Commercial database I/O configurations

Colin Eldridge and Ed Menze

Acronyms

Acronym	*Description*
BIO	File system buffered i/o
DIO	Direct i/o
DDIO	Discovered direct i/o
QIO	Quick i/o
CQIO	Cached QIO
CIO	Concurrent i/o
ODM	Oracle disk manager
CODM	Cached ODM

Session objectives

- To explain the Database I/O configuration choices using VxFS/CFS
- To explain when file data is buffered in the file system cache
- To explain why it is sometimes better to not buffer the file data
- To explain the page size, the sector size and file system block size
- To explain how EXCLUSIVE write file locking can be avoided
- To also demonstrate the usefulness of the vxtrace diagnostic tool

- Overriding session objective:

  To assist planning and decisions concerning the configuration choices for Commercial Database environments utilising VxFS and CFS.

  VxFS (and CFS) are not solely about improving storage utilisation, they're also about improving system memory utilisation.

What is the file system cache?

- All operating systems provide a mechanism of buffering fs file data in memory, often called the buffer cache, we call it:

  the pagecache

- File data is held in units of pages aligned along the file offset:

  struct page

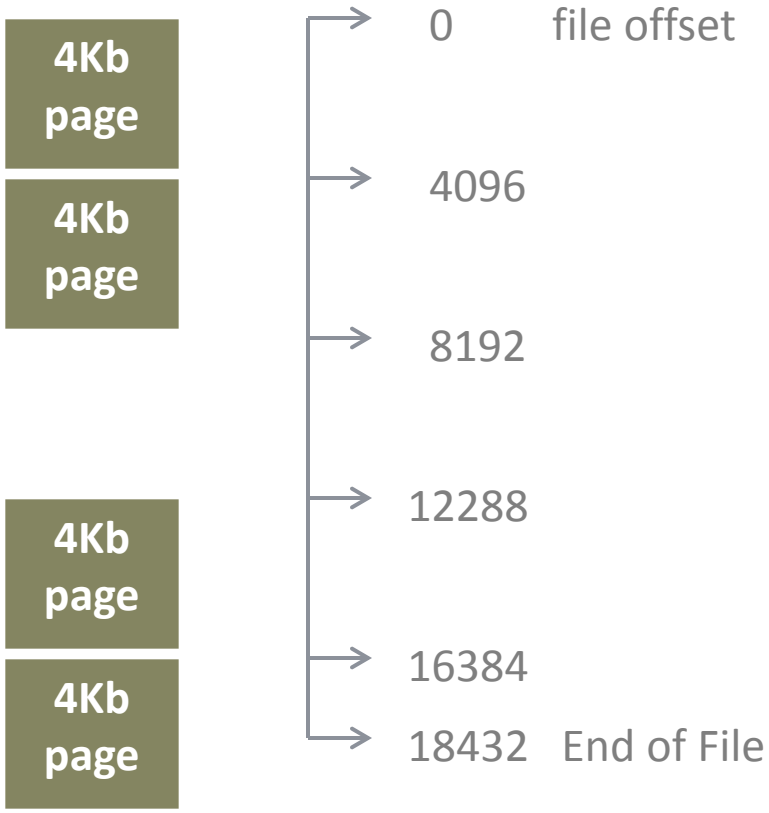
- The page size can differ between operating systems:

  the page size is commonly either 4Kb or 8Kb

- The pagecache is independent of the file system type, part of:

  the operating system's Virtual Memory Management [VMM]

file data held in pages in the pagecache



The diagram depicts a file which has:
- a size 18432 bytes (18Kb)
- 4 associated pages (total 16Kb)
- 14Kb of its file data held in memory
- 4Kb of file data not held in memory

How is file data buffered in the pagecache?

- By simply performing buffered i/o to a file, reads and/or writes
- A file's data will remain in the pagecache after the i/o is complete, this is "file system buffered i/o" [BIO]
- asynchronous write BIO marks each updated page as 'dirty' (modified), pages are later flushed and marked as 'clean'
- synchronous write BIO also marks each updated page as 'dirty'
- BIO "to and from disk" is performed in one or more multiples of the page size, except when the page spans EOF or a sparse area.
- BIO takeaway: BIO can perform read-ahead (pre-fetching data from disk) by populating more pages than requested whilst performing a sequential read i/o pattern.

What is the direct I/O caching advisory?

