



# Unbreakable

*James Morle, Scale Abilities, Ltd.*

## Introduction

This is a white paper about Oracle Real Application Clusters. It is not meant to be a comprehensive guide into using RAC, but instead an introduction to the pertinent points of the technology, how it impacts the application, and to what extent the technology is unbreakable.

In order to do this, we will break the paper into several parts. First of all, we will look at the history of RAC, followed by a quick overview of the technology and its resilience to failure. Finally, we will look at the impact upon the application, and how the technology impacts the life of the DBA.

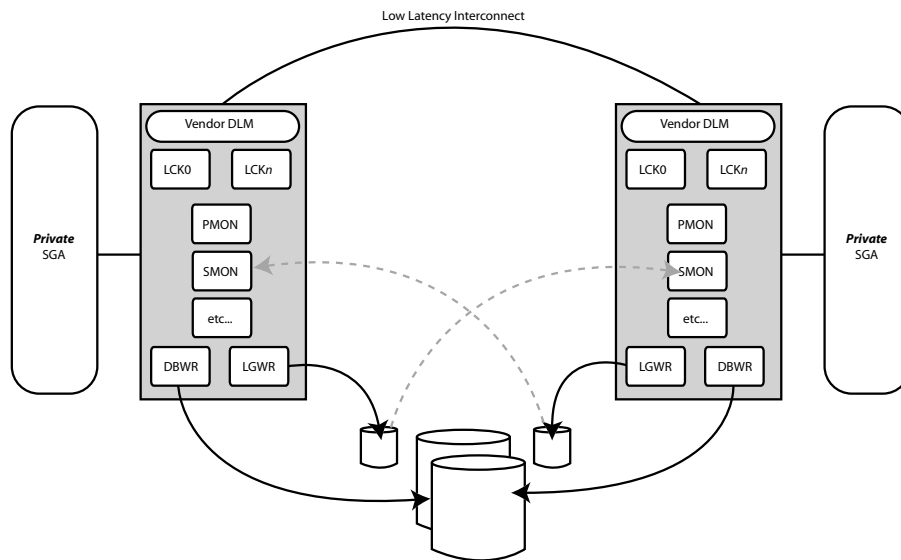
## The Beginning - Oracle Parallel Server

Perhaps the first point to establish is that RAC is not a new technology. From Oracle's perspective this is probably a double-edged sword - longevity normally brings stability with it, but novelty makes better marketing! The truth is, prior to 9i, RAC was known as Oracle Parallel Server (OPS). First conceived as a VMS-only product in the latter days of Oracle 6, it's first real public outing was in Oracle7 release 7.0. The first releases of OPS were less than stable, suffering in many instances (pun 100% intended, sorry) with architectural deficiencies.

One of the issues that caused many problems with OPS in Oracle7 was the separation of responsibility between the hardware vendor (who were tasked with providing the Distributed Lock Manager (DLM)) and Oracle (who *used* the DLM). This separation almost certainly stemmed from the beginnings of OPS on VMS, where DEC provided the DLM. However, this separation caused Oracle two problems:

1. The specification of the DLM and it's associated APIs could not be modified very quickly. Due to the number of interested parties, all with their own development schedules, the DLM was effectively frozen in functionality as soon as the specification was published.
2. If problems occurred in an OPS cluster, it could quickly turn into a finger-pointing exercise between Oracle and the hardware vendors

So, throughout the Oracle7 releases, OPS looked mostly the same, just with a decreasing number of bugs<sup>1</sup>. The architecture looked like this:



Like most OPS or RAC diagrams, this one is difficult to understand without a few words. The grey boxes represent processes running on a server. Each server is physically attached to the same disk as the other, and logically attached to each other via a low latency network of various descriptions. The processes on each machine operate mostly independently, but because they work on the same physical database, they need to ensure they coordinate their actions with the other OPS nodes. For example, all caches on all instances must be in sync with each other, and all transactions must be coordinated between nodes. It is this synchronisation overhead that has made OPS scalability a skilled affair, and we shall see later how RAC has improved upon this compared to OPS.

The best known element of OPS is that of Parallel Cache Management (PCM), the name given to the buffer cache (db\_block\_buffers) synchronisation between nodes. In order to keep the caches in sync, the DLM maintained a record across all instances of which instance currently had permission to write to a portion datafile. When another node needed to write to a block, it communicated with the DLM to make sure any other instances with write permission to that block downgraded their stake on that block. This resulted in a forced write to disk by that instance, so that the current version of the block was on disk. The requesting instance could then read the current version from disk and continue to modify it as it saw fit. This process of writing out to the shared disk is known as a **ping**, and a great deal of OPS tuning in the Oracle7 days was focused on minimising this activity.

