
        $t_0$ = current wall time;
        sleep for min(f(try), _max_exponential_sleep) centiseconds;
        $t_1$ = current wall time;
        increment sleeps and sleep[try];
        register 10046 level-8 event "latch free" for $ela = t_1 - t_0$;
end
```

[5] For an excellent article on this feature, see [Adams 2000b].

[6] In the context of this discussion, the *reader* of a chain is a process that searches the chain to test for the existence of a specified buffer header. The *writer* of a chain is a process that somehow modifies the content of the chain, for example, either by inserting into the chain or deleting from the chain.

```
function fastget_latch(latch)
    if test(latch) shows that latch is available
        if test_and_set(latch) is successful
            return true;
    return false;
end
```

If a process determines that a latch is unavailable, then on a multi-CPU system it is fair for the process to hope that the latch holder might be able to complete its serialized work on another CPU and release the desired latch within just a few microseconds. So, rather than endure the performance penalty of volunteering for a context switch right away (which would typically result in about a 10-millisecond sleep), the process will *spin* for the latch.

When a process *spins* for a latch, it executes a tight loop of repeated test operations upon the latch. If the latch becomes available during the spin, then the process will detect that availability immediately and acquire the latch. If the process does a *fastget* for *_spin_count* iterations without finding the latch available, then the process will voluntarily sleep for a specified number of milliseconds. By sleeping, the process notifies the operating system that it is volunteering itself for a context switch. When the sleep completes and the process is granted a new CPU time slice, the process will "try" again with another spin for the latch. If the second spin completes without making the acquisition, then the process will sleep again, and so on. As the number of tries for a latch acquisition increases, the sleep durations usually increase as well.[7]

The algorithm beneath our *buffer_cache_address* pseudocode function described earlier has some important performance vulnerabilities. Most obviously, the time it takes to search a chain for a block is proportional to the length of that chain—longer chains degrade performance. There are two ways that a chain can become longer. First, there's bad luck. Anytime there are fewer chains than there are buffers in the database buffer cache, there will necessarily be *collisions*, in which two or more data block addresses will hash to the same chain. You can see an example of this collision phenomenon within chain 0 in the example picture shown earlier, where blocks 0.6131, 0.5107, 0.4083, and 2.514 (these are shown in *file#.block#* format) all hash to the same chain. Second, even when there are as many chains as there are buffers,[8] the chains can still grow long. This phenomenon is a side-effect of how Oracle implements read consistency.

Here is how it happens. Whenever Oracle executes a read, there is a possibility that the block being read has been modified since the time when the query began. If the LIO is a *consistent mode* read, then the Oracle kernel takes the appropriate steps to revert its target block's contents to the appropriate point in history. When a process detects that a block's contents have changed since that process' query began, it creates a *clone* of that block in the database buffer cache. The session then applies the appropriate undo to the clone that is required to revert the block's content to the way it appeared at the appropriate point in history. The problem is this: all the clones of a given block will always hash to exactly the same cache buffers chain, because they all share a single data block address. On systems that generate a lot of block clones, there will necessarily be growth in the length of one or more cache buffers chains. You can see an example of this type of collision within chain 3 in the example picture shown earlier, where several clones of block 1.925 coexist on the same chain.

Long chains degrade end user response times in several ways. First, long chains increase the number of machine instructions required for chain searches. Second, longer search durations mean holding cache buffers chains latches for longer durations; therefore, processes trying to acquire a latch will tend to consume more CPU time spinning for that latch. Third, the resulting latch contention introduces sleeps from "latch free" waits into the response time. The new shared latch design of Oracle9*i* should help to reduce the negative impact of the response time consumed by spinning and sleeping, but *not* the time consumed searching the chain (and of course, *not* the time consumed simply processing the content of the Oracle data block).

[7] The dominant reasons that latch sleep durations recorded in a 10046 level-8 trace file may appear not to increase as the *try* count increases are: (1) coarse timer granularity can cause both latency overestimates and latency underestimates to appear in the trace file; (2) latch sleep durations are apparently Oracle port-specific.

[8] That is, when you have set `_db_block_hash_buckets` to a number greater than or equal to your setting of `db_block_buffers`….