- VxFS direct i/o [DIO] does not buffer data in the pagecache
- DIO requires specific alignment and sizing criteria to be met.
- If the criteria is not met, the i/o will be performed as [data-synchronous] BIO, for this reason DIO is an 'advisory' only.
- DIO's smallest i/o size is the sector size. The sector size on AIX, Solaris and Linux is 512 bytes, on HP-UX it is 1024 bytes.
- DIO requires the i/o to start on a sector-aligned (file-offset) boundary and the i/o must be a multiple of sectors in size.
- DIO is performed as synchronous i/o, meaning that the data is already on disk when the write syscall returns to the user.
- DIO cannot perform read_ahead (as DIO does not buffer data)

How is the VxFS DIO advisory enabled?

- VxFS mount options
  - convosync=direct (convosync affects files open()ed using O_SYNC/O_DSYNC)
  - mincache=direct (mincache affects all other files in the file system)
- The VX_SETCACHE ioctl (vxfsio) interface – passing VX_DIRECT
- Opening the file using flag O_DIRECT (non posix, where available)
- Triggering via discovered_directo_sz [DDIO] tunable, default 256Kb

```
# /opt/VRTS/bin/vxtunefs /vxfs.mounted.here
Filesystem i/o parameters for /vxfs.mounted.here
...
discovered_direct_iosz = 262144
```

- Memory mapped I/O cannot utilise DIO. If the file is memory mapped, any DIO accesses are done as data synchronous I/O.
- DIO is performed synchronously without buffering in the pagecache
  - ODM has its own form of direct i/o which can be asynchronous if requested

DIO and DDIO takeaways

- Database utilisation of DIO is achieved via “convosync=direct”
  - Use of “mincache=direct” is not required
- DIO avoids double buffering of file data, but not the EXCL rwlock
- Mixing DIO/DDIO and BIO to the same file can hit performance
- Be aware of DDIO when performing I/O with larger I/O sizes
- DIO will be performed as data-synchronous buffered I/O if:
  - The file is memory mapped
  - The alignment and sizing criteria are not met
- DIO and DDIO are not identical, because DDIO does not:
  - require a synchronous commit of the inode when the file is extended or blocks are allocated.
  - take an alternative action if the alignment and sizing criteria are not met

VxFS Fundamentals – mkfs block sizes

```
# mkfs -V vxfs -o bsize=1024 /dev/vx/rdsk/diskgroup/volume
209715200 sectors, 104857600 blocks of size 1024, log size 65536 blocks

# mkfs -V vxfs -o bsize=2048 /dev/vx/rdsk/diskgroup/volume
209715200 sectors, 52428800 blocks of size 2048, log size 32768 blocks

# mkfs -V vxfs -o bsize=4096 /dev/vx/rdsk/diskgroup/volume
209715200 sectors, 26214400 blocks of size 4096, log size 16384 blocks

# mkfs -V vxfs -o bsize=8192 /dev/vx/rdsk/diskgroup/volume
209715200 sectors, 13107200 blocks of size 8192, log size 8192 blocks
```

The above mkfs commands are all using the same 100GB volume, the log size is 64MB

VERITAS file system block numbers and extents

VxFS splits the device into a linear list of equal sizes blocks, starting at block number 0

/dev/vx/rdsk/diskgroup/volume

fsblock 0	fsblock 1	fsblock 2	fsblock 3	fsblock 4	fsblock 5	fsblock 6	...	fsblock 'n'

An "extent" is simply a contiguous number of file system blocks. An extent is either currently allocated to a file [inode] or it is currently free (i.e. not allocated to a file).

/dev/vx/rdsk/diskgroup/volume

fsblock 0	...	fs block 30724	fsblock 30724	fsblock 30725	fsblock 30726	fsblock 30727	...	fsblock 'n'

Above in **blue** is an example of a single extent, 3 file system blocks in size, starting at file system block number **30724** , so this extent consists of blocks **30724, 30725, 30726**.

Using dd and vxtrace

```
# vxtrace -g testdg &
[1]     647324
# dd if=/dev/vx/rdsk/testdg/vol2 bs=1024 skip=2464 count=1 1>/dev/null 2>&1 &
[2]     651480
10263 START read vdev vol2 block 4928 len 2 concurrency 1 pid 651480
10263 END read vdev vol2 op 10263 block 4928 len 2 time 0
# kill -9 647324
```

