AIX/VIOS Disk and Adapter IO Queue Tuning

Dan Braden

IBM AIX Advanced Technical Skills

Version: 1.2
Date: July 8, 2014

Table of Figures:

Acknowledgements

Thank you to Brian Hart, Jim Allen, Steven Lang, Grover Davidson, Marc Olson, and others who've provided education and/or assistance on this subject.

- ***Disclaimers***

While IBM has made every effort to verify the information contained in this paper, this paper may still contain errors. IBM makes no warranties or representations with respect to the content hereof and specifically disclaim any implied warranties of merchantability or fitness for any particular purpose. IBM assumes no responsibility for any errors that may appear in this document. The information contained in this document is subject to change without any notice. IBM reserves the right to make any such changes without obligation to notify any person of such revision or changes. IBM makes no commitment to keep the information contained herein up to date.

- ***Trademarks***

The following terms are registered trademarks of International Business Machines Corporation in the United States and/or other countries: AIX, AIX/L, AIX/L (logo), IBM, IBM (logo), pSeries, Total Storage, Power PC.

The following terms are registered trademarks of International Business Machines Corporation in the United States and/or other countries: Power VM, Advanced Micro-Partitioning, AIX 5L, AIX 6 (logo), Micro Partitioning, Power Architecture, POWER5, POWER6, POWER 7, Redbooks, System p, System p5, System p6, System p7, System Storage.

A full list of U.S trademarks owned by IBM may be found at:
http://www.ibm.com/legal/copytrade.shtml

UNIX is a registered trademark in the United States, other countries or both.

- ***Feedback***

Please send comments or suggestions for changes to dbraden@us.ibm.com.

- ***Version Updates***

  - Version 1.0 - initial Techdoc

- Version 1.1 – reorganized and updated with new material
- Version 1.2 – added information about limit on NPIV virtual adapters for num_cmd_elems

Introduction

This paper is intended for IBM Power Systems customers, using AIX, IBM Technical Sales Specialists and consultants who are interested in learning more about tuning IO queues for optimal performance, and how to do it. If your application has a disk IO bottleneck, this paper will help you evaluate the problem and potentially improve performance.

The paper explains how IO queuing works, and explains how to tune the queues to improve performance, including in VIO environments. This will help ensure you don't have unnecessary IO bottlenecks at these queues. Thanks to Moore's law, disk IO is getting relatively slower compared to processors and memory, and becoming more and more a bottleneck to performance. Reducing IO latency from the application's point of view improves performance, and tuning these queues is important for high IOPS thruput. This document examines tuning the queues for the hdisk driver and adapter drivers, including in VIO environments. It doesn't examine IO tuning from the application to the hdisk driver.

This paper contains best practices which have been collected during the extensive period of time team colleagues and I have spent working in the AIX environment. It is focused on AIX versions 5.3, 6.1 and 7.1.

1. The AIX IO Stack and Queues

1.1. The AIX IO Stack

Following is the IO stack from the application to the disk:

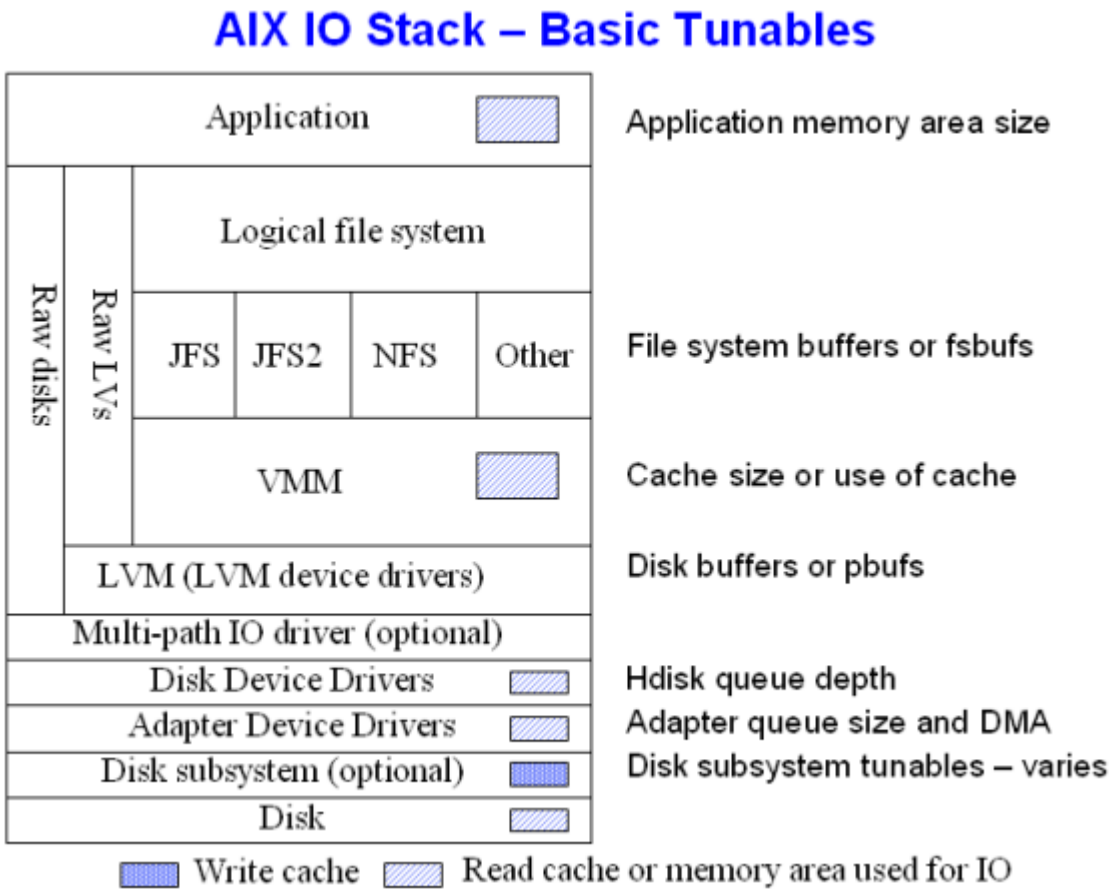

Figure 1 - AIX IO stack and basic tunables

This paper is concerned with tuning the AIX disk driver and adapter driver queue sizes. Note that even though the disk is attached to the adapter, the hdisk driver code is utilized before the adapter driver code. So this stack represents the order software and hardware come into play over time as the IO traverses the stack. The disk subsystem is typically SAN attached, and the disk subsystem will have its own internal IO stack. Note that this IO stack also exists in the VIOS, though there's no application, other than the VIOS function of virtualizing IOs from VIOCs and sharing IO adapters. The file system layers and VMM also aren't used for VIO virtualized IO except for file backed hdisks which aren't covered in this paper.

1.2. How IO Queues Work and Associated Attributes

AIX's disk and adapter drivers each use a queue to handle IO, split into an in-service queue, and a wait queue. IO requests in the in-service queue are sent to the storage, and the queue slot is freed when the IO is complete (AIX gets either the data for a read, or the acknowledgement for a write, and returns the result back up the stack). IO requests in the wait queue stay there until an in-service queue slot is free, at which time they are moved to the in-service queue and sent to the storage. IO requests in the in-service queue are also called in-flight from the perspective of the device driver.

The size of the hdisk driver in-service queue is specified by the queue_depth attribute, while the size of the adapter driver in-service queue is specified by the num_cmd_elems attribute. And generally in this paper the in-service queue size is also called the queue size, the queue depth, or number of queue slots. This paper will also refer to the traditionally used "adapter driver" - since adapters initially had one port, though now often have multiple ports - when it actually refers to the fcs device or specifically an adapter port's driver.

Here's how to show the fcs adapter port and hdisk attributes:

```
root # lsattr -EHl fcs0
attribute     value    description                       user_settable

intr_priority 3        Interrupt priority                False
lg_term_dma   0x800000 Long term DMA                     True
max_xfer_size 0x100000 Maximum Transfer Size             True
num_cmd_elems 200      Maximum Number of COMMAND Elements True
sw_fc_class   2        FC Class for Fabric               True

root # lsattr -EHl hdisk0
attribute       value                                 description           user_settable

PCM             PCM/friend/vscsi                      Path Control Module        False
algorithm       fail_over                             Algorithm                  True
hcheck_cmd      test_unit_rdy                         Health Check Command       True
hcheck_interval 60                                    Health Check Interval      True
hcheck_mode     enabled                               Health Check Mode          True
max_transfer    0x40000                               Maximum TRANSFER Size      True
pvid            00c4c6c7b35f297700000000000000000     Physical volume identifier False
queue_depth     3                                     Queue DEPTH                True
reserve_policy  no_reserve                            Reserve Policy             True
```

Attributes in blue are of interest regarding tuning the queues. Both physical and virtual disk drives typically support command tagged queuing, which allows us to have multiple in-flight IO requests to a single hdisk. This improves performance in several ways.

A physical disk can only do one IO at a time, but knowing several of the IO requests allows the disk to do the IOs using an elevator algorithm to minimize actuator movement and latency. Virtual disks typically are backed by many physical disks, so can do many IOs in parallel. The elevator analogy is appropriate for individual physical disk drives, where passengers are IO requests, the elevator is the disk arm, and the floors are the disk tracks. If the elevator went to the floors in the order the buttons were pushed (or the or-

der of IO requests), rather than the way they normally work, the average service time for each passenger would be longer. Thus, submitting multiple in-flight IOs to a disk subsystem allows it to figure out how to get the most thruput and fastest average IO service time. Increasing the queue size is like increasing the passenger capacity of the elevator.

Theoretically, the maximum IOPS one might achieve for a LUN is:

Maximum LUN IOPS = queue_depth/ (avg. IO service time)

Thus, increasing queue_depth, and consequently the number of in-flight IOs, increases potential IOPS and thruput. Similarly:

Maximum adapter IOPS = num_cmd_elems/ (avg. IO service time)

The maximum in-flight IOs a system will submit to SAN storage is the smallest of the following

- The sum of the hdisk queue_depths
- The sum of the adapter num_cmd_elems
- The maximum number of in-flight IOs submitted by your application(s)

As IOs traverse the IO stack, AIX needs to keep track of them at each layer. So IOs are essentially queued at each layer, and using queue slots or buffers to keep track of them. Generally, some number of in-flight IOs may be issued at each layer and if the number of IO requests exceeds that number, they reside in a wait queue until the required resource become available. So there is essentially an in-service queue and a wait queue at each layer, with the size of the in-service queue limiting the number of in-flight IOs.

While this paper doesn't examine IO tuning the from the hdisk driver to the application, one can also do IO tuning at the file system layer, as file system buffers limit the maximum number of in-flight IOs for each file system. Also at the LVM device driver layer, hdisk buffers, pbufs, limit the number of in-flight IOs from that layer. The hdisks have a maximum number of in-flight IOs that's specified by its queue_depth attribute. And FC adapters also have a maximum number of in-flight IOs specified by num_cmd_elems. The disk subsystems themselves queue IOs and individual physical disks can accept multiple IO requests but only service one at a time. There are many queues within the operating system and the hardware, including with TCP/IP, memory, and even the queue of commands to use the processors.