The ping process was further complicated by the cost of managing the state of the shared (or global) cache. To maintain a status of every database block would be very expensive, and so Oracle grouped a number of blocks into a single unit of management called a **lock element**. The exact number and grouping of blocks covered by one lock element was governed by the gc\_files\_to\_locks parameter, the most famous tuning parameter of OPS systems! A side effect of this grouping was that, when an instance requested write permission on a block within the lock element, **all** dirty blocks covered by it would be written to disk. If forced writes occur on

---

1. One major change did occur in the Oracle7 time frame - the introduction of releasable locks, enabling block-level cache synchronisation.

blocks not requested by the requesting instance as a result of this grouping, a **false ping** is said to occur.

It took until release 7.3 for the initial problems with OPS to finally be resolved in public releases, including the implementation of releasable locks, which made single block mapping of lock elements possible (though at a cost of increased latency), thereby reducing false pings.

The first major changes to the architecture started after the decision by Oracle to take on the responsibility for the DLM for all platforms starting with release 8.0, though the specific improvements implemented in 8.0 were restricted to providing a uniform view of database-wide instance statistics. The platform vendor retained the responsibility for maintaining the health of the cluster, however.

In Oracle8i, the first major step forward was taken in improving the latency and overhead of the cache coherency between nodes. This improvement was known as the CR Server, or Cache Fusion Step 1, where pings were eliminated for instances that only needed to read blocks that were dirty in a remote instance's cache. This is implemented by constructing a consistent read (CR) cache block of the requested block and directly shipping it over the interconnect to the requesting instance. This was a big step forward, but did not help at all when multiple instances needed to write to the same block, as is often the case in an OLTP application!

## Modern Day - Real Application Clusters

Where the CR Server was referred to as Cache Fusion Step 1, in 9i Oracle implemented the next and final step, that which addressed concurrent writes to the same block by multiple instances. This final step is the piece that prompted Oracle to rename the product to become Real Application Clusters, on the basis of being able to run a real, off-the-shelf application on multiple nodes of the cluster concurrently. In fact, this capability always existed since 7.0, but realistically needed measures in the application to make it scale effectively. Whether RAC is a complete solution to this or not, we shall see in the rest of this paper.

It is worth going into a little more detail on the implementation of write/write cache fusion both because it is interesting, and because it is necessary to understand this in order to make a determination about the scalability of RAC for your application.

The fundamental enabling technology in Oracle9i is that of maintaining a scope for each lock element across all instances. If an instance is the only one to write to a block, a releasable lock element is created of *local scope*. This means that if the instance remains the only one to write to this block, further synchronisation with the DLM is not required - by virtue of the lock being locally scoped, it cannot be written to by other instances. So far, there is no change to the OPS days.

If another instances requests write permission to that same block, the lock element covering it is changed into one of *global scope*. At the same time, the instance holding the dirty copy of the block downgrades its ownership of the lock element, ships the current version of the block to the requesting instance, and retains a Past Image (PI) of the block. The retention of the PI block is for two purposes:

1. To service consistent reads of the block
2. For recovery in the event that the new owner of the current version fails

Consider the following diagram of the actions:

**Table 1: Write/Write Cache Fusion**

Operation	Instance 1 Lock Status	Instance 2 Lock Status
Instance 1: Initiate write to block Z	N/A	N/A
Instance 1: Read block from disk	Local: X	N/A
Instance 2: Initiate write to block Z	Local: X	N/A
Instance 1: Downgrade PCM lock to Null, change scope to Global	Global: N:	N/A
Instance 1: Ship current block to Instance 2, retain PI	Global: N: PI	N/A
Instance 2: Upconvert PCM lock to X	Global: N: PI	Global: X
Instance 1: Initiate checkpoint	Global: N: PI	Global: X
Instance 2: Write block to disk	Global: N: PI	Global: X
Instance 2: Change scope to Local	Global: N: PI	Local: X
Instance 1: Release lock element, discard PI,	N/A	Local: X

The lower part of the table demonstrates the actions is a checkpoint is issued from instance 1. Yes, a forced write does indeed still occur in these circumstances!

## Is it Unbreakable?

Oracle claims that by using RAC, the database becomes many orders of magnitude more reliable, and thus Unbreakable. Is this really the case?

### What is Unbreakable?

Before we can determine whether RAC does indeed make Oracle Unbreakable, let's explore what Unbreakable really means. From Oracle's perspective, Unbreakable means that the database will always be available, by virtue of multiple instances running on different machines. The real definition of Unbreakable, however, means that there is no event that can impact the database as a whole, even if it has multiple instance on different machines.