Using vxtrace to watch raw disk i/o and direct i/o

```
# mount -V vxfs -o mincache=direct /dev/vx/dsk/testdg/vol2 /mnt
# cd /mnt; ls -li bfile
5 -rw-r--r--   1 root     system       377910 Oct 31 13:37 bfile

# echo '5i.mapall' | /opt/VRTS/bin/fsdb /dev/vx/rdsk/testdg/vol2
offset    device          block           length
0              0           2464              368
376832         0           1816                2

# vxtrace -g testdg | grep START &
# dd if=/dev/vx/rdsk/testdg/vol2 bs=1024 skip=2464 count=1 1>/dev/null 2>&1
10264 START read vdev vol2 block 4928 len 2 concurrency 1 pid 651480

# dd if=/mnt/bfile bs=1024 count=1 1>/dev/null 2>&1
10265 START read vdev vol2 block 4928 len 2 concurrency 1 pid 651296
```

Using vxtrace to watch buffered i/o reads

```
# mount -V vxfs /dev/vx/dsk/testdg/vol2 /mnt
# cd /mnt
# vxtrace -g testdg | grep START &
# dd if=bfile of=/dev/null bs=1024 count=1
10960 START read vdev vol2 block 4928 len 8 concurrency 1 pid 1335442
1+0 records in.
1+0 records out.
#
# dd if=bfile of=/dev/null bs=1024 count=1
1+0 records in.
1+0 records out.
# dd if=bfile of=/dev/null bs=1024 count=1
1+0 records in.
1+0 records out.
```

Performing read_ahead

```
# dd if=bfile of=/dev/null bs=1024 count=4
4+0 records in.
4+0 records out.
10961 START read vdev vol2 block 4936 len 128 concurrency 1 pid 676004
10962 START read vdev vol2 block 5064 len 128 concurrency 2 pid 676004
10963 START read vdev vol2 block 5192 len 128 concurrency 3 pid 676004
10964 START read vdev vol2 block 5320 len 128 concurrency 4 pid 676004
10965 START read vdev vol2 block 5448 len 120 concurrency 5 pid 676004
```

Pre-allocating and identifying extents

```
# touch newfile
# setext -r 20000 -f contig ./newfile
# du -k ./newfile
20000   ./newfile
# ls -li ./newfile
4 -rw-r--r--   1 root     root            0 Feb 22 10:49 ./newfile
# dd if=/dev/zero of=./newfile bs=100k count=200
200+0 records in
200+0 records out
# ls -li ./newfile
4 -rw-r--r--   1 root     root     20480000 Feb 22 10:51 ./newfile
# /opt/VRTS/bin/fsmap -a ./newfile
  Volume  Extent Type     File Offset     Extent Size     File
    vol2         Data               0        20480000     ./newfile
# /opt/VRTS/bin/ncheck -oblock=- /dev/vx/rdsk/testdg/vol2 | grep newfile
UNNAMED     999      4    -      - 0/32768-0/52767 /newfile
```

Small fragmented file – buffered read i/o

```
# echo '8i.mapall' | /opt/VRTS/bin/fsdb /dev/vx/rdsk/testdg/smallvol
offset  device            block      length
0            0            38584           1
1024         0            38640           2
3072         0            38672           1
# ls -li 4k.fragmented.file
8 -rw-r--r--   1 root     system      4096 Dec 15 08:49 4k.fragmented.file
#
# dd if=./4k.fragmented.file of=/dev/null bs=1024 skip=1 count=1
6 START read vdev smallvol block 77168 len 2 concurrency 1 pid 831646
7 START read vdev smallvol block 77280 len 4 concurrency 1 pid 831646
8 START read vdev smallvol block 77344 len 2 concurrency 1 pid 831646
1+0 records in
1+0 records out
```

Small fragmented file – buffered write i/o

```
Note: File system has since been umounted and freshly mounted again
# ls -li 4k.fragmented.file
8 -rw-r--r--   1 root     system      4096 Dec 15 08:49 4k.fragmented.file
# vxtrace -g testdg | grep START &
# dd if=/dev/zero of=./4k.fragmented.file bs=1024 seek=1 count=1 conv=notrunc
12 START read vdev smallvol block 77168 len 2 concurrency 1 pid 544850
13 START read vdev smallvol block 77280 len 4 concurrency 1 pid 544850
14 START read vdev smallvol block 77344 len 2 concurrency 1 pid 544850
1+0 records in
1+0 records out
#
15 START write vdev smallvol block 77168 len 2 concurrency 1 pid 98432
16 START write vdev smallvol block 77280 len 4 concurrency 1 pid 98432
17 START write vdev smallvol block 77344 len 2 concurrency 1 pid 98432
```