Here is one type of hdisk's default attributes (attributes vary across different disk subsystems):

```
# lsattr -El hdisk33
PR_key_value      none        Reserve Key                    True
location                      Location Label                 True
lun_id            0x5515000000000000 Logical Unit Number ID True
lun_reset_spt     yes         Support SCSI LUN reset         True
max_transfer      0x40000     N/A                            True
node_name         0x5005076300c096ab FC Node Name            False
pvid              none        Physical volume identifier     False
q_type            simple      Queuing TYPE                   True
qfull_dly         20          delay in seconds for SCSI TASK SET FULL True
```

```
queue_depth       20          Queue DEPTH                  True
reserve_policy    single_path Reserve Policy               True
rw_timeout        60          READ/WRITE time out value    True
scbsy_dly         20          delay in seconds for SCSI BUSY True
scsi_id           0x620713    SCSI ID                      True
start_timeout     180         START unit time out value    True
ww_name           0x5005076300cd96ab FC World Wide Name    False
```

The default queue_depth is 20, but can be changed to as high as 256 as shown with:

```
# lsattr -Rl hdisk33 -a queue_depth
1...256 (+1)
```

This indicates the value can be anywhere from 1 to 256 in increments of 1. In general for device attributes that have a user_settable value of True, you can display allowable values using:

```
# lsattr -Rl <device> -a <attribute>
```

Here's a FC adapter's attributes:

```
# lsattr -El fcs0
bus_intr_lvl      65703       Bus interrupt level     False
bus_io_addr       0xdec00     Bus I/O address         False
bus_mem_addr      0xe8040000  Bus memory address      False
init_link         al          INIT Link flags         True
intr_priority     3           Interrupt priority      False
lg_term_dma       0x800000    Long term DMA           True
max_xfer_size     0x100000    Maximum Transfer Size   True
num_cmd_elems     200   Maximum number of COMMANDS to queue to the adapter True
pref_alpa         0x1         Preferred AL_PA         True
sw_fc_class       2           FC Class for Fabric     True
```

Currently default queue sizes (num_cmd_elems) for FC adapters range from 200 to 500, with maximum values of 2048 or 4096.

The FC adapter also directly accesses a configurable amount of system memory to hold in-flight IO data, which also acts as a queue since if there isn't sufficient free memory to hold the data, then the IO will be blocked until there is. This DMA memory area is controlled by the max_xfer_size attribute, and controls two separate things; thus, is a bit complex. This attribute's value specifies the max IO size that the adapter will send to the disk subsystem (provided the LUN's max_transfer attribute isn't smaller). The size of the DMA memory area is not equal to the max_xfer_size value. By default for the 8 Gb dual port adapter, with the default max_xfer_size of 0x40000=256 KB, the DMA memory area size is 16 MB. Using any other allowable value for max_xfer_size increases the memory area to 128 MB. To get the full bandwidth of the adapter, one needs the DMA memory area of 128 MB, though for many customers, the default is sufficient. Changing this value isn't always possible, depends on the system hardware, and has some risks discussed in the next section.

Since the DMA memory area sizes supported varies across adapters, you can use:

```
# fcstat –D <fcs#>
```

To see the value of the DMA memory size of your adapter, if you have a recent version of AIX or of the VIOS OS. This allows you to try the alternative max_xfer_size settings, and see if the DMA pool size changes. Sample output is shown in section 2.3 on fcstat.

1.3. Changing Device Attribute Values

In general one changes device attribute values using the chdev command:

```
# chdev –l <device> -a <attribute>=<new value>
```

However, to change these attributes for hdisk and fcs devices, they must not be in use. Thus, typically these values are changed during setup or planned maintenance. One can make the change in the ODM via the –P flag, so that the change goes into effect when the system is rebooted or the device is reconfigured:

```
# chdev –l <device> -a <attribute>=<new value> -P
```

Be aware that after making the change to the attribute in the ODM, the lsattr command displays the value in the ODM, not the value being used by the kernel. You'll need to keep track of what's actually in use by the kernel for tuning.

Changing hdisk attributes is possible provided they aren't in a varied on VG, or opened and accessed via other applications.

Adapter attributes can be changed provided no disks are using the adapter. Assuming one has multiple paths to all disks via redundant adapters and you're using multi-path code compliant with the AIX PCM, then you can dynamically change the adapter attributes: you stop using one adapter by putting all disk paths using the adapter into the Defined state, make changes to the adapter attributes, reconfigure the adapter, then reconfigure the paths back into the Available and enabled state via the cfgmgr command. This can be achieved via:

```
# rmdev –Rl <fcs#>
# chdev –l <fcs#> -a <attribute>=<new value>
# cfgmgr –l <fcs#>
```

Due to limited memory on PCI Host Bridge (PHB) chips, it's not always possible to increase max_xfer_size for all FC adapters. This is less of an issue on newer hardware, and on older systems where fewer adapters share PHBs, so it is configuration dependent. The only way to find out if you can change the value for each adapter is to try changing it. However, be aware that if you're booting from SAN and your configuration doesn't have sufficient PHB memory, this can cause boot failure. Lack of PHB memory results in hdisk devices configuring into a Defined, rather than Available state, and if your boot disk is one of them, boot will fail. Recovery involves going into SMS and changing the value back to the default, or restoring from a mksysb.

Assuming you can boot, failure to configure hdisks or paths into the Available state will lead to errors in the error log that look like:

```
 LABEL: DMA_ERR
 IDENTIFIER: 00530EA6
 …
 Class: H
 Type: UNKN
 Resource Name: PCIDMA
 ...
 Description
UNDETERMINED ERROR

Probable Causes
SYSTEM I/O BUS
SOFTWARE PROGRAM
ADAPTER
DEVICE

Recommended Actions
PERFORM PROBLEM DETERMINATION PROCEDURES
```

If you get these errors, you'll need to change the max_xfer_size back to the default value.

1.4. Multi-path IO Code and Queue Tuning

The multi-path IO code for the storage is closely integrated with the hdisk and adapter drivers, as it picks a path that specifies the host port and storage port that will handle the IO. The disk vendor specifies what multi-path code may be used with their storage, and in many cases you have a choice, even with IBM storage

Thus, tuning the queues means you need to know about the multi-path code, how it affects the queues, and what reports it provides to facilitate tuning. This paper mainly discusses tuning from the perspective of IBM AIX supported multi-path code which includes, the AIX PCM (aka. MPIO), SDDPCM and SDD.

Note that SDDPCM is compliant with the MPIO architecture, as is the AIX PCM which comes with AIX, so standard AIX IO and path management commands work with both. This includes commands such as iostat, lspath, chpath, rmpath, etc.

This paper doesn’t cover OEM multi-path code that isn’t compliant with the MPIO architecture. Please refer to other vendor’s documentation for their multi-path code.

1.4.1. SDD and SDDPCM

SDD is no longer strategic, isn’t compliant with the MPIO architecture, and the author recommends changing to either SDDPCM or the AIX PCM depending on what’s supported for the specific model of storage.

SDD adds a layer of queuing, while SDDPCM just chooses a path for each IO.

SDD has a vpath device for each logical disk plus an hdisk device for each path to it, while with SDDPCM we just have an hdisk with paths to it listed with the lspath command. There is also a dpo device for SDD. Here's the dpo device's attributes for one release of SDD:

```
#                      lsattr                          -El                                dpo
Enterpr_maxlun    600 Maximum LUNS allowed for Enterprise Products      True
Virtual_maxlun    512 Maximum LUNS allowed for Virtualization Products  False
persistent_resv   yes Subsystem Supports Persistent Reserve Command     False
qdepth_enable     yes Queue Depth Control                               True
```

When qdepth_enable=yes, SDD will only submit queue_depth IOs to any underlying hdisk (where queue_depth here is the value for the underlying hdisk's queue_depth attribute). When qdepth_enable=no, SDD just passes on the IOs directly to the hdisk driver. So the difference is, if qdepth_enable=yes (the default), IOs exceeding the queue_depth will queue at SDD, and if qdepth_enable=no, then IOs exceed the queue_depth will queue in the hdisk's wait queue. In other words, SDD with qdepth_enable=no and SDDPCM do not queue IOs and instead just pass them to the hdisk drivers. Note that at SDD 1.6, it's preferable to use the datapath command to change qdepth_enable, rather than using chdev, as then it's a dynamic change, e.g., datapath set qdepth disable will set it to no. Some releases of SDD don't include SDD queuing, and some do, and some releases don't show the qdepth_enable attribute. Either check the manual for your version of SDD or try the datapath command to see if it supports turning this feature off.

With SDD one can submit queue_depth x # paths to a LUN, while with SDDPCM, one can only submit queue_depth IOs to the LUN. Thus, if you switch from SDD using 4 paths to SDDPCM, then you'd want to set the SDDPCM or AIX PCM hdisks' queue_depth to 4x that of SDD hdisks for an equivalent effective queue depth.

2. Tools to Monitor the Queues

Basic commands to monitor the queues include:

- When using the AIX PCM or multi-path code compliant with the MPIO architecture:
  - iostat for hdisk driver queues
  - sar for hdisk driver queues
  - fcstat for adapter driver queues
  - topas
  - nmon (interactive or via NMON recordings)
- When using SDDPCM
  - pcmpath
- When using SDD
  - datapath

Note you will still use iostat, sar, fcstat, topas and nmon with SDD and SDDPCM.

2..1 The iostat Command

For AIX iostat is the basic tool to monitor the hdisk driver queues. The iostat -D command generates output such as:

```
hdisk6 xfer: %tm_act bps   tps bread bwrtn
                4.7  2.2M 19.0  0.0   2.2M
read: rps avgserv minserv maxserv timeouts fails
      0.0    0.0     0.0   0.0      0       0
write: wps avgserv minserv maxserv timeouts fails
      19.0  38.9     1.1     190.2   0       0
queue: avgtime mintime maxtime avgwqsz avgsqsz sqfull
         15.0     0.0    83.7    0.0    0.0     136
```

avgwqsz - average wait queue size
avgsqsz - average service queue size
avgtime - average time spent in the wait queue in ms
sqfull – rate of IOs submitted to a full queue per second

The sqfull value has changed from initially being a count of the times we've submitted an IO to a full queue, to now where it's the rate of IOs per second submitted to a full queue. The example report shows the prior case (a count of IOs submitted to a full queue), while newer releases typically show lower values and decimal fractions indicating a rate.

It's nice that iostat -D separates reads and writes, as we would expect the IO service times to be different when we have a disk subsystem with cache. This helps us evaluate the IO service times more accurately. The most useful report for tuning is just running "iostat -Dl" which shows statistics since system boot, as-

suming the system is configured to continuously maintain disk IO history (run # lsattr -El sys0, or # smitty chgsys to see if the iostat attribute is set to true). The author's preferred iostat command flags are:

```
# iostat -RDTl [ <interval> [ <#intervals> ]]
```

This lists the data in a long format (one line per hdisk). The -R flag resets minimum/maximum values for each interval, the –T adds a time stamp, and generates output that is too wide to display here in this font, but it contains the data above from iostat –D on one line for each hdisk. Here's sample output:

```
# iostat -RDTl

System configuration: lcpu=12 drives=3 paths=8 vdisks=4

Disks:                   xfers                                read                                write                                queue                   time
-------------- ------------------------------- ------------------------------------ ------------------------------------ ------------------------------------ ---------
               %tm    bps   tps  bread  bwrtn   rps    avg    min    max time fail   wps    avg    min    max time fail    avg    min    max   avg   avg  serv
               act                                  serv   serv   serv outs              serv   serv   serv outs              time   time   time  wqsz  sqsz qfull
hdisk0         0.5   4.6K   0.7   1.2K   3.4K   0.1   1.2    0.1   30.0     0    0   0.6   8.3    1.2   69.4     0    0  18.1    0.0  142.4   0.0   0.0   0.2  06:20:54
hdisk1         0.0  25.0K   2.9  25.0K  52.1    2.9   0.4    0.1  218.9     0    0   0.0   5.2    0.8  248.4     0    0   0.0    0.0    0.0   0.0   0.0   0.0  06:20:54
hdisk2         0.0   3.0K   0.4   3.0K   0.0    0.4   0.2    0.1  194.1     0    0   0.0   0.0    0.0    0.0     0    0   0.1    0.0  194.1   0.0   0.0   0.4  06:20:54
```

Metrics in blue are the main ones examined for tuning the queues.

2..2 The sar Command

The sar -d command generates output such as:

```
16:50:59     device    %busy    avque    r+w/s    Kbs/s   avwait   avserv
16:51:00     hdisk1      0      0.0        0        0      0.0      0.0
             hdisk0      0      0.0        0        0      0.0      0.0
```

avwait – average time spent in the wait queue in ms
avserv – average IO service time in ms
avque – average number of IOs in the wait queue

This command doesn't distinguish between read and write service times, unlike iostat.