## **And is RAC Unbreakable?**

At least two single points of failure exist in RAC:

1. The database - there is only one
2. The cluster and DLM - it is a shared entity

Taking the first point, all instances that have the database mounted are reliant upon the content of the database on disk. This means that, whether through hardware error or through software error, a corruption in the database will affect all instances. This applies equally for single instance Oracle as it does for 128 instances of Oracle RAC.

The second point is perhaps more pertinent, however, as corruptions are thankfully rare. All active instances in the RAC are reliant upon the other instances to co-operate in the operation of the each other. If an instance or node fails, the other instances must recover the state of cluster (active node count, remaster locks, etc) and then perform cache recovery on behalf of the failed instance. This can impose a variable period of system unavailability, where the whole cluster is temporarily 'hung' until the operation completes.

Likewise, each node in the cluster is dependent upon the lock manager resources of the others. For example, if instance 2 in the above table was under heavy load, it may be unable to satisfy the requests from instance 1 in a timely manner. This can result on one of the following scenarios:

- In the best case, this will result in instance 1 becoming artificially slow, whilst retaining plenty of idle CPU.
- In the intermediate case, the cluster software may determine that the node supporting instance 2 has gone away, and shoot it down out of the cluster.
- In the very worst case, a communication could go missing between the instances, and a lock element could be left in a state of flux. This is a situation that the author has not observed since the DLM came under the wing of Oracle, but the impact would be that the whole cluster becomes stuck trying to access the lock element that is effectively corrupt. In this case, the database is unavailable from any instance, very broken indeed!

In summary, simply having multiple instances does not, in itself, make the database Unbreakable.

## **No Application Changes? Really?**

The big question, regardless of how interesting the technology is, is: Will my application run adequately on RAC without modification? The answer to this is less than clear, and certainly application specific. The one thing that is certain is that you should bank upon putting considerable thought into configuration, tuning and operation of the cluster.

To explore the very simple facts of running RAC, we need a simple test that behaves like a typical application. For this purpose, a rudimentary Order Entry application was developed, one that performs tasks comparable to an OLTP application, though with substantially less complexity.

The main transaction loop is written in PL/SQL for simplicity, and is listed below for reference:

```
DECLARE
l_cust_id    NUMBER;
num1         NUMBER;
i            NUMBER;
l_level     NUMBER;
l_order_id  NUMBER;
l_stock_id  NUMBER;
l_whs_id    NUMBER;
l_item_id   NUMBER;

TYPE itemListType IS TABLE OF NUMBER(15)
INDEX BY BINARY_INTEGER;
itemList itemListType;

CURSOR c1 (f_cust_id number) IS
SELECT addr_id,forename,lastname
FROM cust
WHERE cust_id= l_cust_id;

CURSOR c2 IS
SELECT cust_id, addr_id,forename,lastname
FROM cust
WHERE lastname like (select substr(name,1,3)||'%' from nameroots
where name_id=trunc(dbms_random.value(1,62000)));

CURSOR c3(in_id number) IS
SELECT item_id, stock_id, whs_id, stock_level
FROM stock
WHERE item_id = in_id
FOR UPDATE OF stock_level;

BEGIN

FOR I IN 1..50 LOOP
if (mod(I,2)=1) then
-- Case 1: Have the customer ID
l_cust_id:=dbms_random.value(1,500000);
OPEN c1(l_cust_id);
FETCH c1 INTO num1,str1,str2;
CLOSE c1;

ELSE
--Case 2: Need to search for customer ID
num1:=0;
WHILE (num1=0) LOOP
OPEN c2;
FETCH c2 INTO l_cust_id, num1,str1,str2;
CLOSE c2;
END LOOP;

END IF;

SELECT order_seq.nextval INTO l_order_id FROM SYS.DUAL;

INSERT INTO orders values ( l_order_id,
trunc(dbms_random.value(1,500000)),
l_cust_id, sysdate, 'N','OPEN');

commit;

-- Try to find some line items that are in stock
itemList(0):=dbms_random.value(1,10000);
itemList(1):=dbms_random.value(1,10000);
itemList(2):=0;
itemList(3):=0;
FOR i IN 2..3 LOOP
WHILE (itemList(i)=0) LOOP
BEGIN
SELECT item_id
INTO itemList(i)
FROM items
WHERE name like (select substr(name,1,3)||'%' from nameroots
where name_id=trunc(dbms_random.value(1,62000))
)
AND rownum=1;
EXCEPTION
WHEN NO_DATA_FOUND THEN
itemList(i):=0;
END;

END LOOP;

-- Check stock levels for selected items and grab them if possible
FOR i IN 0..3 LOOP
OPEN c3(itemList(i));
FETCH c3 INTO l_item_id, l_stock_id, l_whs_id, l_level;
CLOSE c3;
WHILE (l_level<1) LOOP
rollback; -- clear bogus locks
OPEN c3(itemList(i));
FETCH c3 INTO l_item_id, l_stock_id, l_whs_id, l_level;
CLOSE c3;
IF (l_level<1) THEN
SELECT item_id
INTO itemList(i)
FROM items
WHERE name like (select substr(name,1,3)||'%' from nameroots
where name_id=trunc(dbms_random.value(1,62000))
)
AND rownum=1;
END IF;
END LOOP;
UPDATE stock SET stock_level=stock_level-1 WHERE item_id=itemList(i);
INSERT INTO stock_allocations values ( l_stock_id, l_order_id, 1);
INSERT INTO line_items VALUES (line_seq.nextval, l_order_id, l_item_id,1);
commit;
END LOOP;
END LOOP;
END;
```