Quick I/O

- Not an API, databases access each regular file as a raw device
- No exclusive write locking is performed on the file
- I/O is performed direct, thus prevents double-buffering of data
- Enabled/disabled via the VxFS mount options qio and noqio

```
# qiomkfile -s 100m /database/dbfile
# ls -al
-rw-r--r-- 1 oracle dba 104857600 Oct 22 15:03 .dbfile
lrwxrwxrwx 1 oracle dba        19 Oct 22 15:03 dbfile -> .dbfile::cdev:vxfs:
```

- The kernel driver utilised by QIO is called “fdd”
- QIO is supported on CFS and enabled by default.
- QIO is not supported on Linux (CIO is the alternative option)

Concurrent I/O caching advisory

- Not an API, allows concurrent readers and writers
- No exclusive write locking is performed on the file
- I/O is performed direct, thus prevents double-buffering of data
- Enabled for all files in a file system via the cio mount option
- The cio option cannot be disabled by remounting the file system
- Enabled per file descriptor:
  - Via VX_SETCACHE ioctl passing VX_CONCURRENT
  - On AIX only, via the O_CIO open flag for the open() system call
  - Write syscalls to the same file through other fd's remain unaffected
- Advisory only, same alignment and sizing criteria as DIO
- CIO is supported on CFS

Intermediate Session Summary

- DIO/QIO/CIO/ODM all avoid the double buffering of data
- BIO permits use of VxFS read_ahead
- QIO/CIO/ODM all avoid the exclusive file write lock
- Commercial databases submit synchronous I/O to file systems
- DB2 customers commonly use CIO
- Sybase customers commonly use QIO or CIO
- Oracle customers commonly use ODM
- Database customers who commonly use BIO are often unaware of the choice of I/O options available.

This session will now explain ODM/CODM/CQIO

ODM – Oracle Disk Manager

- ODM i/o is very efficient
  - Interface and implementation
- Cached ODM can provide additional performance
  - Can help in specific situations
  - Most configurations should stick with traditional ODM

ODM – Interface

- An i/o interface created by Oracle
  - VERITAS worked closely with Oracle in design of API during the implementation.
  - Not general-purpose API, but efficient for Oracle's i/o model
  - Interface is specific to Oracle