2..3 The fcstat Command

For fibre channel (FC) adapter queues, the fcstat command is used to look for blocked IOs in the adapter's queues, e.g.:

```
# fcstat fcs0

FIBRE CHANNEL STATISTICS REPORT: fcs0
...
Seconds Since Last Reset: 7597342
...
FC SCSI Adapter Driver Information
   No DMA Resource Count: 0
   No Adapter Elements Count: 104848
   No Command Resource Count: 13915968
...
```

The value No Command Resource Count is the number of times, since the adapter was configured (typically boot time), that an IO was temporarily blocked waiting for resources, such as an adapter buffer due to a num_cmd_elems attribute value that is too low. Non-zero values indicate that increasing num_cmd_elems may help improve IO service times. Of course if the value increments slowly, then the improvement may be very small, while quickly incrementing values means tuning is more likely to have a measurable improvement in performance. One can calculate the rate of blocked IOs as the blocked count divided by the seconds since last reset. Here we're getting no command resource blocks at a rate of about 2 per second. Assuming we reduce latency by 1 ms by tuning the queue, overall savings in an hour would be:

2 blocks/s x 1 ms/block x 3600 s/hr = 7.2 sec each hour

So this isn't a significant bottleneck and may not be worth worrying about. Typically the amount of time waiting for a buffer to free is very small. We are waiting for the storage to complete next in-flight IO.

The fcstat has recently been enhanced with the –D flag which produces output as follows:

```
   # fcstat -D fcs0

   FIBRE CHANNEL STATISTICS REPORT: fcs0

   Device Type: 8Gb PCI Express Dual Port FC Adapter (df1000f114108a03)
   (adapter/pciex/df1000f114108a0)
...
World Wide Node Name: 0x20000120FA0B9ED6
   World Wide Port Name: 0x10000090FA0B9ED6
   ...
   Port Speed (supported): 8 GBIT
   Port Speed (running):   4 GBIT
   Port FC ID: 0x6a0000
   Port Type: Fabric

   Seconds Since Last Reset: 7597342

           Transmit Statistics     Receive Statistics
           -------------------     ------------------
   Frames: 18892318                16100927
   Words:  6621450752              2363730688
   ...
     I/O DMA pool size:  0x1000000

     FC SCSI Adapter Driver Queue Statistics
       Number of active commands:   0
       High water mark of active commands:   180
       Number of pending commands:   0
       High water mark of pending commands:   91
       Number of commands in the Adapter Driver Held off queue:  0
       High water mark of number of commands in the Adapter Driver Held off queue:  0

     FC SCSI Protocol Driver Queue Statistics
       Number of active commands:   0
```

```
        High water mark of active commands:   180
        Number of pending commands:   2
        High water mark of pending commands:   78

...
FC SCSI Adapter Driver Information
  No DMA Resource Count: 0
  No Adapter Elements Count: 0
  No Command Resource Count: 1711973
...
Adapter Effective max transfer value:   0x100000
```

These statistics are especially useful in tuning the adapter queues. Items highlighted in blue show the statistics of interest and other useful information including the adapter type, WWPN, supported speed, the running speed, the size of the DMA pool memory area, and the max_xfer_size value currently used by the kernel.

2..4 The Interactive topas Command

Starting the interactive topas command with:

```
# topas -D
```

Or just pressing D while in topas will yield this disk detail panel:

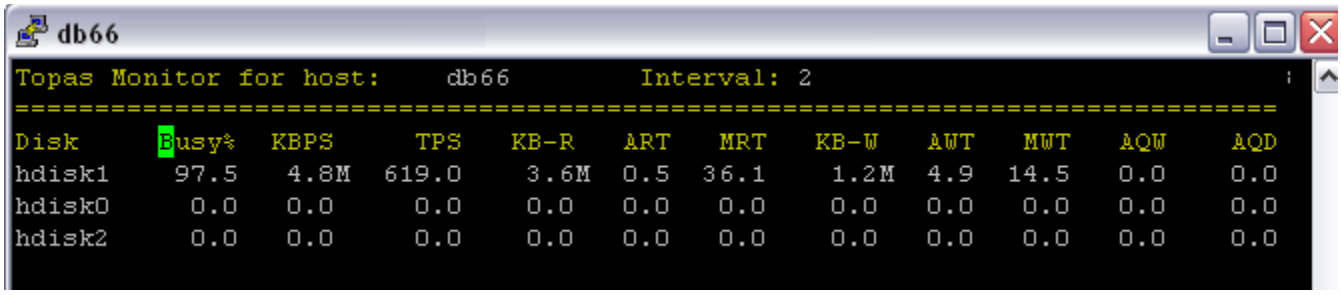

Figure 2 - topas -D sample output

KB-R = KB read
ART = Average Read IO service Time
KB-W = KB Written
AWT = Average Write IO service Time
AQW = Average Queue Wait = average time waiting in the queue
AQD = Average Queue Depth = average number of IOs in the wait queue

This command, like iostat, reports both read and write IO service times, and queue wait time as well.

Pressing the **d** key while in the disk detail panel, will alternate from the disk detail panel to the adapter panel:

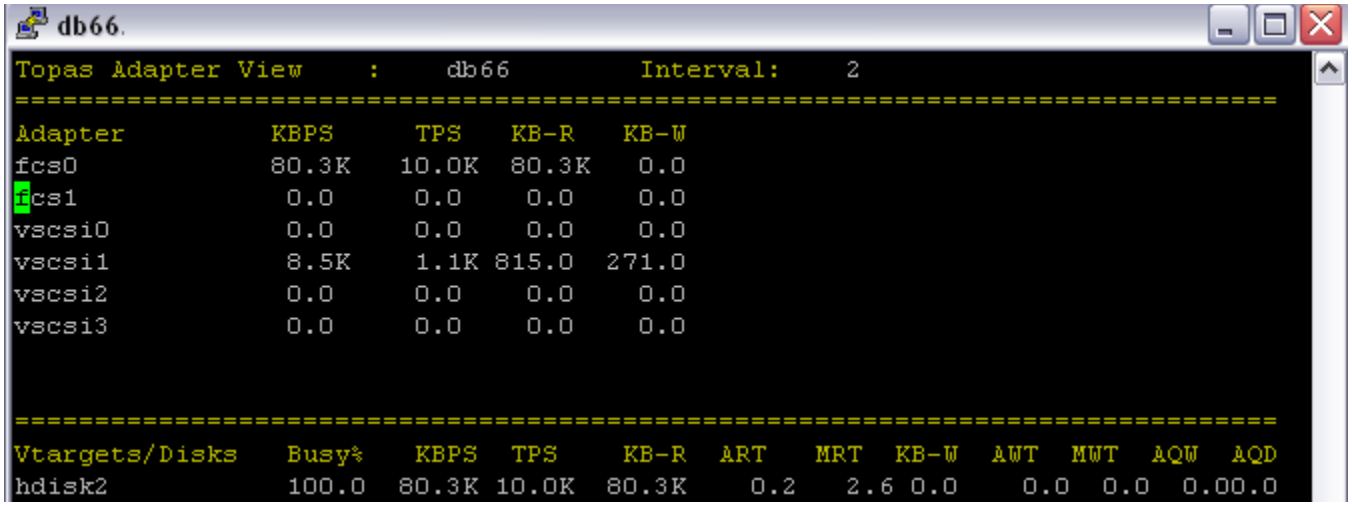

Figure 3 - topas adapter panel

Pressing **d** then **v** will show the virtual adapter panel:

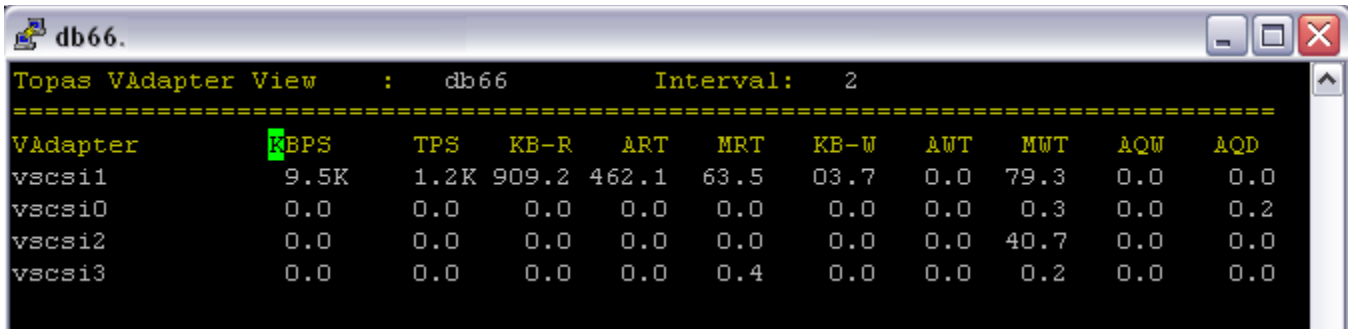

Figure 4 - topas virtual adapter panel

Note that the performance metrics refer to average values for the LUNs attached to the adapter. Thus the AQW and AQD refer to wait time for the LUNs in their hdisk driver queues, not wait time in the adapter queue.

2..5 The Interactive nmon Command

One enters the interactive nmon tool via:

```
# nmon
```

Pressing **D** will cycle thru four different disk screens, the first showing each disk's thruput, the second showing the disk size, number of paths and an adapter connected to the disk, the third screen shown below, and the fourth screen shows disk thruput statistics with a graph indicator of thruput. Here's the third screen showing IO service times:

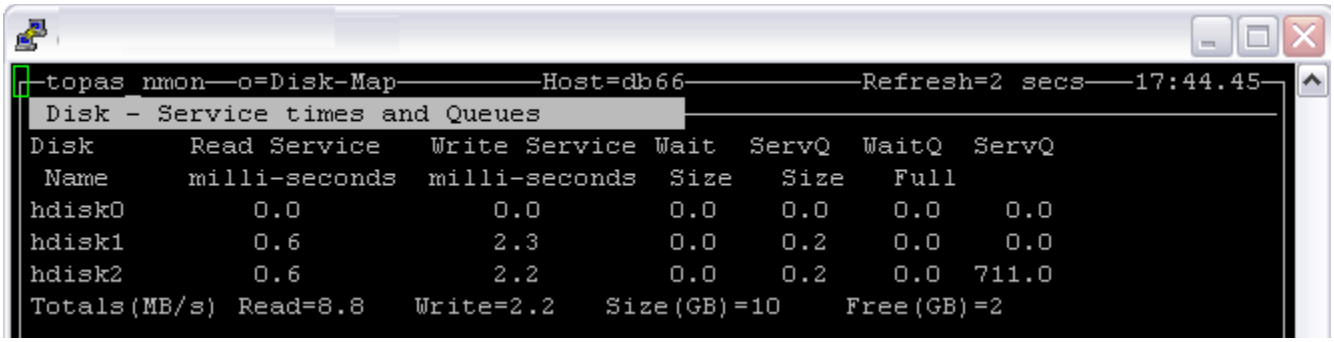

Figure 5 - interactive nmon disk IO service statistics

Wait = average wait time in the queue in ms
ServQ Size = Average service queue size
WaitQ Size = Average wait queue size
ServQ Full = Number of IO requests sent to a full queue for the interval

If you've a lot of hdisks, you can press the "**.**" subcommand to just show the busy disks.