The transaction chooses a customer, either by a name search or by customer ID, and then places an order for 4 random line items. The stock level is checked, and the items are allocated to that order if stock is available. In addition to this transaction loop, other processes perform stock replenishment (via a nasty range scan on the STOCK table), and others do crazy stuff like switching the forename and lastname of customers to emulate a customer details update.

The implication of this is that several indexes are needed on the tables, and each transaction involves several SELECTs, sequence numbers, 6 INSERTs and one UPDATE. The locking model was designed to avoid deadlocks, but collisions could occur, though not for long as the locks take place in a short section immediately followed by a COMMIT or ROLLBACK.

This application simulator was executed against a small Linux-based cluster, consisting of two dual-processor machines, each with 1GB of memory. The storage consisted of a simple shared SCSI array connected to the two machine simultaneously. One of the machines had dual Pentium 3 Tualatin processors at 1.2GHz, the other a pair of lowly Pentium 3 450MHz processors. A series of tests was performed, with different RAC configurations and varying levels of user load.

### Single Instance, RAC Unlinked

As a baseline, the various levels of user load were first executed against a plain-vanilla Oracle9i database, with the RAC module unlinked. This is an important first test, because having RAC linked in imposes additional codepath overhead, notably the GES/GCS (DLM). As with all the other tests, the procedure was as follows:

1. Bounce the Oracle instance
2. Run a 10 user test twice to prime the buffer cache and library cache
3. Run 10 user test
4. Run 25 user test
5. Run 35 user test
6. Run 50 user test

In this special case, a 100 user test was also executed so that the aggregate throughput could be compared to the forthcoming 2-node 50 user test. Statspack was used as the primary measurement tool, with a snapshot taken at the start and end of each test, and the same sections were observed on each run to compare the differences:

- Top 5 Wait Events section
- Statistics: CPU used by this session, execute count, redo size, consistent gets, physical reads, physical writes, user commits - all reported as average rates per second across the period.

The results follow:

**Table 2: Single Instance NORAC Wait Events**

		Top 5 Wait Events		
10user	Event	Waits	Wait Time (s)	% Total Wt Time
	buffer busy waits	357	15	34.69
	latch free	251	6	15.24
	enqueue	69	6	15.13
	db file sequential read	232	6	14.86
	log file parallel write	912	6	13.87

**Table 2: Single Instance NORAC Wait Events**

Top 5 Wait Events				
25user	~~~~~ Event	Waits	Wait Time (s)	% Total Wt Time
	-----	-----	-----	-----
	db file sequential read	2,160	87,863	42.45
	log file parallel write	2,246	49,104	23.73
	buffer busy waits	2,680	48,946	23.65
	enqueue	465	10,661	5.15
latch free	884	6,850	3.31	
-----	-----	-----	-----	-----
35user	~~~~~ Event	Waits	Wait Time (s)	% Total Wt Time
	-----	-----	-----	-----
	buffer busy waits	2,772	341	67.58
	latch free	1,849	68	13.39
	enqueue	625	56	11.13
	log file parallel write	2,684	19	3.80
db file sequential read	279	8	1.57	
-----	-----	-----	-----	-----
50user	~~~~~ Event	Waits	Wait Time (s)	% Total Wt Time
	-----	-----	-----	-----
	buffer busy waits	5,599	677	62.73
	enqueue	1,605	198	18.32
	latch free	3,109	126	11.65
	log file parallel write	3,446	26	2.45
db file sequential read	934	26	2.43	
-----	-----	-----	-----	-----
100user	~~~~~ Event	Waits	Wait Time (s)	% Total Wt Time
	-----	-----	-----	-----
	buffer busy waits	16,184	1,499	44.02
	enqueue	5,276	730	21.43
	latch free	10,412	450	13.23
	db file sequential read	8,463	256	7.52
buffer busy due to global cache	4,382	101	2.97	
-----	-----	-----	-----	-----