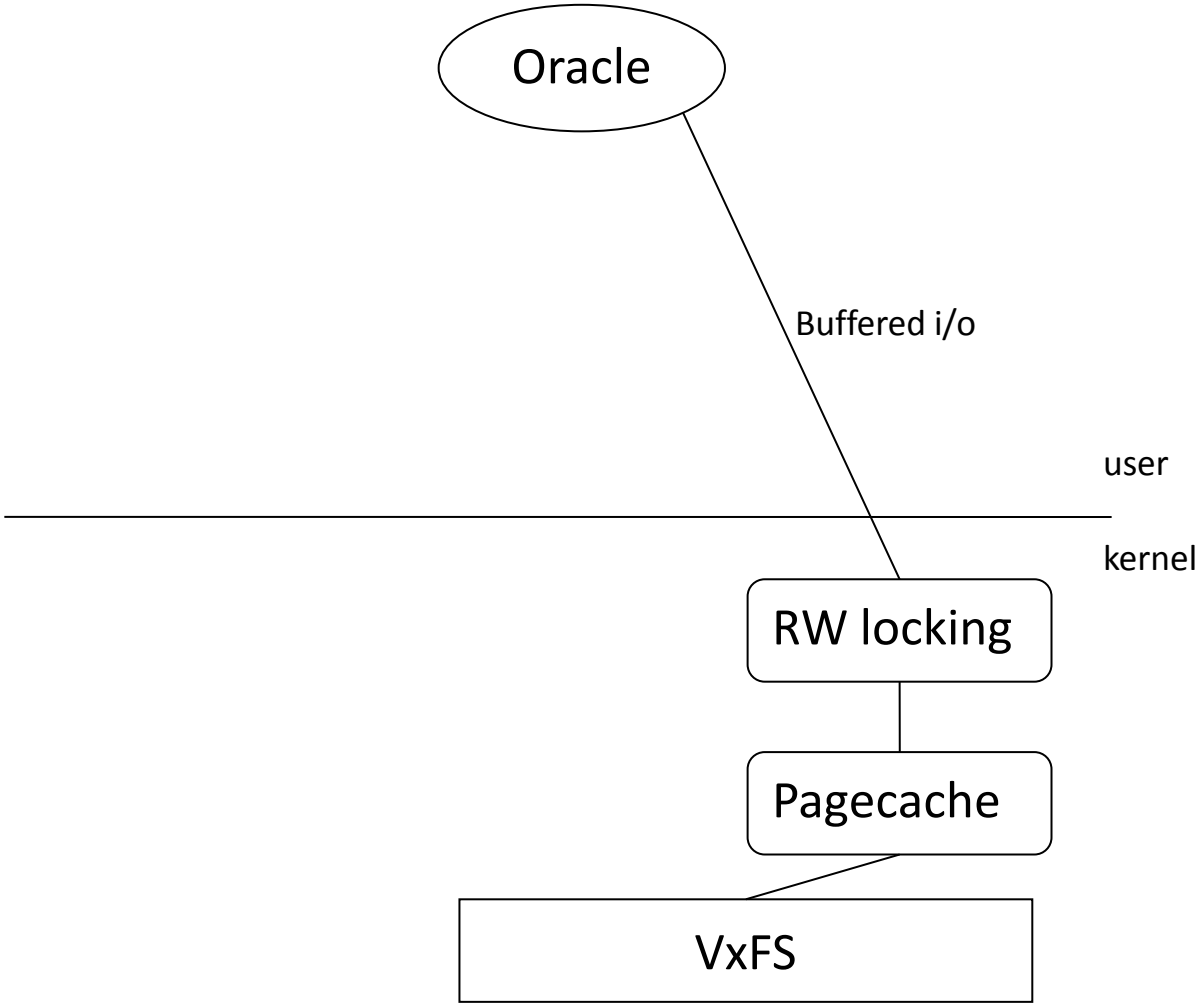
ODM Interface -- SmartSync

- File type optimizations -- SmartSync
  - ODM advises VxVM of i/o which does not require mirror resync
    - Each of those i/o's is faster
    - Recovery is faster