2..6 NMON Recordings

One can setup NMON recordings from smit via:

```
# smitty topas -> Start New Recording -> Start local recording -> nmon
```

However, the smit panel doesn't list the option we need to get disk IO service times, specifically the –d option to collect disk IO service and wait times. Thus, one needs to use the command line with the nmon command to collect and report these statistics. Here's one set of options for collecting the data:

```
# nmon -AdfKLMNOPVY^ -w 4 -s <interval> -c <number of intervals>
```

The key options here from a disk viewpoint include:

-d collect and report IO service time and wait time statistics
-^ includes the FC adapter section (which also measures NPIV traffic on VIOS FC adapters)
-V includes the disk volume group section

To get the recording thru the NMON Analyzer tool (a spreadsheet tool that runs on PCs and generates performance graphs, other output, and is available via the internet), it's recommended to keep the number of intervals less than 300. Samples of the main reports of interest for IO queue tuning follow:

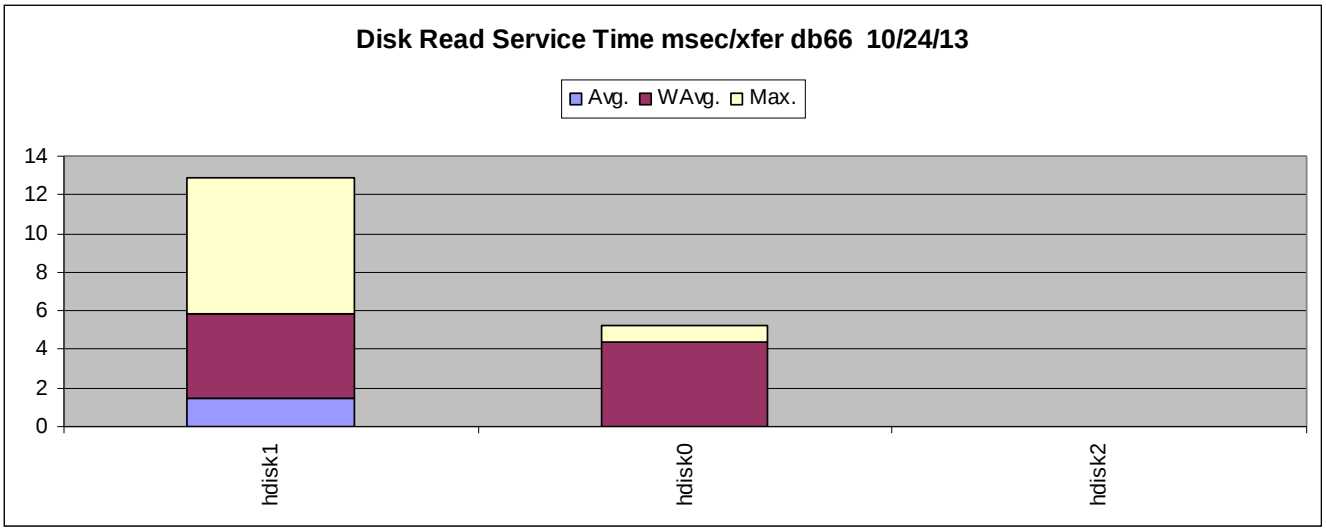

Figure 6 - NMON analyzer disk read IO latency

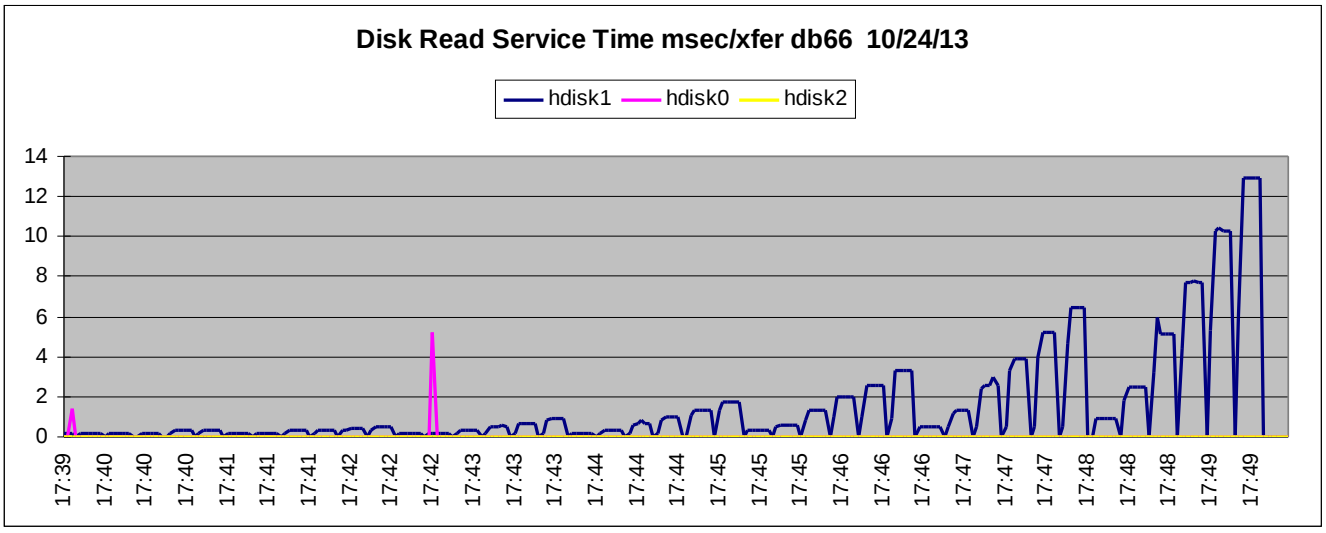

Figure 7 - NMON analyzer disk read IO latency over time

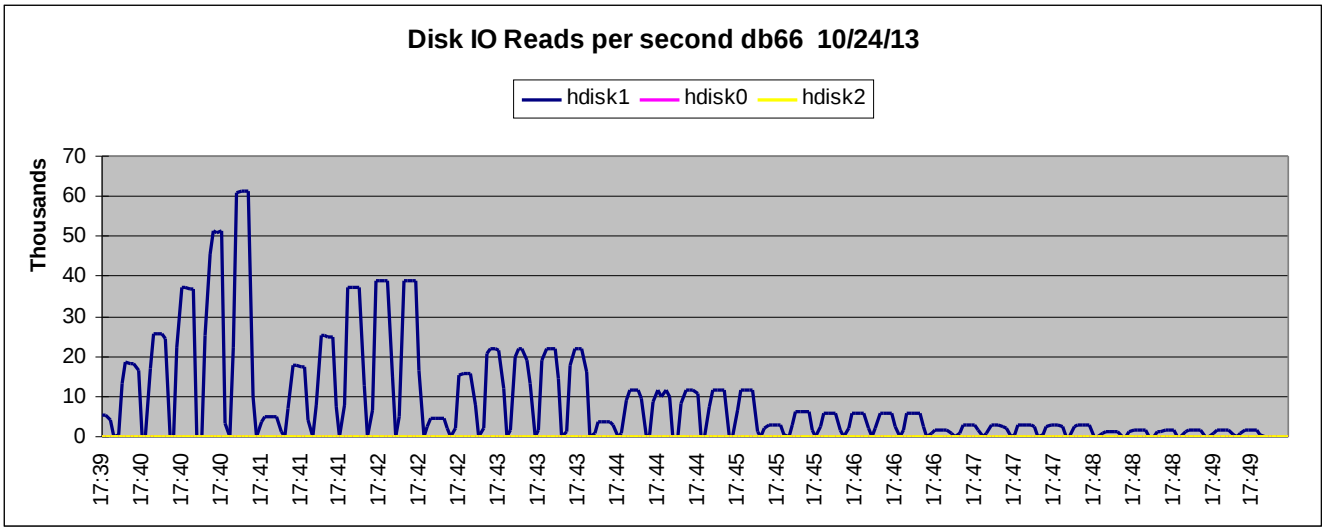

Figure 8 - NMON analyzer disk read IOPS over time

Similar reports exist for write service times (not shown). Also provided are graphs of average wait time in the hdisk driver queue for IOs:

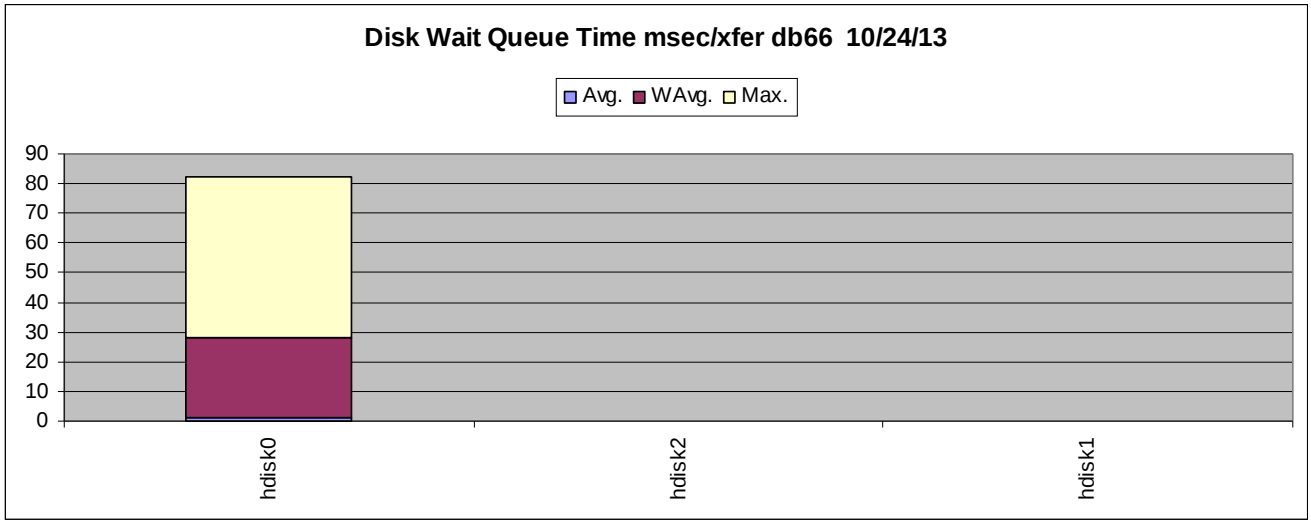

Figure 9 - NMON analyzer disk queue wait time

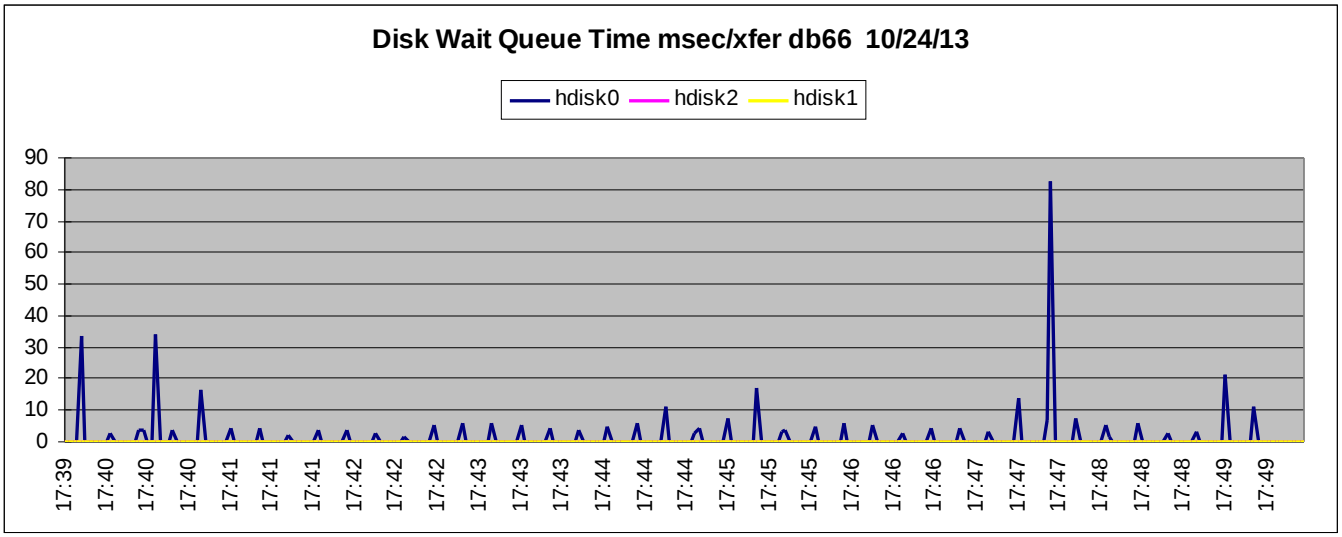

Figure 9 - NMON analyzer disk queue wait time, over time

The raw data is also available in case it's difficult to read the graphs, e.g., the average disk wait time for hdisk0 (the nearly invisible very narrow blue bar in figure 9, is 1.3 ms.). NMON recordings have the benefit that you can quickly see when the hdisk queue is overflowing, the duration, and how long the IOs are waiting in the queue.