**Table 3: Single Instance NORAC Statistics**

	CPU Used By This Session	Execute Count	redo size	consistent gets	physical reads	physical writes	user commits
10user	112.9	1,358	767,148	6,416	36.4	57.9	253.6
25user	88.1	977.6	542,061	7,496	67.5	61.1	182.3
35user	160.6	1,810	1,810	14,072	14.7	92.8	339.3
50user	154.2	1,731.6	974,170	11,601	26.5	80.0	323.4
100user	134.3	1,028.5	581,504	23,424	92.1	70.1	185.2

The first thing to observe is that the statspack results for the 25 user test were clearly incorrect. This distorts both the wait events and the statistics, and so these results must be discarded. Unfortunately, this was observed after the test period was over, and the true cause could therefore not be determined. Of the remaining testing, the 35 user test was the most productive, striking

the right balance between contention on enqueues and raw throughput. It successfully used 1.6 CPUs executing the work, and executed 339 tps. All the tests suffered quite badly from buffer busy waits, an unfortunate side effect of working on a limited data set. However, it does provide an interesting problem for RAC to deal with once those data references are occurring on two nodes.

**Single Instance RAC**

The purpose of this test was to see what, if any, overheads there were simply by having RAC linked in and the database started up with `cluster_database=true`. Results follow:

**Table 4: Single Instance RAC Wait Events**

Top 5 Wait Events				
10user	~~~~~ Event	Waits	Wait Time (s)	% Total Wt Time
	-----	-----	-----	-----
	db file sequential read	2,490	38	29.42
	buffer busy waits	2,629	37	28.99
	ges remote message	95	34	26.29
	log file parallel write	1,264	7	5.65
latch free	357	7	5.42	
	-----	-----	-----	-----
25user	~~~~~ Event	Waits	Wait Time (s)	% Total Wt Time
	-----	-----	-----	-----
	buffer busy waits	2,023	164	52.93
	latch free	1,794	61	19.75
	enqueue	531	26	8.49
	ges remote message	209	23	7.41
log file parallel write	2,321	15	4.94	
	-----	-----	-----	-----
35user	~~~~~ Event	Waits	Wait Time (s)	% Total Wt Time
	-----	-----	-----	-----
	buffer busy waits	2,872	384	62.29
	enqueue	943	80	12.92
	latch free	2,098	74	11.97
	ges remote message	322	32	5.16
log file parallel write	2,808	21	3.45	
	-----	-----	-----	-----
50user	~~~~~ Event	Waits	Wait Time (s)	% Total Wt Time
	-----	-----	-----	-----
	buffer busy waits	6,336	827	62.10
	enqueue	1,853	210	15.78
	latch free	4,506	188	14.08
	ges remote message	611	45	3.35
log file parallel write	3,731	30	2.24	
	-----	-----	-----	-----

**Table 5: Single Instance RAC Statistics**

	CPU Used By This Session	Execute Count	redo size	consistent gets	physical reads	physical writes	user commits
10user	46.6	431.9	230,950	3,415	74.6	41	80
25user	142.9	1,466	827,782	8,288	21.1	97	275.1

**Table 5: Single Instance RAC Statistics**

	CPU Used By This Session	Execute Count	redo size	consistent gets	physical reads	physical writes	user commits
35user	135.8	1,472	853,473	8,585	12.3	111.3	276.7
50user	146.2	1,493	841,472	9,947	7.1	95	280.5

In this case, the 50 user test was the most successful, though not reaching the 339tps of the NO-RAC test. In fact, this test was 18% down in throughput compared to the NORAC tests. The reason for this is that, although there are no other nodes currently in the cluster, Oracle still needs to go through the motions as if there were. This includes additional codepath, and even the allocation of global cache locks on the buffers used. The statspack report showed that the average global lock get time took 52ms, and that there were 12 of these every second during the 50 user test. This is likely to be the culprit for the degraded throughput.