How to Measure LIOs and PIOs Operationally

Measuring the performance impact of LIOs, PIOs, and cache buffers chains latch contention is straightforward with the following Perl program:[9]

```
#!/usr/bin/perl
# lio - summarize LIO and PIO statistics from 10046 level-8 trace file
# Cary Millsap
# (c) 2001, 2008 Method R Corporation

use warnings;
use strict;

my $ORACLE_RELEASE = 8;     # use 7, 8, or 9
my $CBCid = 66;
    # use "select latch# from v$latchname where name='cache buffers chains'"

my ($trcfile) = @ARGV or die "Usage: $0 trcfile\n";
open TRCFILE, "<$trcfile" or die "$0: can't open '$trcfile' ($!)";
my ($nLIO, $cLIO) = (0, 0);
my ($nPIO, $ePIO) = (0, 0);
my ($nSLP, $eSLP) = (0, 0);
while (<TRCFILE>) {
    if (/^(PARSE|EXEC|FETCH|UNMAP).*c=(\d+).*cr=(\d+),cu=(\d+)/i) {
        $cLIO += $2;
        $nLIO += ($3 + $4);
        # print "c=$cLIO, n=$nLIO: $_"; # for debugging, testing
    }
    elsif (/^WAIT.*nam='db file.*read' ela=\s*(\d+).*p3=(\d+)/i) {
        $ePIO += $1;
        $nPIO += $2;
        # print "e=$ePIO, n=$nPIO: $_"; # for debugging, testing
    }
    elsif (/^WAIT.*nam='latch free' ela=\s*(\d+).*p2=$CBCid/i) {
        $nSLP ++;
        $eSLP += $1;
        # print "e=$eSLP, n=$nSLP: $_"; # for debugging, testing
    }
}
if ($ORACLE_RELEASE >= 9) {
    $cLIO *= 0.000_001;
    $ePIO *= 0.000_001;
    $eSLP *= 0.000_001;
} else {
    $cLIO *= 0.01;
    $ePIO *= 0.01;
    $eSLP *= 0.01;
}
my $fmt0 = "%-35s %16s %15s %16s\n";
my $fmt1 = "%-35s %15.2fs %15d %15.6fs\n";
printf $fmt0, "Event", "Duration", "# Calls", "Dur/Call";
printf $fmt0, "-"x35, "-"x16, "-"x15, "-"x16;
printf $fmt1, "LIO call CPU time", $cLIO, $nLIO, $nLIO?$cLIO/$nLIO:0;
printf $fmt1, "PIO call elapsed time", $ePIO, $nPIO, $nPIO?$ePIO/$nPIO:0;
printf $fmt1, "Cache buffers chains latch sleeps", $eSLP, $nSLP, $nSLP?$eSLP/$nSLP:0;
close TRCFILE or die "$0: can't close '$trcfile' ($!)";
```

Note that the program shown here will overestimate LIO latency (1) for applications that consume a lot of CPU capacity executing parse calls, and (2) for applications that consume a lot of CPU capacity doing PL/SQL or SQLJ language processing. Of course, if your database spends a lot of time parsing, then you should try to reduce the number of parse calls that your application requires. If your database spends a lot of time doing language processing, then you should carefully consider moving that application logic to a middle tier, where scalable CPU capacity is much less expensive than it is in your database.

[9] See [Nørgaard 2000] for an introduction to producing the Oracle 10046 level-8 trace data that this program requires.

A Great Buffer Cache Hit Ratio Doesn't Mean Great Performance

One strong Oracle tuning tradition holds that the higher the database buffer cache hit ratio, the better the performance of the database. *If* you first minimize the number of LIO calls that an application makes, then you *can* generally improve response times if you use a bigger database buffer cache to convert LIO+PIO calls into bare LIOs. However, two popular ideas about the database buffer cache hit ratio are deeply flawed:

Not true: "A high buffer cache hit ratio corresponds to good system performance."

Not true: "The buffer cache hit ratio is a useful measure of SQL statement efficiency."

If the first conjecture were true, then we would expect for the database buffer cache hit ratios to be low for Oracle programs with performance problems. Yet, in our field work, we find that high cache hit ratios are not reliable indicators of good performance. For example, Table 1 represents 71 trace files uploaded to *www.hotsos.com* by customers with performance problems. The buffer cache hit ratios of these files are all over the map. Notably, in 37 of our 71 performance problem cases, the database buffer cache hit ratio already exceeds 99%:

Range	File count
0% ≤ *hit ratio* < 90%	18
90% ≤ *hit ratio* < 99%	16
99% ≤ *hit ratio* < 99.9%	27
99.9% ≤ *hit ratio* < 99.99%	7
99.99% ≤ *hit ratio* < 100%	3

The second conjecture is dangerous because it can lead people to perceive an extremely inefficient SQL statement as "well-tuned." Earlier this year, I published some common examples of SQL statements that become immensely faster and more efficient after executing steps that actually reduce each statement's buffer cache hit ratio [Millsap 2001]. The database buffer cache hit ratio is a good indication of how efficiently the Oracle instance is serving the SQL application sitting atop it in the stack. Unfortunately, the database buffer cache hit ratio can mislead you if your application itself is the source of the inefficiency.