2..7 The SDDPCM pcmpath Command

SDDPCM provides the "pcmpath query devstats" and "pcmpath query adaptstats" commands to show hdisk and adapter queue statistics. You can refer to the SDD/SDDPCM manual for syntax, options, explanations of all the fields and many other useful commands including the ability to disable all paths to a specific host/storage port with one command. Here's the output of "pcmpath query devstats" for one hdisk:

```
Device #: 0  DEVICE NAME: hdisk2
================================
       Total Read Total Write Active Read Active Write Maximum
I/O:    29007501     3037679           1            0          40
SECTOR: 696124015  110460560           8            0       20480

Transfer Size: <= 512  <= 4k    <= 16K  <= 64K   > 64K
                21499 10987037 18892010 1335598 809036
```

and here's output for one adapter from "pcmpath query adaptstats"

```
Adapter #: 0
=============
          Total Read Total Write Active Read Active Write Maximum
I/O:        439690333 24726251           7       0             258
SECTOR:     109851534 960137182        608       0          108625
```

Here, we're mainly interested in the Maximum field. For the hdisks, the maximum value represents the maximum in-flight IOs sent to the LUN, and this won't exceed queue_depth. For adapters, the maximum value represents the maximum number of IOs submitted to the adapter over time, and this can exceed num_cmd_elems. Thus, the adapter maximum tells us the value we should assign to num_cmd_elems.

2..8 The SDD datapath Command

SDD provides the datapath command, which is very similar to the pcmpath command where we use the output of "datapath query devstats" and "datapath query adaptstats" to examine device statistics.

3. Tuning the Queues

3.1 Check for SAN/Storage Problems First

Before tuning the queues, you should make sure there aren't problems with the SAN or the storage inhibiting performance. Fixing these problems can resolve queue wait issues, as reflected in this formula:

Max LUN IOPS = queue_depth / <avg IO service time>

If we reduce the IO service time by fixing the SAN/storage, max IOPS will increase, and IOs won't be using the queue slot as long, reducing usage of the queue slots, and reducing wait time in the queue. Faster IOs from the storage can fix your queue wait problem without adjusting the queue sizes.

The fcstat command is used to examine FC link health where it shows the following statistics:

```
# fcstat fcs0

FIBRE CHANNEL STATISTICS REPORT: fcs0
…
Seconds Since Last Reset: 557270
…
LIP Count: 0
NOS Count: 0
Error Frames:  0
Dumped Frames: 0
Link Failure Count: 0
Loss of Sync Count: 49
Loss of Signal: 0
Primitive Seq Protocol Error Count: 0
Invalid Tx Word Count: 470
Invalid CRC Count: 0
```

We expect a certain amount of some of these errors (typically loss of sync and invalid TX words) over time. So a good way to examine them is to compare the counts for the adapters on the LPAR, and to investigate any that have significantly more errors than the rest of the ports.

3.2 How to Tune the Queues

First, one should not indiscriminately increase these values. It's possible to overload the disk subsystem or cause problems with device configuration at boot. It's better to determine the maximum number of submitted IOs to each queue over time, for tuning, if possible. We don't always have this statistic, and in such a case we have to guess how much to increase the queue sizes, to avoid filling the queues up and adding latency to IOs.

When you increase the queue_depths and the number of in-flight IOs that are sent to the disk subsystem, the IO service times are likely to slightly increase, but throughput will also increase. If IO service times start approaching the disk timeout value, then you're risking submitting more IOs than the disk subsystem can handle. If you start seeing IO timeouts and errors in the error log indicating problems completing IOs, then this is the time to look for hardware problems or to make the pipe smaller by reducing queue sizes.

Once IO service times start increasing, we've pushed the bottleneck from the AIX disk adapter driver to somewhere below the hdisk driver in the stack. This could be the adapter driver, the SAN, a VIOS, or the disk subsystem (or components within it) itself. Typically it will be the disk subsystem and often the physical disks within it.

Be cognizant if the IO is high rate asynchronous sequential IO. In such a case, we hope and expect the application to initiate many IO threads so IOs are sent to the storage as soon as possible to drive up thruput. This typically fills the queues making IOs wait, so we don't expect to be able to make the queues big enough to avoid filling, nor do we necessarily need to. E.G., consider if we have a 10 GB table, an application needs to read using a typical max IO size of 256 KB. So the application issues 40,960 256 KB read requests asynchronously, to read the entire table. Unless we have lots of hdisks and very big queue_depths, the queues will be full for awhile (in this case just a few seconds). Then there's still the question of whether the storage can handle that many in-flight IOs, whether we'll run into interconnect bandwidth limits, and whether increasing queue_depth helps performance. Increasing queue_depth to handle such IO only helps to the extent it increases thruput, regardless of wait time in the queue. Increasing queue_depth here increases how many IO requests can be done in parallel, assuming the storage can do more IOs in parallel without significantly degrading IO latency or running into other SAN/storage bottlenecks.

3.2.. 1 Tuning hdisk queue_depth

Two approaches to tuning queue depth are 1) base the queue depths on actual IO requests your application generate or 2) use a test tool and tune the queues based on what the disk subsystem can handle. The ndisk tool (part of the nstress package available on the internet at http://www-941.ibm.com/collaboration/wiki/display/WikiPtype/nstress) can be used to stress the disk subsystem to see what it can handle, and to provide the data to generate IOPS versus. IO service time graphs for the storage. The author's preference is to tune based on your application IO requirements, especially when the disk is shared with other servers.

For tuning, we can categorize the situation into one of 6 cases:

IO Service Time	Good	Poor	Timeout Errors

Queues	Not filling	Filling	Not filling	Filling	Not filling	Filling
Tuning Case and Action:	1. No tuning needed	2. Increase queue sizes	3. Add resources below the hdisk driver	4. Add resources below the hdisk driver. Maybe tune queue_depth.	5. Check the system for bottlenecks below the hdisk driver. Tune the storage, add storage resources, and/or reduce queue sizes	6. Reduce queue sizes and add resources below the hdisk driver

Figure 10 – Tuning Situations

We like to tune the queues, so the queues don't fill up, and still have good IO service times, leaving us in case 1. Limited storage resources often leave us in case 3 or 4, getting the most thruput possible for

Case 2 is very common. The default queue_depth values aren't appropriate for everyone, so increasing the queue sizes is important. For example the default vSCSI hdisk has a queue_depth of 3, limiting its IOPS bandwidth to 3/<avg IO service time>. In choosing how much to increase queue depth, some of the considerations include:

- How many maintenance windows do you get to tune this?
- How much do you need to improve performance?
- What kind of planned growth in IO performance is needed in the future?
- How much more bandwidth does your storage have?
- The average IO service time and the average IO wait time in the queue
- How long are your data intervals, and are there peaks within those intervals are you missing?

For hdisks, there isn't a simple method to know how much to increase the queue sizes. The longer IOs are waiting in the queues, the more we need to increase queue_depth. Sometimes the queue sizes need to be increased significantly, and doubling them in many cases isn't unreasonable.

A reasonable change would be, at a minimum:

- new queue_depth = queue_depth x <avg wait time/avg IO service time>

Or alternatively:

- new queue_depth = queue_depth x (1 + <hdisk blocked IO rate>/<hdisk IOPS rate>)

Usually the considerations listed above call for increasing queue_depth even more. And these variables are related in a non-linear function, while we're using a linear proportional estimation.

Case 3, where we have poor IO service times and we're not filling the queues, indicates a situation in which we need to examine the IO stack below the hdisk driver for bottlenecks, and relieve them by adding resources. And this applies to cases 4-6 as well, where IO service times are poor or worse. This includes looking at the adapter driver, the SAN and the disk subsystem including its various components. If one is using VIO, then we also need to examine the real and virtual adapters, and the VIOS as well for sufficient

resources. The storage administrator will appreciate the systems administrator checking the system for bottlenecks prior to asking them to analyze the storage for bottlenecks.

Case 4, where IO service times are poor and the hdisk driver queues are filling up, we have a bottleneck in the storage, and we've examined the stack below the hdisk driver and didn't find a bottleneck on the system, then increasing queue_depth may or may not improve thruput. It might improve thruput by allowing the disk subsystem to use its elevator algorithms to reorder IOs, but only testing will tell. Tuning the storage or adding storage resources is the sure way to improve performance.

Cases 5 and 6 are situations to avoid, where we're getting IO timeout errors. Here the system is submitting more IOs than the storage can handle. If you have this situation or are close to it (as shown by maximum IO service times in the seconds), check for bottlenecks on the system below the hdisk driver, then work on improving the storage via tuning if that's an option, or by adding storage resources.

All disks and disk subsystem have limits regarding the number of in-flight IOs they can handle, mainly due to memory limitations to hold the IO request and data. When the storage loses IOs, the IO eventually will time out at the host, recovery code might be used to resubmit the IO, but in the meantime transactions waiting on that IO will be stalled. This isn't a desirable situation, as the CPU ends up doing more work to handle IOs than necessary, the application will likely stall waiting on the IO time out (typically 30 or 60 seconds), or eventually an IO failure may be returned to the application which can lead to corrupt data or an application crash. Some applications have their own internal time outs, which are often shorter than IO timeouts. Checking your storage documentation to understand its limits will help you avoid this situation.

Some storage vendors or storage administrators may recommend you do not change your hdisk queue_depth values. In such a case, be sure to plan for enough hdisks to get the IOPS bandwidth your application needs, estimating the IOPS bandwidth per LUN as something less than queue_depth/<avg. IO service time>.

Typically we'll set our queue sizes so that when we're filling the queues, random read disk service times are at worst averaging around 3-5X reasonable IO service times for the disk technology. E.G., with 15 K RPM HDDs in a large disk subsystem with read cache, we expect typical random small block read IO service times in the 5-10 ms range, and we don't want them to average more than 50 ms. And in such a case, you have a significant IO bottleneck. Having a graph showing IOPS vs. IO service time for the storage allocated to the LPAR will help you decide what a reasonable maximum IO service time is for your storage, such as shown in figure 12 (this graph is for a single hdisk, but the shape of the graph is similar for any storage, including entire disk subsystems, or for what's been allocated to an LPAR). In figure 12, we might say a reasonable IO service time is 7.5 ms or less, while we don't want to exceed about 28 ms. What reasonable IO service times are, for your storage, is discussed in section 4.5 *What Are Good, Reasonable and Poor IO Service Times?*

3.2.. 2 Tuning fcs Device Queue Attributes

Cases 2 thru 6 figure 10 all show poor IO service times, indicating that there is a bottleneck in the IO stack below the hdisk driver. Just below the hdisk driver is the adapter driver, so we should check it to make sure it isn't the bottleneck, before we start looking into the SAN or disk subsystem.

The adapter driver statistics capture the peak workload. For num_cmd_elems, the following commands tell us exactly how to set num_cmd_elems:

```
# fcstat –D <fcs#>
# pcmpath query adaptstats
# datapath query adaptstats
```

For the AIX PCM:

- new num_cmd_elems = <high water mark of active commands> + <high water mark of pending commands>

E.G. with this output:

```
# fcstat –D fcs1
...
     FC SCSI Adapter Driver Queue Statistics
       Number of active commands:   0
       High water mark of active commands:   180
       Number of pending commands:   0
       High water mark of pending commands:   91
       Number of commands in the Adapter Driver Held off queue:  0
       High water mark of number of commands in the Adapter Driver Held off queue:  0
...
```

We'd set num_cmd_elems to 180+91= 271.