**Two node RAC**

Now things get interesting. Let's see how RAC behaves with this workload:

**Table 6: Two Instance RAC Wait Events**

	Top 5 Wait Events				
10user	~~~~~				
	Event	Waits	Wait Time (s)	% Total Wt Time	
	-----				
	buffer busy due to global cache	4,057	122	26.65	
	global cache null to x	3,069	89	19.48	
	buffer busy waits	1,250	77	16.84	
	ges remote message	11,456	47	10.32	
enqueue	666	36	7.93		
	-----				
25user	~~~~~				
	Event	Waits	Wait Time (s)	% Total Wt Time	
	-----				
	global cache busy	1,004	838	54.75	
	buffer busy due to global cache	3,024	180	11.75	
	buffer busy waits	1,460	125	8.16	
	global cache null to x	2,198	112	7.30	
enqueue	1,030	102	6.67		
	-----				
35user	~~~~~				
	Event	Waits	Wait Time (s)	% Total Wt Time	
	-----				
	enqueue	7,351	1,992	36.82	
	buffer busy due to global cache	21,072	1,134	20.96	
	buffer busy waits	8,072	1,003	18.53	
	global cache null to x	10,503	480	8.87	
global cache cr request	14,846	183	3.38		
	-----				

**Table 6: Two Instance RAC Wait Events**

Top 5 Wait Events	
50user	<pre> ~~~~~ Event                               Waits   Wait      % Total                                       Time (s)  Wt Time ----- enqueue                             105,047   50,899   91.63 buffer busy waits                     6,245     1,320    2.38 buffer busy due to global cache       15,172     1,231    2.22 ges remote message                    37,963     1,101    1.98 global cache null to x                 6,969       446     .80 ----- </pre>

**Table 7: Two Instance RAC Statistics**

	CPU Used By This Session	Execute Count	redo size	consistent gets	physical reads	physical writes	user commits
10user	47.6	294	164,768	1,862	4.5	33.5	54.6
25user	46.9	421	225,961	2,462	3.8	50.5	78.4
35user	47.5	268.9	151,393	2,181	1.4	30.9	50.1
50user	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Oh dear, oh dear, what has happened here?! The best throughput was from the 25 user test, which is essentially the same as the 50 user single instance test due to the presence of 25 users on each node. Let's concentrate on the 25 user test, and supplement the statistics from the second node.

**Table 8: Two Instance RAC Statistics (2nd Instance)**

	CPU Used By This Session	Execute Count	redo size	consistent gets	physical reads	physical writes	user commits
25user	84.6	299	171,832	2,239	16.4	27.6	55.8

Even with the addition of these results, the total transaction rate is a mere 132tps, almost 66% down on the best single instance test. One thing that is immediately apparent is the additional time spent waiting for enqueues. In fact, the 50 user test had to be aborted when it well and truly wrapped itself around the enqueue axle. In the case of this testing, it was the enqueue contention (specifically, TX enqueues - row level locks) that caused the biggest problem to scalability, and so further testing was performed to assess the changes in enqueue latency when using RAC compared to single instance NORAC. Before going on to that topic, we should briefly touch upon the global cache statistics.

Absolutely *no tuning* was performed on the instances, to see what the outcome was. Plug-and-play, as it were. In this case, the overhead was quite significant, with quite considerable waits for global cache events. In fact, if the enqueue problem did not appear, global cache wait would have dominated the response time of each session. It should be pointed out, however, that the single instance tests spent most of their time waiting for buffers, and that a buffer serviced from a remote instance will always incur a latency penalty when compared to a simple memory ac-

cess on the local instance. However, it does highlight that the scalability of RAC is not free, and that partitioning of data access is still a significant issue.

## Enqueue Rates

After the enqueue onslaught encountered in the high load dual instance RAC tests, it seemed appropriate to investigate simple enqueue rates between two instances. To simulate this, a very basic SQL script was used:

```
create table enq ( a number);
insert into enq values (1);
execute statspack.snap;
set timi on
declare
dummy number;
begin
for i in 1..10000 loop
    select * into dummy from enq where rownum=1 for update ;
    rollback;
end loop;
end;
/
execute statspack.snap;
```

This script was executed against both nodes (with the table create excluded from the second node), and the results compared to the same script executing against a single instance (with RAC linked in and started with `cluster_database=true`). The statistics from the faster of the two machines will be compared for both tests. For the single instance, the following rate was achieved:

```
enqueue requests          332.2
```

Compare this to the result with two nodes competing for the same TX lock:

```
enqueue requests          121.1
```

When the script is run twice concurrently against a single instance, even under a VMware Linux installation through Windows on a uniprocessor laptop<sup>1</sup>, the following results were obtained:

```
enqueue requests          181.3
```

A full 50% faster than two node contention resolution.