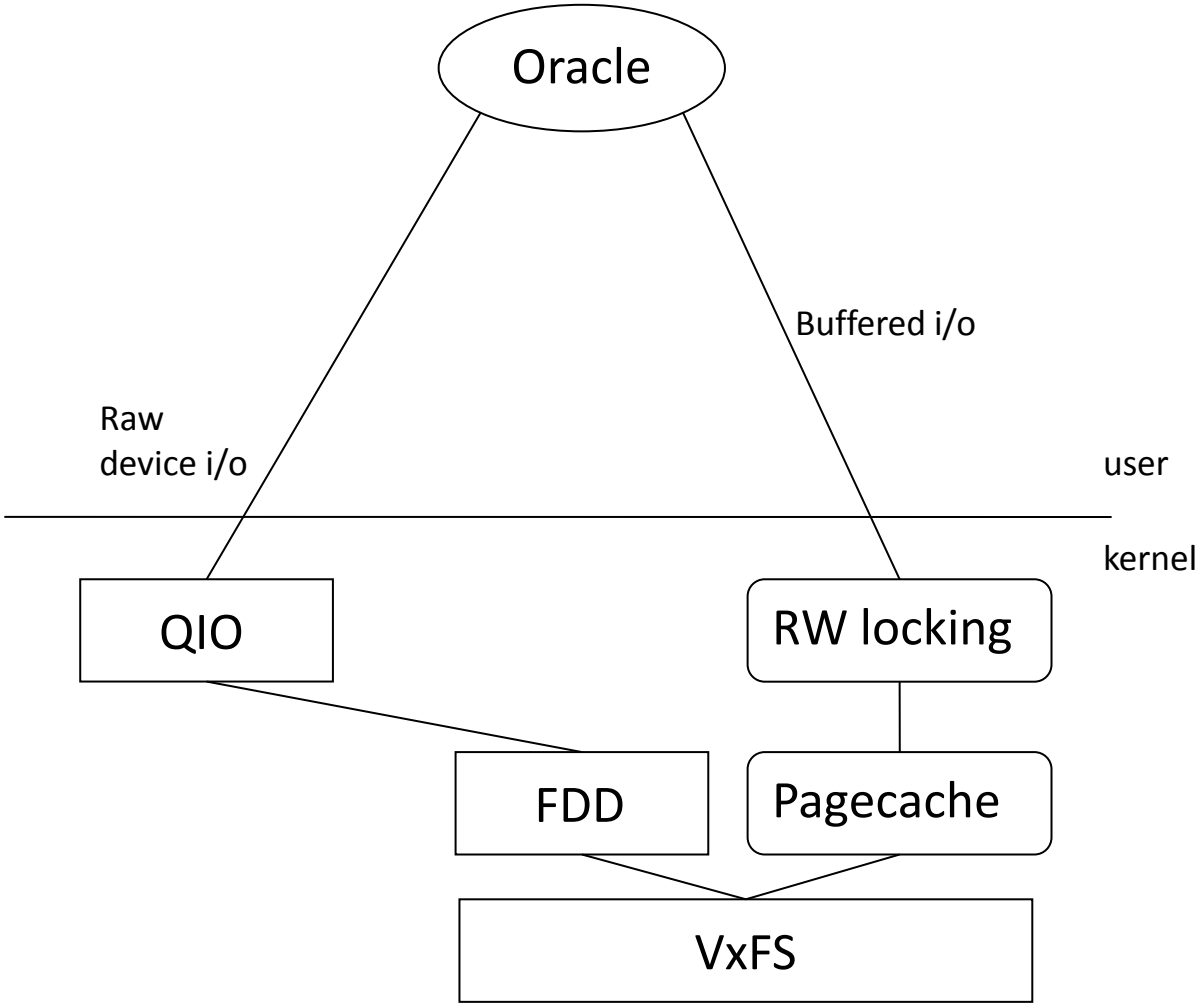
ODM – Implementation details

- Implementation has much in common with QIO
  - Does not use VxFS pagecache, so Oracle can use larger SGA
  - Bypasses Posix rwlock, relies on Oracle to avoid i/o conflicts
  - No read-aheads

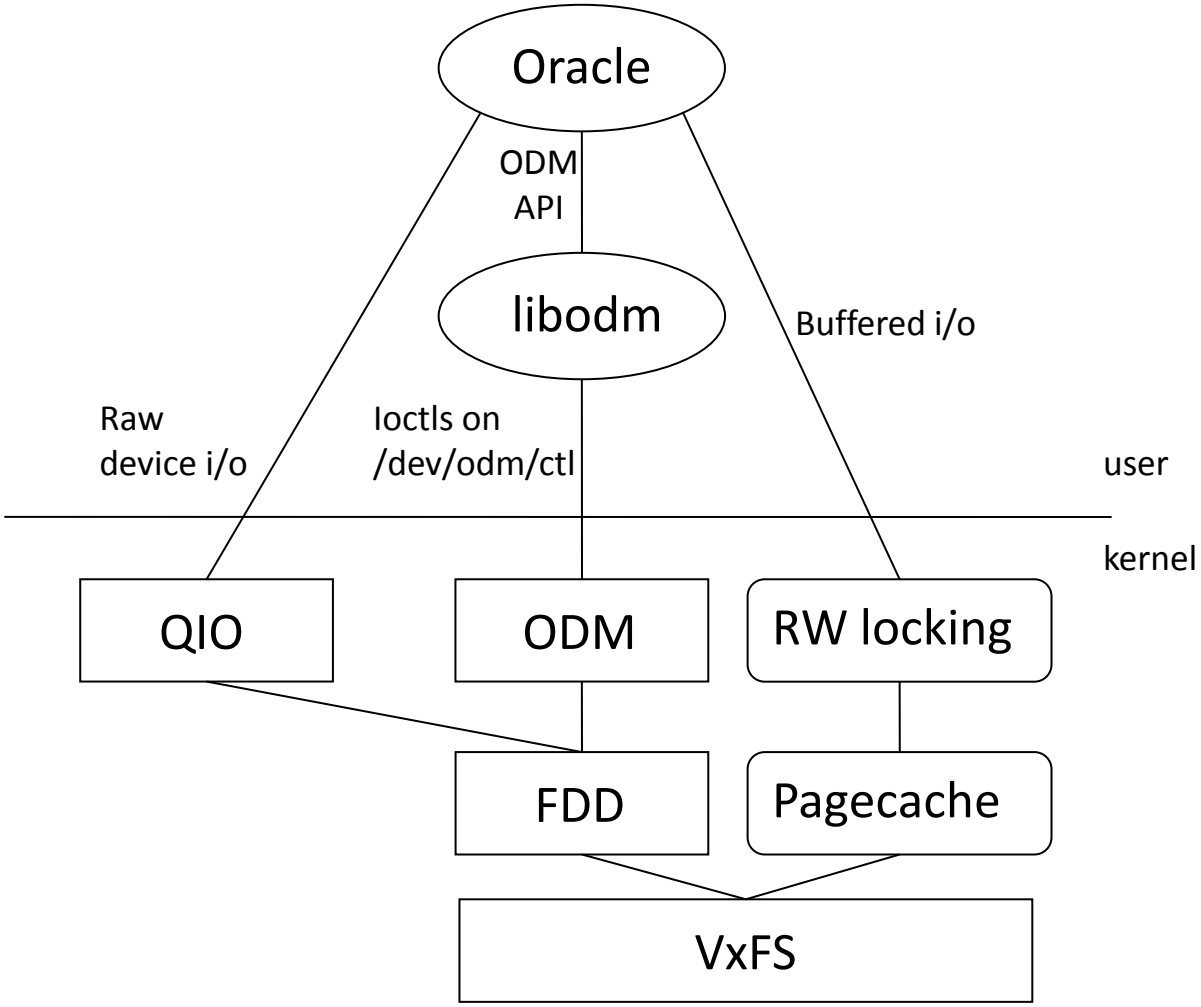
ODM Architecture



ODM Architecture



ODM Architecture



Configuring Oracle with ODM

- Oracle will use ODM for all file i/o (where it can), so will not make use of the pagecache