For SDDPCM or SDD:

- new num_cmd_elems = maximum I/O value displayed with the respective above command

E.G., with this output:

```
      Adapter #: 0
      =============
                Total Read Total Write Active Read Active Write Maximum
      I/O:        439690333 24726251          7       0            258
```

We'd set num_cmd_elems=258.

If you're at a level of AIX/VIOS lacking the fcstat –D command data, we can use fcstat:

```
# fcstat fcs0
...
FC SCSI Adapter Driver Information
No DMA Resource Count: 0
No Adapter Elements Count: 0
No Command Resource Count: 0
...
```

This fcs0 data shows an example of an adapter that has sufficient values for num_cmd_elems and max_xfer_size. Non-zero values indicate a situation in which IOs waited in the adapter driver queue due to lack of resources. Increase num_cmd_elems for non-zero values of No Command Resource Count when the rate is significant. Increase the DMA pool size for non zero values for No DMA Resource when the rate is significant, if possible. Note that changing max_xfer_size isn't always possible, and can lead to boot failure. Please see section 1.3 *Changing Device Attribute Values* that discusses this if you haven't already.

Here, similar to increasing queue_depth, we'd increase num_cmd_elems to something more than:

- new num_cmd_elems = num_cmd_elems x <blocked IO rate>/<IOPS rate>

No adjustment of the DMA memory size is possible on virtual FC adapters, just on the real adapter.

3.3. After Your Initial Tuning

After changing the values, run your application and look at the statistics for peak IO periods and tune again, if needed. While entirely eliminating queue wait time is the goal, the returns from investing time in this tuning diminish as fewer IOs wait in the queues, and as they wait less time as well.

3.4. Tuning Order

Regarding the order to tune the stack, one generally tunes queue_depth, num_cmd_elems, and max_xfer_size together in one maintenance window. Tuning the lower layers of the stack first (e.g., tuning the adapter queue sizes before the hdisk queue sizes) can resolve queue waits higher in the stack. On the other hand, tuning higher layers in the stack typically increase the number of in-flight IOs sent to the lower layers which can result in more blocking there. So tuning these is an iterative process.

3.5. What Are Good, Reasonable and Poor IO Service Times?

What is good, reasonable, or poor is a factor of the storage technology, the storage cache sizes, and the IO workload for the disk subsystem. Assuming no IOs are queued to a HDD, a typical read will take somewhere from 0 to 15 ms, or so, depending on how far the actuator has to travel (seek time), how long it takes the disk to rotate to the right sector (rotation time based on spindle speed), and how long it takes to read the data (transfer time). Then the data must move from the storage to the host. Typically the time is dominated by seek time + rotation time, though for large IOs transfer time also can be significant. This includes times to transfer the IO to/from the disk, as well as sending it across various interfaces (storage backend, storage to host interface, etc. including FC, SAS, iSCSI and FCoE). Sometimes the data will be in disk subsystem read cache, in which case the IO service time is around 1 ms or better. Typically for large disk subsystems that aren't overloaded with 15K RPM disks, read IO service times will average

around 5-10 ms. When small random reads start averaging greater than 15 ms with 15K RPM disks, this indicates the storage utilization is high.

Writes typically go to write cache (assuming it exists) with average IO service times typically less than about 2.5 ms, but there are exceptions. If the storage is synchronously mirroring the data to a remote site, writes can take much longer as we have to add the inter-site round trip latency. And if the IO is large (say 64 KB or larger) then the transfer time becomes more significant and the average time is slightly worse. If there's no cache, then writes take about the same as reads. Some disk subsystems don't cache large write IOs and send them directly to the storage.

Here's a table showing expected IO service times and IOPS for lightly loaded disk technologies:

IOPS for different disks

Disk Drive	Speed	Rotational Latency	Avg. Seek Time	IOps
15k 3.5" FC	15000 rpm	2 ms	3.5 ms	182
10k 3.5" FC	10000 rpm	3 ms	4.5 ms	133
15k 2.5" SAS	15000 rpm	2 ms	3.1 ms	196
10k 2.5" SAS	10000 rpm	3 ms	4.2 ms	139
7.2k - SATA2	7200 rpm	4.2 ms	9 ms	76

Figure 11 - IOPS for different disk technologies

These times are simply calculated as IOPS = 1/ (rotational latency + avg. seek time)

In reality, HDDs can do more or less IOPS at higher/lower IO service times respectively. And one can characterize a HDD in an IOPS vs. IO service time chart as follows (in this case for a 7200 RPM SATA disk doing 4 KB IOs):

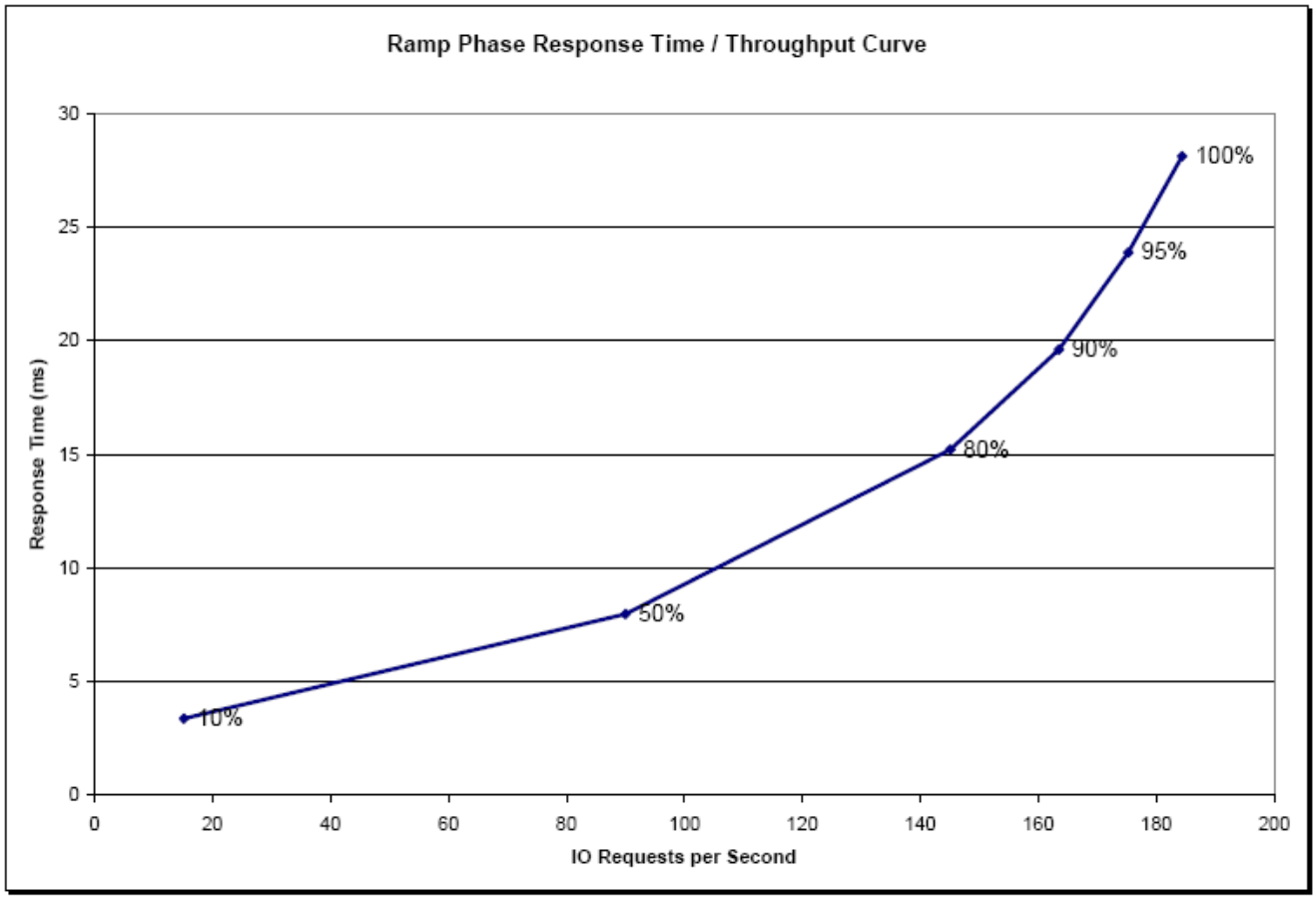

Figure 12 - Sample IOPS vs. IO service time graph

So while this 7200 RPM SATA disk can achieve over 180 IOPS, we may not want to drive it at that rate, and prefer a lower IOPS rate and a better IO service time. The percentages on the graphed line are the HDD utilization. Thus, what's reasonable depends on the workload and the customer's performance requirements. For a HDD like this, an IO service time of 20 ms is reasonable, unless you've planned to do fewer IOPS and get better IO service times to meet your application performance requirements. Similar charts exist for disk subsystems, and one can make such a chart for storage allocated to an LPAR.

If the IO is large block sequential, then besides the increased transfer time, we expect IOs to queue at the physical disk, and IO service times to be much longer on average. E.G., if an application submits 50 IOs (say 50 64 KB IOs reading a file sequentially) then the first few IOs will have reasonably good IO service times, while the last IO will have had to wait for the other 49 to finish first, and will have a very long IO service time. We hope to queue up a bunch of IOs for sequential IO threads, and so we hope they will be waiting in the queue ready to go. Thus, we don't get concerned with long IO service times for sequential IO, and instead focus on thruput.

IOs to SSDs/flash are typically less than 1 ms and for SSDs/flash in disk subsystems, typically less than 2 ms, and occasionally higher. IO latencies for flash have been improving, as shown in this chart:

Power eMLC SSD Performance by Generation

2.5-inch (SFF)	IO OPERATIONS PER SECOND (IOPS)			Throughput (MB/s)		Latency - Response Time (ms)
SSD	Random Read	Random Write	Random Mixed Read/Write	Read	Write	Random Mixed Read/Write
Gen 1 177GB	15 k	4 k	11 k (70%-30%)	170 MB	64 MB (24-123)	0.31 ms
Gen 2 387GB	39 k	22 k	24 k (70%-30%)	340 MB	375 MB	0.25 ms
Gen 3 387GB 775GB	80 k	49 k	58 k (60%-40%)	340 MB	450 MB	0.15 ms
For grins … 15k rpm HDD	0.12 - 0.4 k	0.12 - 0.4 k	0.12 - 0.4 k	~175 MB	~175 MB	8.3 – 2.5 ms

Note these are single drive specific measurements reflecting sustained drive workloads (not burst). The values assume 528 byte sectors running RAID-0 with no protection. Hypothetically if measured with unsupported 512 byte sectors, values would be higher. The values are highly workload dependent. Factors such as read/write mix, random/non-random data, drive cache hits/misses, data compressibility in the drive controller, large/small block, type of RAID or mirroring protection, etc will change these values. The choice of which SAS controller/adapter is running the drive can also impact these values.. These values produced by a server with plenty of processor, memory and controller resources to push this much I/O into the SSD. Most client system applications don't push SSD nearly this hard. Latency measurements using OLTP1 60/40 random 4k transfers.

Figure 13 - eMLC SSD performance

Where the flash is implemented also matters. The numbers in the chart are for SAS attached SSDs. If the flash is in a disk subsystem, add about 0.5 ms to the latency for the IO to go thru the disk subsystem back end interfaces and also thru the SAN for what is expected. If using VIO, then add 0.05 ms (50 micro seconds) to latency to go thru the VIOS. The FC attached IBM FlashSystem can achieve 0.3 ms for IOs thanks to its use of a data path using hardware rather than software.

Understanding what reasonable IO service times are for your solution is important in tuning the queues, so you know if you have a bottleneck below the hdisk driver, or not.