## Stability

Before we summarise, a little word on stability. Though the test systems were not those with a proven heritage of stability, and almost certainly not supportable by Oracle, the stability of the test clusters was hair-tearing! During a short period of final testing, the following failures were experienced:

- Four total cluster failures. These were not system failures, but pure failures of the cluster software infrastructure. Two of these seemed to coincide with periods of high load, under which it is possible that one node may be deemed 'down' by the cluster. However, BOTH instances were terminated by the cluster software. The other two, whether by coincidence or

---

1. Unfortunately, this was the last test, and the prospect of yet another reboot was too much to bear.

otherwise, were experienced when trying to disable cache fusion by setting static hash PCM locks against the most contended for datafiles...

- Two system crashes. Almost certainly a deficiency in the Linux SCSI driver, where it was not really designed for this multiple host adapter situation.
- Two corrupted redo logs. Probably for the same reason as the system failures, these corruptions resulted in the inability by Oracle to perform cache recovery on startup, thus necessitating a database rebuild.
- One corrupted system tablespace. The segment header of the system undo segment became fractured and unrecoverable. Almost certainly an artifact of using VMware nodes in this particular case!

## Summary of Testing

Hopefully, the test results will show that RAC is not a simple option, and that an application still needs some analysis of its workload profile before it can be deployed optimally on a RAC configuration. RAC is certainly a huge improvement over OPS, but it is important to understand that the *reduced* synchronisation latency provided by RAC is not the same a *zero* synchronisation latency, and it still needs to be carefully considered. However, there are some very valid reasons for going with RAC, even if it does require more than a little effort. The best two reasons for going with RAC for scalability are:

- When the biggest machine is not big enough
- Commodity hardware

The first of these is pretty clear; if the largest machine cannot offer enough compute power for your application, RAC is the only way to go. Of course, if this is the case, it is likely that the budgetary concerns surrounding the staffing levels for RAC are unlikely to be an issue, simply due to the implied size of the implementation. However, it is paramount that a risk analysis is performed to assess whether or not RAC will provide the scalability required by the application within the limits of change possible with the application. If the application cannot be made as scalable as necessary in a RAC environment, it is likely that it will provide negative scalability across the nodes, thus negating the purpose of using it!

The commodity hardware reason is completely different. Commodity hardware is essentially anything based upon Intel Pentium processors, because the Intel-based servers provide tremendous power for a very low cost. However, this cost advantage quickly disappears as soon as the components used to build the server are not mass-produced. For example, Sequent worked for years producing very powerful servers based upon Intel components. Unfortunately, their servers were very expensive - comparable to proprietary RISC servers, or more - because of the R&D cost of building very scalable servers, regardless of the underlying processors.

If the mass-produced chipsets and motherboards are used, the cost is substantially less than any RISC server. However, the scalability inevitable suffers. It is uncommon to find an Intel-based server with greater than four processors while still retaining the commodity pricing, and this is where RAC provides a compelling solution. Using RAC, a significant amount of compute power can be provided in the database server layer with a modest cost outlay.

For example, a 4-way Intel Pentium 4 based server costs around \$5,000. Putting 16 of these in a rack makes a 64-way database services tier for only \$80,000. Compared to around \$1M for a RISC-based solution, probably with less compute power, this is a major cost advantage.

Once again, though, you need to be confident that your application will scale across all the nodes.

## Implications in Daily Operations

Perhaps the most overlooked aspect of running OPS and RAC is the burden it pushes onto the support staff. Whereas running single instance Oracle is practically a lights-out operation, RAC requires significantly more knowledge from the supporting DBA staff. Apart from all the technical knowledge required by the DBAs, which is probably an order of magnitude greater than single instance, aspects such as recovery are necessarily more complex. Each instance, for example, writes to its own set of redo logs. These in turn are archived to a local filesystem and need to be backed up as a coherent unit to assure recoverability.

Clustered filesystems provide a nice solution to the backup and recovery problem, where the same archive destination can be made available on all nodes running instances. In addition, clustered filesystems provide an alternative to using raw disk, previously the only method by which Oracle datafiles could be shared between nodes. However, the filesystem used should be able to provide raw-like access to the datafile, such as Veritas Quick-I/O.

## Papa's Secret Laptop RAC Recipe

Mention has been made to a laptop cluster throughout this paper. It's so interesting and useful as a way to 'play' with RAC, that it merits a few words by way of an epilogue. Though there are many ways to construct a RAC cluster on a laptop, the easiest way is to use VMware Workstation on top of Linux. Here's the recipe:

- One Laptop, as fast as you can get, stuffed to the gills with memory
- One large hard disk - you quickly chew through disk space
- A recent Linux distro (I used RedHat 7.2). A 2.4 kernel is essential
- Sistina's LVM tools ([www.sistina.com](http://www.sistina.com))
- VMWare Workstation, version 2+

Method: Take a freshly installed Linux distribution, and configure the kernel for use with RAC. Essential options are: Character Devices->Watchdog Cards-> Software Watchdog (module), Multi-Device Support->Logical Volume Manager(module or static). Before doing the kernel build, edit `/usr/src/linux/drivers/char/softdog.c`, and add the line:

```
#define ONLY_TESTING
```

just below the `#define` for `TIMER_MARGIN`. Alternatively, when you load the module, specify the `soft_noboot` option. This is important, otherwise the machine will reboot during one of your many future cluster failures. Make sure during your install that you leave plenty of unpartitioned disk space for use later.