I have found that performance problems are most often the result of excessive LIO counts. The philosophy of "higher hit ratios are better" has one very critical flaw at its core: the ratio appears the most *excellent* when a system executes an excessive number of LIOs. This is why the best Oracle performance analysts I've known since 1990 actually count on high cache hit ratios as reliable indicators of performance *problems*. I strenuously object to using a good database buffer cache hit ratio as a measure of good system performance, because it motivates system owners to purchase disk or memory upgrades in situations in which it would have been cheaper to reduce excessive LIO call counts.

PIOs Might Not Be Your Bottleneck

My aim in this paper is *not* to convince you that it's okay to have SQL statements in your application that generate too many PIOs. LIO+PIO calls *are* of course more expensive than bare LIO calls, for reasons including the following:

- Disks *are* slower than memory, especially if you configure those disks poorly or load them up with so much data that you exceed their effective throughput capacity (e.g., [Millsap 1996; Millsap 2000]).
- PIO calls do require some CPU service time (e.g., for memory-to-memory copying). There is speculation about how much CPU time a PIO requires. The number of machine instructions required to execute a PIO will vary depending upon your I/O subsystem architecture (e.g., whether you use raw or buffered I/O).
- During a PIO call, an Oracle kernel process executes an OS read call, which volunteers the process for a context switch. On a busy system with lots of competition for CPU service, each context switch can account for several milliseconds of elapsed time. (Of course, a likely reason for a system to be so busy in the first place is a wastefully large number of Oracle LIO calls.)
- Even when an Oracle PIO results in only a memory read (e.g., from an OS buffer cache), the Oracle kernel process doing the read will generally have to modify *two* cache buffers chains: (1) it will delete from the chain con-

taining the buffer header of least-recently-used block that will be evicted from the buffer cache, and (2) it will insert into the chain that will contain the buffer header for the newly acquired block.

However, recognizing when your PIO calls are actually *not your performance problem* can prevent you from buying disks or memory in situations where faster disks and more memory won't help your performance.

The data in Table 1 show that the average PIO latency across 71 trace files is only about *two milliseconds.* Considering typical advertised disk latencies, this statistic is remarkable. These days a 10-millisecond read is about average for sites with reasonably cheap disks. Sites with fast disks experience read latencies in the 5-millisecond range. Yet the average PIO latency is less than 2 milliseconds in 25 of our 71 situations, *all of which represent problem performance!* How is this possible?

There are several factors that can drive the average Oracle PIO latency below the physical disk latencies that we all expect. First, a PIO does not necessarily represent a read from disk. Many I/O subsystems today have cache at several architectural layers, including OS, controller, and disk cache. These caches are designed specifically to maximize the probability that an application's OS read call would be satisfied from memory, without having to wait for a retrieval from an actual physical disk. Ironically, applications with the most inefficient SQL execution plans—ones that revisit some set of blocks an enormous number of times—are the most likely applications to benefit from the latency-reducing benefits of these caches. But as we'll see shortly, feeding more cache to an inefficient execution plan is a dreadfully bad economic decision.

Second, when Oracle executes a multi-block read, the expensive I/O setup operations (the seek and rotational positioning operations) are shared across multiple blocks. For example, a 20-block read might consume only about four times as much elapsed time as a 1-block read. In this case, each block that came into the cache via the 20-block read will have an average latency that is only about 20% of the expected latency for a 1-block read.

In my career, I have visited many Oracle customers who had spent vast sums of money upgrading their disk I/O capacity with no perceptible improvement in user response times. Typically, these customers determined that, because their number of Oracle PIO operations was very high, buying either more disks or faster disks would provide a performance benefit. In many of these cases, however, their PIO latencies were so low to begin with that there was no way these customers would reduce the latencies further, no matter how much hardware expenditure they allocated. In several cases, the total contribution of PIO calls to user response time was so negligible that even reducing PIO latencies by 100% would have created no perceptible difference in user response times.