3.5..1 Storage Cache Management Algorithm Effects on IO Service Times

Some disk subsystems use cache management algorithms, which sometimes take away write cache for a LUN. This occurs where the IOPS write rate to a LUN is very low, and where there is a high read rate from the LUN, or heavy use of the cache elsewhere. In such cases, the disk subsystem may take all or nearly all the write cache for a LUN and use it elsewhere. The result when an application sends a write in this situation is the disk subsystem cache management algorithms have to take the time to free up and allocate some cache for the write, and place the write data in the cache so the disk subsystem can then send

back an acknowledgement that the write is complete. This adds significant latency to these IOs, and your iostat data may look like:

```
# iostat -lD
Disks:                    xfers                                read                                write
------------- ------------------------------- ---------------------------------- ----------------------------------
               %tm    bps   tps  bread  bwrtn   rps    avg    min    max time fail   wps    avg    min    max time fail
               act                                      serv   serv   serv outs              serv   serv   serv outs
hdisk0         0.3  26.7K   3.1  19.3K   7.5K   1.4    1.7    0.4   19.8      0    0   1.6    0.8    0.6    6.9      0    0
hdisk1         0.1 508.6    0.1 373.0  135.6    0.1    8.1    0.5   24.7      0    0   0.0    0.8    0.6    1.0      0    0
hdisk2         0.0  67.8    0.0   0.0   67.8    0.0    0.0    0.0    0.0      0    0   0.0    0.8    0.7    1.0      0    0
hdisk3         1.1  37.3K   4.4  25.1K  12.2K   2.0    0.8    0.3   10.4      0    0   2.4    4.4    0.6  638.4      0    0
hdisk4        80.1  33.6M 592.5  33.6M  38.2K 589.4    2.4    0.3  853.6      0    0   3.1    6.5    0.5  750.3      0    0
hdisk5        53.2  16.9M 304.2  16.9M  21.5K 302.2    3.0    0.3    1.0S     0    0   2.0   16.4    0.7  749.3      0    0
hdisk6         1.1  21.7K   4.2   1.9K  19.8K   0.1    0.6    0.5    0.8      0    0   4.0    2.7    0.6  495.6      0    0
```

Note the high write IO service times in red and the high read IOPS in blue. Overall, the LPAR is doing 903.8 IOPS, but latency is only affected for 7.4 IOPS; thus, less than 1% of the IOs are affected.

The disk subsystem might offer some tuning to prevent this, but if not, there isn't any easy method to deal with it. Consider that since the impact is to a small fraction of the IOs, that overall it isn't likely to have a significant performance impact. Thus, one needs to be aware of this, prior to deciding the writes are too slow.

3.6. Tuning Queue Sizes in VIO Environments

IO thru VIO is done either with vSCSI or NPIV, using a virtual SCSI or a virtual Fibre Channel (vFC) adapter as shown in figures 14 and 15 respectively.

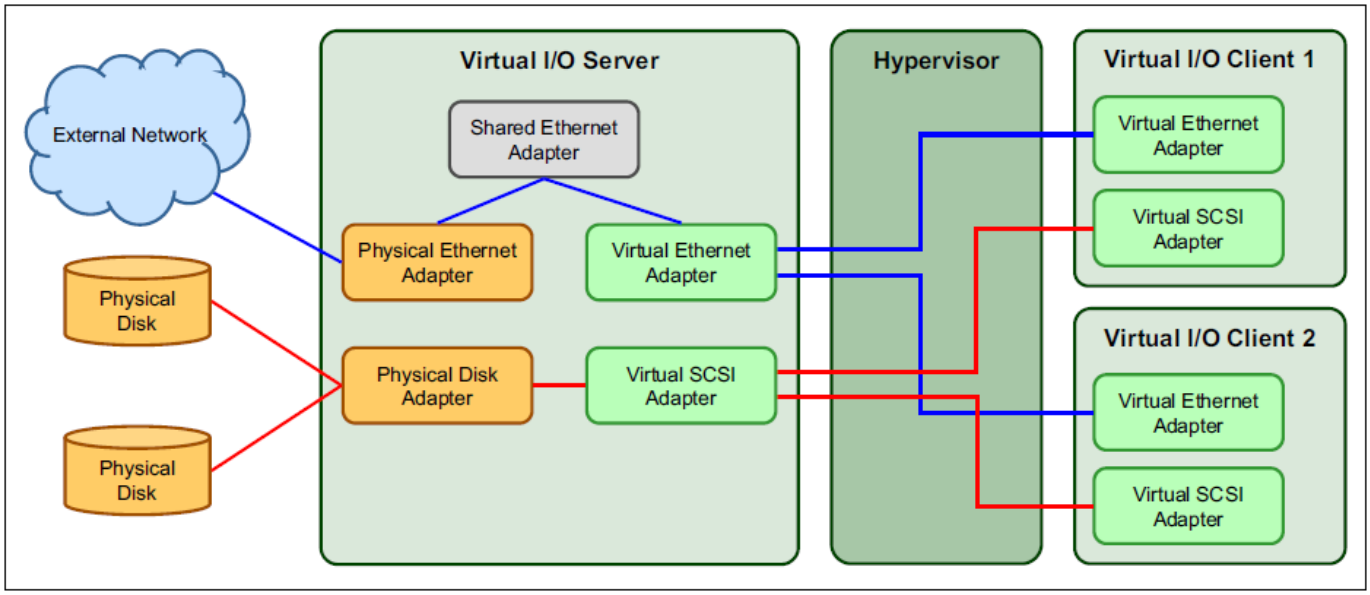

Figure 14 – vSCSI Architecture

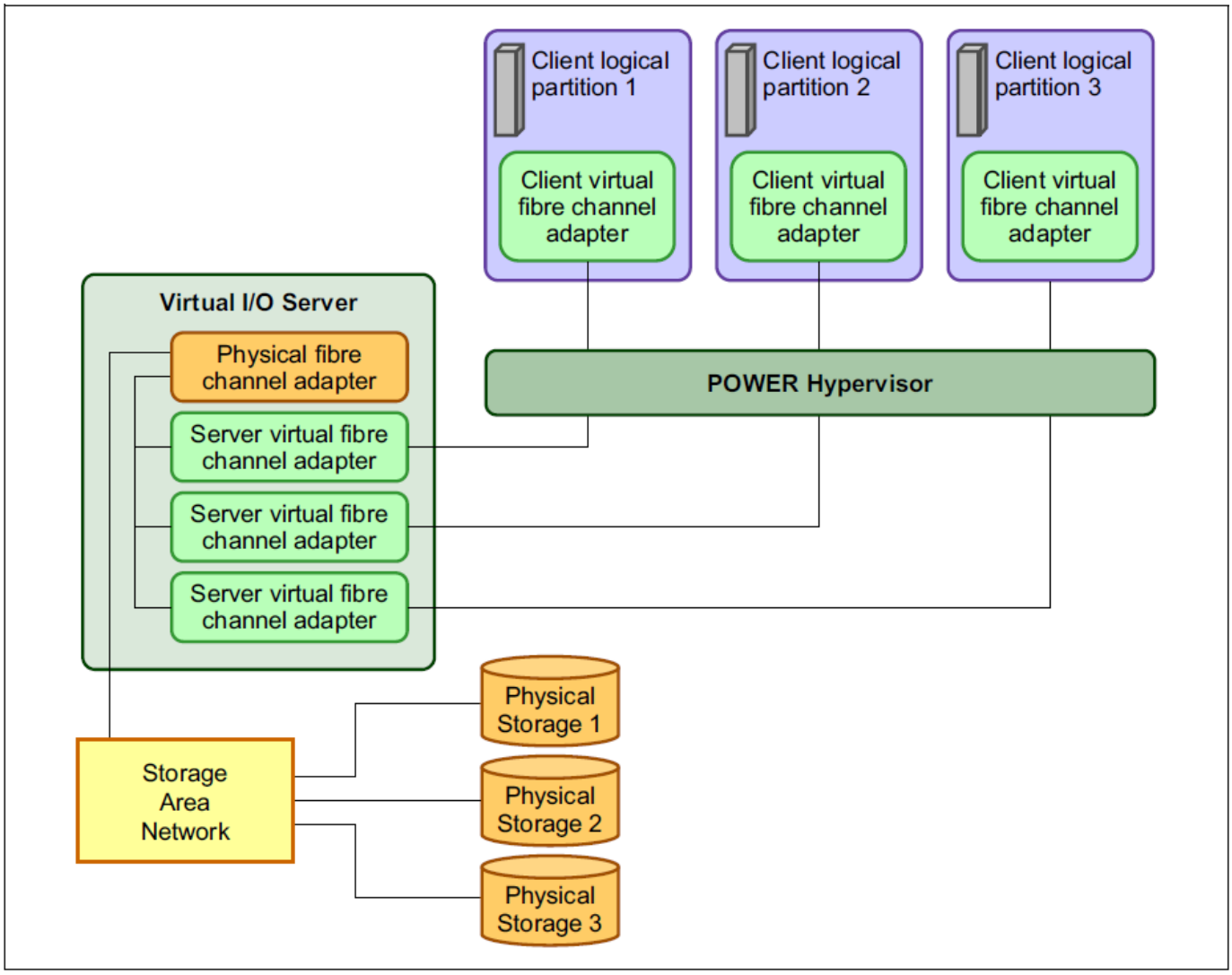

Figure 145 – NPIV Architecture

These diagrams show the logical connections of the virtual and real hardware, but they don't show the IO stack, so let's look at that:

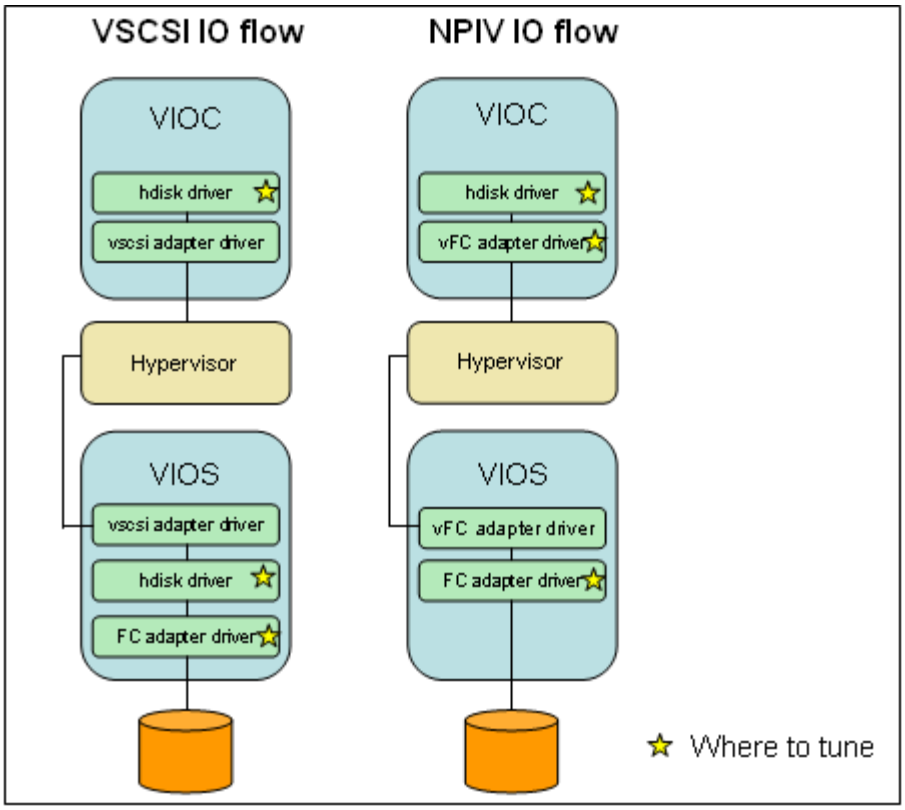

Figure 156 – VIO IO Stack Tuning

Note that we do not go thru the hdisk driver in the VIOS with NPIV. Thus, NPIV IO statistics are not captured via iostat, sar or other commands at the VIOS that access the hdisk driver statistics! We can still capture IO statistics via fcstat and nmon that access the adapter driver statistics.