Next, take a dash of VMware Workstation, and install it onto Linux. Configure the virtual machine for half of your physical memory, and Host-Only networking. After simmering for a few minutes, boot the Linux CD from within the VMware virtual machine, and install Linux once gain. This can be a virtual disk, or a 'Raw Disk' install.

Once the VMware Linux has absorbed your attention for a while, configure the kernel as above. Next, switch back to the host Linux system, and create NFS shares for:

- `$ORACLE_HOME`
- A separate copy of `$ORACLE_HOME/oracm`

Back on the guest Linux install, mount the NFS `$ORACLE_HOME`, and mount the copy of `$ORACLE_HOME/oracm` **on top** of the `oracm` directory on the newly mounted `$ORACLE_HOME`.

You are now ready to prepare the database. Taking a sharp knife, slice your free space into a neat LVM-type partition. This is a partition of type 8E (this will make sense when you get into `fdisk!`).

Next, install the LVM tools, and issue the following command:

- pvcreate <your new LVM partition>
- vgcreate -s 4M ractest <your LVM partition>

For each of the following volumes, use lvcreate to make a corresponding logical volume:

- SYSTEM
- UNDO1
- UNDO2
- TEMP
- DATA
- QUORUM
- CTRL1
- CTRL2
- REDO\_THREAD1\_1
- REDO\_THREAD1\_2
- REDO\_THREAD2\_1
- REDO\_THREAD2\_2

Pucker mate<sup>1</sup>. Now, for some strange reason, Linux does not have raw disk devices by default, only block devices. So we need to make raw devices using the ‘raw’ command. For now, just do the QUORUM volume:

- raw /dev/raw/raw1 /dev/ractest/quorum

Add an entry in the /etc/hosts file for each node, on each node. Just call them node1 and node2 for simplicity. The address given should correspond with the Host-only networking address for each Linux installation. Now, edit the \$ORACLE\_HOME/oracm/nmcfg.ora file on both Linux installs (remember that the second node has a separate mount for this), and configure as follows:

```
DefinedNodes node1 node2
CmHostName node1
CmDiskFile /dev/raw/raw1
```

Obviously change the CmHostName as appropriate.

Now try and bring up the cluster agents:

```
insmod softdog soft_margin=60 soft_reboot=1
$ORACLE_HOME/oracm/bin/ocmstart.sh
```

Check the logfiles (\$ORACLE\_HOME/oracm/log) carefully. When both ‘machines’ are running the cluster agents, the cm.log should declare that (2) nodes are in the cluster. If not, you need to fix something.

It’s easiest to build the database with RAC linked out, so make sure it is:

```
cd $ORACLE_HOME/rdbms/lib
make -f ins_rdbms.mk rac_off
make -f ins_rdbms.mk ioracle
```

Now run the ‘raw’ command for all the other volumes, and put a corresponding entry in /etc/sysconfig/rawdevices (RedHat only, SuSe is different) for future sanity reasons.

Taking a generous helping of previous experience, create a database using only the THREAD1 and UNDO1 datafiles (plus the other ones you need, but NOT UNDO2, THREAD2 logfiles...).

Leave to simmer for a good few minutes, until a fragrant database is evident.

Now relink Oracle (rac\_on), and start the database with cluster\_database=true set in the init.ora.

If this works, you are doing well!

---

1. Copyright Jamie Oliver

Add the THREAD2 log files, using the 'THREAD 2' option of the add logfile syntax. Then create the UNDO2 tablespace, and add the appropriate references to it in the init.ora for the second instance. All that remains is to

```
alter database enable public thread 2;
```

and to start the instance on the second (guest) Linux node.

Enjoy!

Serves **many**.

**About the Author.** *James Morle is the founder of Scale Abilities Ltd., a specialist consulting company offering both high-end consulting, and unique software products. With over 10 years experience in architecting and building some of the world's largest Oracle systems, James is a well respected member of the Oracle community. Scale Abilities was founded in 2000 to concentrate on providing the highest quality consulting services, in addition to developing revolutionary new consulting-based software products.*

**<http://www.scale-abilities.co.uk>**