- Make the SGA as large as possible

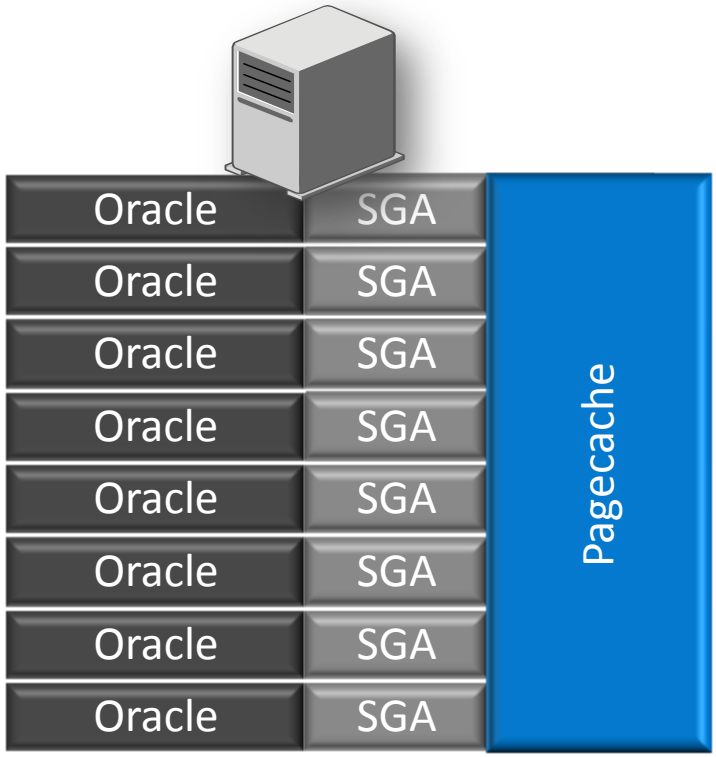
Cached ODM/QIO

- Uses pagecache, conditionally, for some files
- Caching and readahead can help in some special cases
- Not appropriate for most workloads
  - Need to take memory from SGA to give back to pagecache
  - In general, for Oracle performance, Oracle makes better use of memory than the pagecache.

Cached ODM/QIO use cases

- Memory available, but can't (or don't want to) increase the SGA
  - Instance Stacking, load changes with time
- Oracle isn't applying SGA memory effectively
  - Parallel query processes which do not use SGA
  - Extremely read-intensive workloads for which Oracle readahead is insufficient
- Interaction of ODM and non-ODM i/o
  - Pages are invalidated by ODM writes, but reloaded by non-ODM BIO reads. Cached ODM avoids invalidation. E.g., third-party replication.
- Activation of ODM has hurt performance
  - Often a symptom of not increasing SGA to compensate, but not always.