Do you have a physical I/O bottleneck on your system? Chances are that if any of the following is true, then somebody at your business probably thinks that you do:

- Your disk utilization figures are high.
- Your disk queue lengths are long.
- Your average read or write latency is high.

If you suspect that you have a physical I/O bottleneck for any of these reasons, *do not* upgrade your disk subsystem until you figure out how much impact your Oracle PIO latencies have upon user response time. For example, what impact would a disk upgrade have upon the application with the following resource profile?

```
Oracle Kernel Event                                Duration           Calls      Dur/Call
---------------------------------- -------------------- --------------- -------------
CPU service                          48,946.72s   98.0%         192,072     0.254835s
db file sequential read                 940.09s    1.9%         507,385     0.001853s
SQL*Net message from client              60.90s    0.1%         191,609     0.000318s
latch free                                2.17s    0.0%             171     0.012690s
SQL*Net message to client                 1.32s    0.0%         191,609     0.000007s
SQL*Net more data to client               0.06s    0.0%             778     0.000077s
file open                                 0.01s    0.0%               7     0.001429s
---------------------------------- -------------------- --------------- -------------
Total                                49,951.27s  100.0%
```

The process behind this profile ran at one customer site for almost *14 hours*. The profile depicts an Oracle session that has executed 507,385 Oracle PIOs. From the PIO count alone, one might infer that a disk upgrade could save the system. However, look at the total response time attributable to PIOs: it's only 1.9% of the total execution time of the program. Imagine what could have happened…

> Dear Analyst:
>
> We all just wanted to thank you for your work in optimizing our system. Thanks to the $250,000 emergency disk upgrade you recommended to the Board, we have been able to slash our physical I/O processing time by about 50%. The resulting improvement to the response time of the key business process we were trying to optimize was a whopping nine tenths of one percent.
>
> Yours truly, …

The fact that there were over half a million PIOs generated by this session is irrelevant to improving its performance. The problem with this program is all CPU consumption. This program consumed almost 14 hours of elapsed time because it executed 1,342,856,404 LIOs, at approximately 0.000 036 seconds per call.[10] Those "memory accesses" are not so cheap when you execute 1.3 *billion* of them. The database buffer cache hit ratio, by the way, was an "outstanding" 99.9622%. The very fruitful plan of attack that our customer chose was to reduce the response time of the session to just a few minutes by modifying an execution plan to eliminate over a billion LIO calls.

Again, don't get me wrong; I'm not advocating that you take a cavalier attitude toward PIOs. But remember:

If your total time spent executing PIOs is only a small percentage of your total response time,
then don't waste your time trying to "tune" your PIO processing.

You'll see shortly that the best way to reduce most Oracle applications' response times is to eliminate Oracle LIOs, which also happen to eliminate PIOs by side-effect.

More Memory Is Often Not the Best Solution Either

Wouldn't it be nice if we could install an infinite amount of memory into an Oracle system? Then perhaps all of our performance problems would go away. Unfortunately, it's not true. Here's a proof. First, let's assume briefly that there is *no* time cost involved in loading an entire, huge database into a colossal database buffer cache (of course this assumption is not true, but for the sake of argument, let's say that it is). Now that all the Oracle data blocks we'll ever need are already in the buffer cache, we'll never need to visit a disk again (filthy spinning rust[11] anyway!).

Unfortunately though, we have already seen empirical evidence that even after eliminating Oracle PIO calls, the resulting "memory reads" are neither free nor particularly cheap. Applications that execute wastefully many Oracle LIO calls can consume hours of unnecessary CPU time, a few tens of microseconds at a time. Eliminating PIOs entirely can still leave us with an application that chews up a CPU for hours.

For memory upgrades, I offer the same advice that I mentioned earlier in our disk upgrade discussion. If you believe that you should increase the size of your database buffer cache to reduce your Oracle PIO call frequency, *do not* upgrade your memory until you figure out how much impact your Oracle PIO latencies have upon user response time. The same example we used above shows why. In this case, we could spend money to eliminate every single PIO call in the program. However, doing so would have reduced the program's response time only from 13.6 hours to 13.3 hours. If your LIO count is high, then instead of buying memory, try to eliminate Oracle LIOs. Get rid of the LIOs, and you'll eliminate PIOs by side-effect; PIOs go away if the LIOs that motivated them are eliminated.

How to Eliminate LIOs

When either CPU service or waits for "latch free" is a big part of your users' response times, then you probably have an LIO problem.[12] If you do, then the most economically efficient way to solve the problem is probably to optimize the SQL that causes too many LIOs. A high-LIO SQL statement running in one program at a time is a problem for one user who is waiting for the program's output. It's possibly a problem for other users as well, if they are competing for scarce CPU and latch resources with the high-LIO statement. A high-LIO statement running in dozens of concurrent programs is a catastrophic problem for everyone. In both problems, it is difficult to make any real pro-

[10] You can't determine this session's LIO count, LIO latency, or buffer cache hit ratio by looking only at the resource profile data that I have shown here. I used the Perl program shown earlier to compute those additional values. The number of calls shown in this table (192.072) is actually the number of parse, execute, fetch, and unmap calls described in the trace data.

[11] I acknowledge learning the term "spinning rust" from my friend Mark Farnham of Rightsizing, Inc.

[12] It is possible that you have a parsing problem. For discussions about parsing problems, see [Holt & Millsap 2000; Engsig 2001].

gress by masking the issue by manipulating memory or disks. The permanent long-term solution is to eliminate unnecessary LIOs.[13]

It's not so hard to do. Step one: focus your attention upon LIO reduction, not PIO reduction. Stop measuring performance by watching your database buffer cache hit ratio, especially for individual SQL statements. When you look at *tkprof* output, pay attention to the `query` and `current` columns.[14] When you look at `v$sql`, pay attention to the `buffer_gets` column. Don't stop optimizing if a query consumes more than 100 LIOs per row returned,[15] even if you've figured out how to reduce the PIO count for the query to zero. Lookups and simple joins should require fewer than 10 LIOs per row returned.

By eliminating LIOs first (before you try to reduce the PIO count), you will save time and money, because you'll be reducing the fundamental unit of work in Oracle from which most resource consumption derives. Remember, a program can have a PIO count of *zero* and still run for hours. But a program with a low LIO count will consume only a small amount of CPU service, a small number of latch acquisitions, and of course a necessarily small number of PIOs as well, because a small number of LIOs will nearly always motivate only a small number of PIOs. Only after you have optimized the LIO requirement of an Oracle program should you consider a memory of disk upgrade.

The query optimization software in the Oracle kernel gets smarter with every release (for an interesting article on this subject, see [Gorman 2001]). But still, the Oracle performance analyst will encounter systems in which the number one cause of performance trouble is poor query optimization. Some of the more common mistakes that we see in the field include:

- *Executing unnecessary business functions.* There is no more efficient or effective system optimization technique than to determine that a business is expending scarce system resources to produce output that either nobody looks at, or output whose business function can be suitably replaced by less expensive output. Performance analysts who isolate themselves from the functional requirements of a business application also isolate themselves from opportunities to eliminate unnecessary business functions. To optimize workload, you must understand the business of that workload.

- *Carrying massive amounts of row source data through several stages of a complicated execution plan, only to filter out most of the rows in a late stage of the plan.* Use hints and stored outlines if you must, but force Oracle to filter data as early in an execution plan as you can.

- *The idea that all full-table scans are bad.* Full-table scans are efficient in their place; use them when they are more efficient than index-driven plans. If you are in doubt about whether a full-table scan is efficient, *test it*.

- *The idea that all nested loops plans are good.* Nested loops execution plans are notorious for driving "excellent" database buffer cache hit ratios, because they tend to revisit the same Oracle blocks over and over again. Replacing nested loops plans with other joins (full-table scans driving either hash joins or sort-merge joins, for example) can sometimes result in spectacular performance improvements.

- *Failure to use array processing.* Fetching data one row at a time also drives "excellent" database buffer cache hit ratios, because they also tend to cause Oracle to revisit the same blocks over and over again. Using Oracle array operations will reduce not only LIO frequencies, but often enormous amounts of network traffic as well.

Conclusion

In system optimization and capacity planning exercises since 1990, several of my colleagues and I have created reliable SQL performance forecasts by using the assumption that a LIO+PIO operation costs about 100 times as much

[13] One way to remember that eliminating LIO calls should be the focus of your SQL optimization job is to write this on your board at work: "High LIO SQL, away!" I once saw this scrawled onto Dan Tow's office whiteboard at Oracle Corporation in the mid 1990s. I found this expression to be desperately corny but sufficiently unforgettable as to serve me faithfully, even through my recent years of accelerated aging and stress-related memory degradation. The phrase is a pun upon the tag line of the legendary American wild-west hero, the Lone Ranger, who would say to his horse named Silver when it was time to leave, "Hi Yo Silver, away!"