CODM use case: Pagecache with Instance Stacking



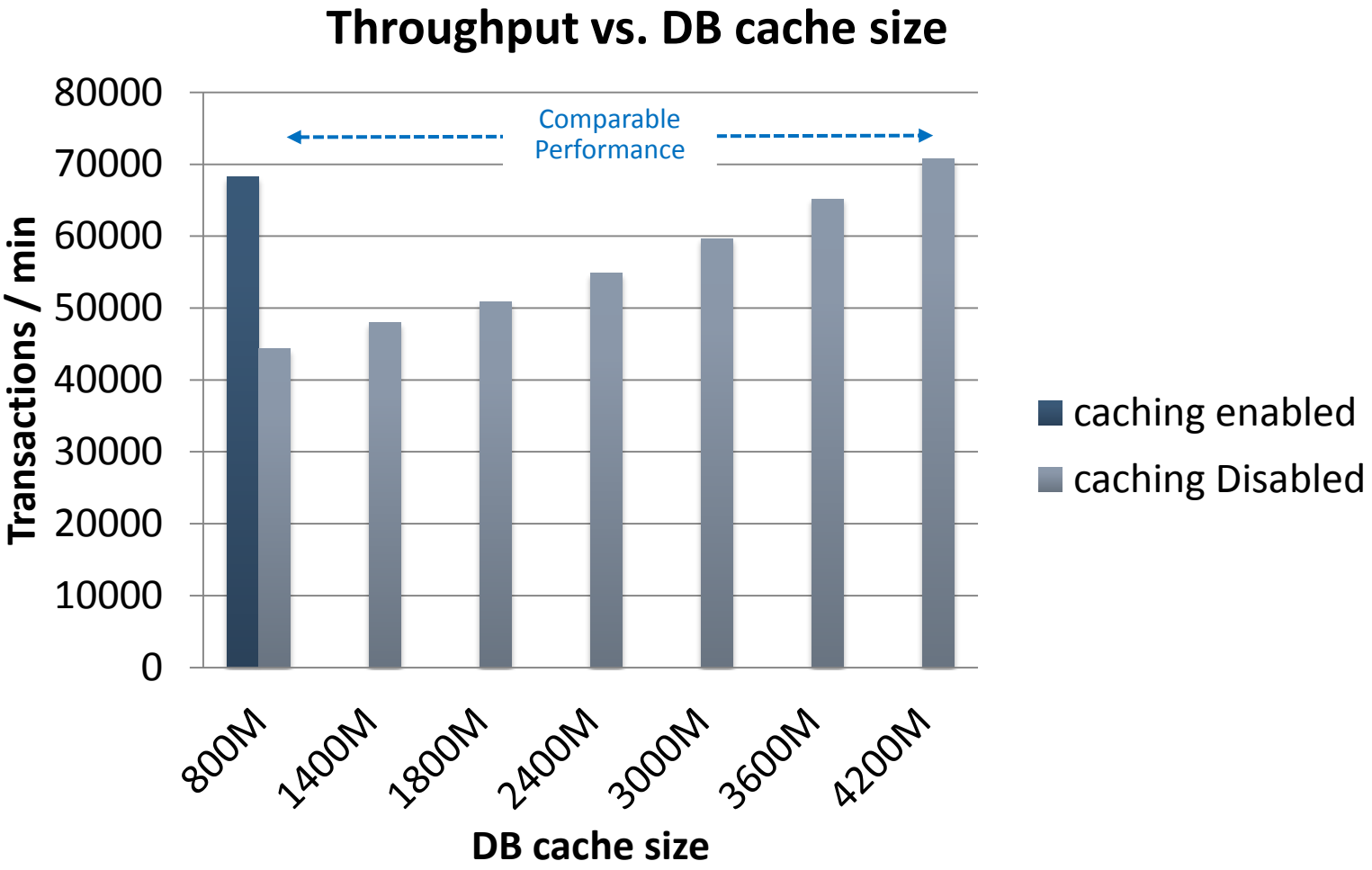
Shared Cache Efficiency

- Pagecache shared between Oracle instances
- CODM leverages pagecache for specific files
- Reduced fixed SGA size per instance

Failover configuration

- Multiple failover instances can be configured with small SGAs
- Pagecache enhances those instances which become active
- CODM supported on CFS for failover, but doesn't help RAC performance

CODM Performance: Parallel Instances



Cached ODM/QIO admin

- Activate caching on the filesystem

```
# vxtunefs -o odm_cache_enable=1 /mnt1
```

- Either turn caching on for individual files:

```
# odmadm setcachefile /mnt1/file1=on
```

- Or set the cachemap to affect caching for all files

```
# odmadm setcachemap file-type/io-type=cache-type
```

Cached ODM advisory tool

dbed_codm_adm

- Analyzes statistics from Oracle AWR
  - can use periodic AWR snapshots, if available
- Used to find candidate files for caching
  - Files that are heavily read, not a lot of writes
- Can also administer caching settings
  - Settings made with this tool are persistent – odmadm/qioadm settings need to be added to a configuration file.

Cached ODM config -- cachemap

- Alternate method of configuring CODM
- Conditional caching based on io-type/file-type combination.
- Cachemap applies to files with default cache settings:
  - on – cache everything
  - off – cache nothing
  - def – use global cachemap

Cached ODM/QIO admin – statistics

- odmstat/qiostat can display caching statistics for each file

```
# odmstat -l /mnt1/file

FILE NAME        NREADS   NWRITES     RBLOCKS     WBLOCKS   RTIME  WTIME
             NREQUESTIO   NDISKIO   HIT RATIO

/mnt1/file         2450         0        18515           0     6.1    0.0
                   2450      1179         56.0
```

Q&A