[14] If you look at Profiler output instead, then pay attention to the *LIO Blks* column.

[15] All ratios are prone to fallacies, and this one is no exception. For queries that return aggregated data (`sum`, `count`, `min`, `max`, etc.), the acceptable limit of this ratio is of course much higher.

as a bare LIO. However, many people believe that the relative cost is 10,000-to-1. Fortunately, the average latencies of LIO+PIO and bare LIO for an Oracle session are simple to compute. In this paper, I have explained how to use easily obtainable operational data to determine for certain which ratio is closer to the truth. In 71 trace files uploaded to *www.hotsos.com*, the relative performance difference is only a factor of 37. The average LIO latency in our test data is about 50 microseconds; that is, Oracle LIOs execute at a typical pace of only about 20,000 per second. The average PIO latency in our data is about 2 milliseconds, which is faster than typical disk hardware latencies because of the effects of various I/O subsystem memory buffers.

The “myth of 10,000” is more than just an inconsequential technical detail. The myth is dangerous because it motivates performance analysts to make a dreadful mistake: to assume that you can optimize a system by eliminating PIOs. On the contrary, the *most* important optimization goal for the Oracle performance analyst is to reduce *LIO* counts. Even if you could install 1-millisecond disks and an infinite amount of memory for your database buffer cache, your Oracle system performance would be constrained as it probably is now, by the CPU capacity required to process all of your application’s LIO calls.

In the field, Oracle performance analysts who focus on LIO reduction routinely eliminate 50% or more of the CPU and elapsed time required to perform key business functions. Not only does LIO call reduction yield a beneficial impact upon CPU consumption, but it eliminates potentially large segments of user response time wasted on “latch free” events. Even systems with *no* PIO calls can waste tens of hours of CPU time per day if they execute too many LIO calls. Ironically, systems with extremely high (99%+) database buffer cache hit ratios are especially good candidates for LIO call reduction.

References

ADAMS, S. 2000a. *Calculating the Cache Hit and Miss Rates*. Ixora: *http://www.ixora.com.au/tips/tuning/cache_miss.htm*.

ADAMS, S. 2000b. “*cache buffers chains* latches.” *Ixora News*. Nov 2000. *http://www.ixora.com.au/newsletter/2000_11.htm#hash_latches*.

ENGSIG, B. 2001. *Efficient Use of Bind Variables, cursor_sharing and Related Cursor Parameters*. Oracle White Paper: *http://otn.oracle.com/deploy/performance*.

GORMAN, T. 2001. *The Search for Intelligent Life in the Cost-Based Optimizer.* Evergreen Database Technologies: *http://www.evdbt.com/library.htm.*

GURRY, M.; CORRIGAN, P. 1996. *Oracle Performance Tuning*. 2d ed. Sebastopol CA: O’Reilly & Associates.

HOLT, J. 2000. *Predicting Multi-Block Read Call Sizes*. *http://www.hotsos.com*.

HOLT, J.; MILLSAP, C. 2000. *Scaling Applications to Massive User Counts.* Hotsos: *http://www.hotsos.com.*

HOTKA, D. 2000. *Oracle8i from Scratch*. Indianapolis IN: Que.

JENKINS, R. J. JR. 1997. *Hash Functions for Hash Table Lookup.* Bob Jenkins: *http://burtleburtle.net/bob/hash/evahash.html*.

KNUTH, D. E. 1973. *The Art of Computer Programming. Vol. 3: Sorting and Searching*. Reading MA: Addison-Wesley.

MILLSAP, C. 1996. *Configuring Oracle Server for VLDB*. Method R Corporation: *http://method-r.com*.

MILLSAP, C. 2000. *Is RAID 5 Really a Bargain?* Hotsos: *http://www.hotsos.com.*

MILLSAP, C. 2001. *Why 99%+ Database Buffer Cache Hit Ratio is NOT Ok*. Hotsos: *http://www.hotsos.com*.

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

NØRGAARD, M. 2000. *Introducing Oracle’s Wait Interface.* Hotsos: *http://www.hotsos.com.*

Acknowledgments

Sincerest thanks go to Jeff Holt for the research, development, and direct support that made this paper possible. I also want to thank Steve Adams, Jeff Holt, Bjørn Engsig, Mogens Nørgaard, Anjo Kolk, Jonathan Lewis, Gary Goodman, Charles Peterson, Dan Norris, and my wife for services including provision of information, encouragement, inspiration, and proofreading. Finally, thank-you to all my Hotsos Clinic students, various presentation audiences, and the friends whom I meet in Denmark each year, who have inspired me to *study the data* instead of just executing the much easier task of simply writing what feels right at the time.

About the Author

Cary Millsap is the founder and president of Method R Corporation, a small business that builds and optimizes software all over the world (*http://method-r.com*). Cary designs and writes software and educational material. He is the author of *Optimizing Oracle Performance* (O'Reilly 2003), for which he and Method R colleague Jeff Holt were named *Oracle Magazine*'s Authors of the Year. He has presented at hundreds of conferences and courses worldwide, and he is also published in *Communications of the ACM*.

At the time this paper was originally written, Mr. Millsap was a limited partner of Hotsos Enterprises, Ltd. Prior to that, he 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 public speaking. 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.

Tables

Table 1. Summarized LIO and PIO statistics from 71 Oracle trace files uploaded to *www.hotsos.com*. Each of the files represents some kind of system performance problem on operating systems including Linux, Microsoft Windows, OpenVMS, and several variants of Unix.

File id	LIO count	Total seconds LIO duration	Avg. seconds LIO duration (R_{LIO})	PIO count	Total seconds PIO duration	Avg. seconds PIO duration (R_{PIO})	Buffer cache hit ratio
1	164,964	162.28	0.000 984	71,181	24.82	0.000 349	56.851%
2	5,860,106	344.80	0.000 059	686,832	4,723.98	0.006 878	88.280%
3	779,084	294.04	0.000 377	11,743	91.22	0.007 768	98.493%
4	53,187,700	6,174.17	0.000 116	81,259	1,747.53	0.021 506	99.847%
5	8,707	24.16	0.002 775	60	0.81	0.013 500	99.311%
6	1,603	0.30	0.000 187	26	0.41	0.015 769	98.378%
7	38,104	0.88	0.000 023	740	0.06	0.000 081	98.058%
8	165,124	3.00	0.000 018	2,011	1.92	0.000 955	98.782%
9	2,180,601	44.59	0.000 020	17,032	34.20	0.002 008	99.219%
10	158,682,835	813.55	0.000 005	51,761	57.35	0.001 108	99.967%
11	29,167,084	594.78	0.000 020	200,096	156.14	0.000 780	99.314%
12	2,572,801	430.07	0.000 167	1,413	0.83	0.000 587	99.945%
13	24,807	4.21	0.000 170	1,413	0.83	0.000 587	94.304%
14	4,478	0.33	0.000 074	4,384	0.16	0.000 036	2.099%
15	490,768	58.15	0.000 118	489,998	29.50	0.000 060	0.157%
16	3,409	0.31	0.000 091	3,019	18.81	0.006 231	11.440%
17	78,070	7.85	0.000 101	209	1.55	0.007 416	99.732%
18	6,349,842	204.66	0.000 032	27,457	172.26	0.006 274	99.568%
19	11,079,353	323.34	0.000 029	45,091	279.35	0.006 195	99.593%
20	2,142,167	389.85	0.000 182	782	5.76	0.007 366	99.963%
21	7,004,605	288.44	0.000 041	11,817	66.85	0.005 657	99.831%
22	124,128	9.12	0.000 073	123,335	173.45	0.001 406	0.639%
23	4,475,944	436.59	0.000 098	40	0.30	0.007 500	99.999%
24	1,214	0.40	0.000 329	41	0.08	0.001 951	96.623%
25	9,034,310	2,495.74	0.000 276	86,870	1.53	0.000 018	99.038%
26	23,822	13.71	0.000 576	13,407	2.11	0.000 157	43.720%
27	7,643,132	680.02	0.000 089	36,116	358.88	0.009 937	99.527%
28	14,213,571	1,636.09	0.000 115	73,633	818.00	0.011 109	99.482%
29	3,615,049	263.35	0.000 073	38,070	421.44	0.011 070	98.947%
30	12,813,884	1,606.62	0.000 125	56,367	684.10	0.012 137	99.560%
31	10,329,286	1,182.33	0.000 114	35,748	438.05	0.012 254	99.654%
32	2,126	0.63	0.000 296	52	0.25	0.004 808	97.554%
33	1,015,254	95.48	0.000 094	2,579	4.31	0.001 671	99.746%
34	610,857	20.87	0.000 034	3,626	20.26	0.005 587	99.406%
35	2,174,936	391.82	0.000 180	1,072,619	439.58	0.000 410	50.683%
36	3,350,477	196.00	0.000 058	16,404	65.21	0.003 975	99.510%
37	5,854,019	2,437.18	0.000 416	2,332,271	2,365.57	0.001 014	60.159%
38	2,906,917	45.95	0.000 016	129,870	19.85	0.000 153	95.532%
39	277,190,690	1,885.76	0.000 007	116,643	21.23	0.000 182	99.958%
40	58,639	62.27	0.001 062	58,623	154.76	0.002 640	0.027%
41	315,342	10.88	0.000 035	1,095	7.53	0.006 877	99.653%

File id	LIO count	Total seconds LIO duration	Avg. seconds LIO duration (R_{LIO})	PIO count	Total seconds PIO duration	Avg. seconds PIO duration (R_{PIO})	Buffer cache hit ratio
42	1,267,286	48.11	0.000 038	23	0.31	0.013 478	99.998%
43	28,575	2.25	0.000 079	1,618	28.29	0.017 485	94.338%
44	132,650	15.85	0.000 119	5,666	119.72	0.021 130	95.729%
45	3,069,592	309.69	0.000 101	3,044,861	1,120.60	0.000 368	0.806%
46	45,398,641	8,193.07	0.000 180	8,080,721	3,379.65	0.000 418	82.201%
47	2,339,664	194.91	0.000 083	66,478	246.92	0.003 714	97.159%
48	197,500	42.57	0.000 216	30,582	22.87	0.000 748	84.515%
49	22,440	1.93	0.000 086	646	2.43	0.003 762	97.121%
50	838,077	7.93	0.000 009	973	7.08	0.007 276	99.884%
51	973	0.39	0.000 401	274	1.67	0.006 095	71.840%
52	257,700,995	26,342.51	0.000 102	344,374	1,500.57	0.004 357	99.866%
53	26,356,546	1,922.79	0.000 073	81,397	289.03	0.003 551	99.691%
54	1,562,353	172.46	0.000 110	5,961	31.86	0.005 345	99.618%
55	14,792,366	1,541.25	0.000 104	62,495	432.03	0.006 913	99.578%
56	2,210,815	556.55	0.000 252	368,409	911.65	0.002 475	83.336%
57	1,802,938	333.15	0.000 185	1,235,921	10,249.16	0.008 293	31.450%
58	206,152	0.33	0.000 002	175	6.61	0.037 771	99.915%
59	6,870,242	630.20	0.000 092	488,534	730.28	0.001 495	92.889%
60	49,429,232	1,559.51	0.000 032	5,529	70.14	0.012 686	99.989%
61	400,597	79.09	0.000 197	317,030	4,617.67	0.014 565	20.861%
62	14,149	0.30	0.000 021	106	0.26	0.002 453	99.251%
63	71,836	6.28	0.000 087	791	0.54	0.000 683	98.899%
64	1,867,400	100.69	0.000 054	7,528	453.92	0.060 298	99.597%
65	205,375,476	21,887.47	0.000 107	563,687	1,679.61	0.002 980	99.726%
66	6,437,624	1,726.31	0.000 268	446	7.03	0.015 762	99.993%
67	1,342,856,404	48,946.79	0.000 036	507,286	938.80	0.001 851	99.962%
68	1,879,582	95.92	0.000 051	17,570	101.69	0.005 788	99.065%
69	3,833,679	362.58	0.000 095	60,244	202.79	0.003 366	98.429%
70	35,292,970	2,149.32	0.000 061	128,214	523.44	0.004 083	99.637%
71	1,839,819	625.86	0.000 340	375,404	438.18	0.001 167	79.596%
Total	2,649,836,382	141,490.63		21,708,676	41,525.63		
Average			0.000 053 396			0.001 912 859	99.181%

Revision History

12 November 2001: Original revision produced to accompany live presentation at *Oracle OpenWorld 2001*.

15 November 2001: Reformatting and minor editing.

18 October 2002: Corrected timing unit computations in the Perl code.

19 February 2013: Edited to reflect the 2008 changes in author affiliation and copyright ownership.