With vSCSI, we tune hdisk queue_depth at both the VIOS and the VIOC. And we also tune the real FC adapters connected to the storage. The adapter also has a fixed queue, so to avoid queuing there, we limit the number of hdisks using their adapters based on their queue sizes.

With NPIV, we tune the hdisk queue depths at the VIOC, the vFC adapters at the VIOCs, and the real adapters at the VIOSs. One thing we can't change for the vFC is the DMA memory size, which is fixed regardless of the max_xfer_size attribute value. It's important to know the vFC adapters have an effective limit of 256 in-flight IOs; thus, increasing num_cmd_elems beyond 256 won't have any affect.

3.6.. 1 Avoiding VIOS Outages

To change device attributes means they cannot be in use. Disks mapped thru vSCSI or vFCs mapped thru real FC adapters are in use. And often there are several vFCs using a single real FC. Thus, changing the real FC adapters supporting NPIV, without a VIOS outage, is difficult.

A simple approach is to use Live Partition Migration (LPM) to move the LPARs, using the VIOS, to another system, make the change to the real adapter attributes, then use LPM to move the LPARs back.

Another approach is essentially to replace the adapter, see http://pic.dhe.ibm.com/infocenter/powersys/v3r1m5/index.jsp?topic=%2Fiphak%2Fiphak-vios-rr.htm, or to add a new adapter, new vFCs, changes at the storage and SAN, then remove the old vFCs and adapter. These approaches are time consuming.

In dual VIOS environments, we can make changes to the ODM for device attributes (via the –P flag of the chdev command), and reboot for a short outage of one VIOS.

NPIV hdisk attributes are changed at the VIOC, so one only needs to stop using them to change the attributes.

VSCSI hdisks can be similarly changed at the VIOC. Changing them at the VIOS requires stopping their use at the VIOC and unmapping them from the client at the VIOS.

For customers who can take outages on the VIOS, it's often desirable for each hdisk to have the same hdisk queue size at both the VIOS and VIOC; thus, when changing it at the VIOC one also changes it at the VIOS.

3.6.. 2 Two Strategies to Limit In-flight IOs to the Storage

As previously mentioned, it's important to limit the number of in-flight IOs to the storage. But there are different ways to do this. IBM has previously recommended that one set queue_depth for vSCSI hdisks to match at the VIOC and VIOS. And we usually recommend tuning both the hdisk and adapter drivers to just have enough resources to handle the peak workload. However, we can also take the approach whereby we limit the IOs at the hdisk drivers, the adapter drivers, or at the VIOCs.

As stated in section 1.2:

The maximum in-flight IOs a system will submit to SAN storage is the smallest of the following

- The sum of the hdisk queue_depths
- The sum of the adapter num_cmd_elems
- The maximum number of in-flight IOs submitted by your application(s)

Thus, we can achieve our objective by limiting the hdisk queue_depths so that their sum is less than the maximum in-flight IOs the storage will handle. Similarly, we can limit the sum of the num_cmd_elems so that it is less than the storage limits. And we can limit the in-flight IOs by adjusting hdisk queue_depth at either the VIOC or the VIOS, and similarly we can limit the IOs by adjusting num_cmd_elems at the VIOS or at the VIOCs when using NPIV.

This gives us two strategies for limiting in-flight IOs:

1. Set all the queue sizes to the minimum size that can handle the peak workload. Set vSCSI hdisk queue_depth to match at the VIOS and the VIOC. Tune adapter num_cmd_elems and DMA memory size at the VIOS and num_cmd_elems at the VIOC, to handle the peak workloads.
2. Set the queue sizes at the VIOS to their maximum values, and limit them at the VIOCs.

Strategy 1 has the virtue of minimizing the use of resources (mostly memory to handle increased queue sizes). Strategy 2 has the virtue of minimizing VIOS outages for tuning and reducing administrator work, but uses extra memory. Both approaches insure that we don't have hdisk driver queue bottlenecks at the VIOS.

Customers may take the approach that best fits their needs.

3.6.. 3 Tuning vSCSI Queues

When using vSCSI, one configures virtual SCSI adapters for the VIOC to access storage configured on the VIOS. For each virtual SCSI adapter, there will be a vscsi device in a VIOC, and a matching vhost device in the VIOS. We don't have knobs to change queue sizes for the vscsi adapters, instead we choose how many we create and which hdisks use which adapters.

These virtual adapters have a fixed queue depth. There are 512 command elements of which 2 are used by the adapter, 3 are reserved for each vSCSI LUN for error recovery and the rest are used for IO requests. Thus, with the default queue_depth of 3 for vSCSI LUNs, that allows for up to 85 LUNs to use an adapter without blocking there: (512 - 2) / (3 + 3) = 85 rounding down. So if we need higher queue depths for the devices, then the number of LUNs per adapter is reduced. E.G., if we want to use a queue_depth of 25, that allows 510/28= 18 LUNs. We can configure multiple vSCSI adapters to handle many LUNs with high queue depths. Each vSCSI adapter uses additional memory and resources on the hypervisor. For a formula, the maximum number of LUNs per virtual SCSI adapter (vhost on the VIOS or vscsi on the VIOC) is =INT(510/(Q+3)) where Q is the queue_depth of all the LUNs (assuming they are all the same). Here's a table for guidance:

vSCSI hdisk queue depth	Max hdisks per vscsi adapter*
3 - default	85
10	39
24	18
32	14
64	7
100	4
128	3
252	2
256	1

Figure 167 – vSCSI LUN limits

* Maximum to ensure no blocking of IOs due to lack of vscsi queue slots

When using flash backed LUNs, where you expect relatively high IOPS, we currently recommend using a single vSCSI adapter per LUN. This is due to other limitations with the vSCSI adapter and because flash IOs are so fast.

For LV vSCSI hdisks, where multiple VIOC hdisks are created from a single VIOS hdisk, then one may take a dedicated resource, shared resource or an in between approach to the VIOS hdisk queue slots. See the section 4.5 *Theoretical Thoughts on Shared vs. Dedicated Resources*.

3.6.. 4 Tuning NPIV Queues

When using NPIV, we have vFC and real FC adapter ports, and often have multiple vFCs tied to a single real FC adapter port.

If you increase num_cmd_elems on the virtual FC (vFC) adapter port, then you should make sure the setting on the real FC adapter port is equal or greater. With multiple vFCs per real FC, the real FC port's resources become a shared resource for the vFCs. Keep in mind that num_cmd_elems is effectively limited to 256 or smaller, so there isn't any benefit to increasing num_cmd_elems to greater than 256 for vFCs.

You can use the fcstat command for both the virtual adapter port as well as the real adapter port for tuning, and determining whether or increasing num_cmd_elems or max_xfer_size might increase performance.

So for heavy IO and especially for large IOs (such as for backups) it's recommended to set max_xfer_size such that we give the maximum DMA memory to the fcs device, when the system configuration will allow it.

3.6.. 5 Tuning Shared Storage Pool Queues

Here's a diagram of a Shared Storage Pool (SSP) cluster:

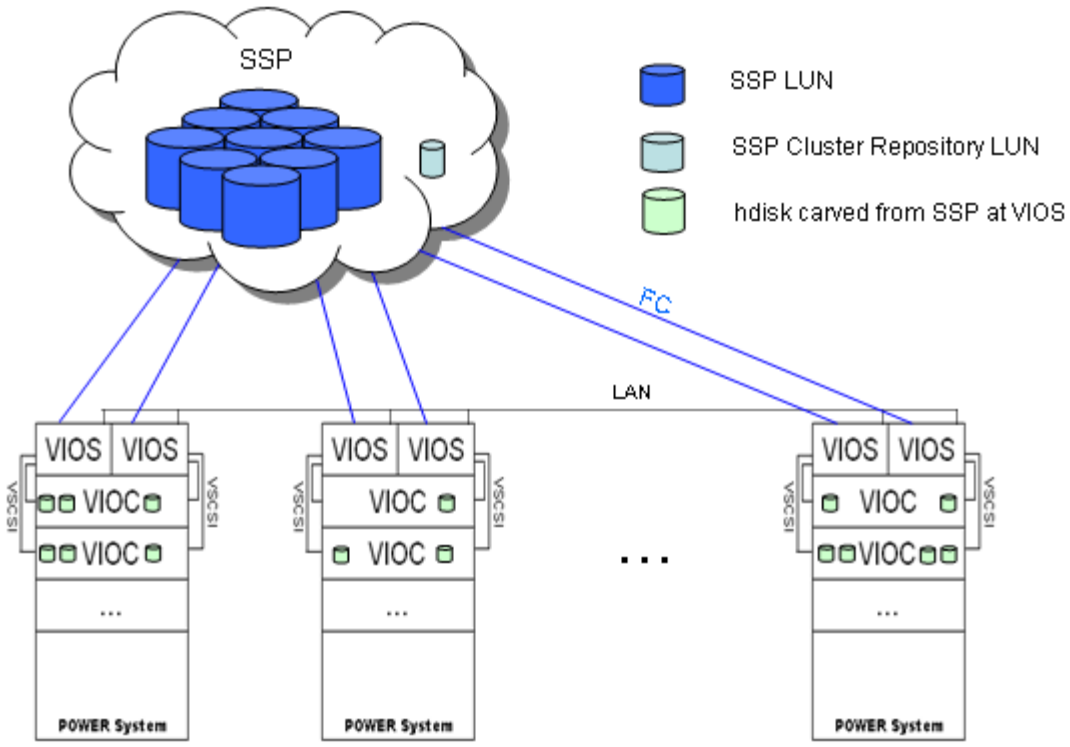

Figure 178 – Shared Storage Pool Cluster

The SSP and SSP cluster repository LUNs are allocated from a RAID protected SAN disk subsystem(s) to all the VIOSs in the SSP cluster. The SSP LUNs are typically very large, while the cluster repository LUN is about 10 GB. These LUNs are configured as hdisks at the VIOS. The VIOS administrator creates virtual hdisks from the SSP and maps them to VIOCs for new or existing LPARs. This removes the need for the storage/SAN administrators to get involved, and speeds the process. The virtual disks the VIOS administrator creates are spread out across the LUNs in the SSP to balance the IO workload across them. SSPs also offer thin provisioning and snapshot capability as well.

From a tuning perspective, we tune:

- FC adapter queues at the VIOS
- queue_depth on the SSP hdisks at the VIOS
- queue_depth of the virtual disks at the VIOCs

We only need to stop using the hdisks at the VIOC to change their attributes. To change attributes at the VIOS will typically require a VIOS reboot.

One should also give some thought to making sure the servers don't overrun the storage with IO requests. One approach to this is for the storage administrator to monitor the storage resource use, and alert the server administrators about it. Then the server administrators can investigate and reduce queue sizes as necessary, typically while the storage team will be getting new resources to handle the demand.

Designing the queue sizes gets into sharing vs. dedicating storage resources, from the maximum in-flight queue slots within the disk subsystem(s) to the virtual disk queue slots at the VIOCs.

From a shared resource perspective:

- The virtual disks at the VIOCs share the SSP hdisk queues at the VIOS
- The SSP hdisks share the FC adapter queues at the VIOS
- The VIOSs share the disk subsystem queues for its limit of in-flight IOs

We can affect how these resources are dedicated or shared by where and how much one limits the queue sizes.

3.6. Theoretical Thoughts On Shared Vs. Dedicated Resources

Queue sizes limit the maximum number of in-flight requests for storage. So reducing the queue size of say an hdisk, gives the other hdisks access to more queue slots further down in the stack, such as the adapter queue slots, all the way to the disk subsystem(s) queue slots. Thus by reducing queue sizes for hdisks, LPARs, VIOSs, adapters and systems, we provide more of the storage resources to the other hdisks, LPARs, VIOSs, adapters and systems respectively.

Typically we have many hdisk drivers sharing multiple adapters and adapter drivers, thus, the FC queue slots are a shared resource for the hdisk drivers:

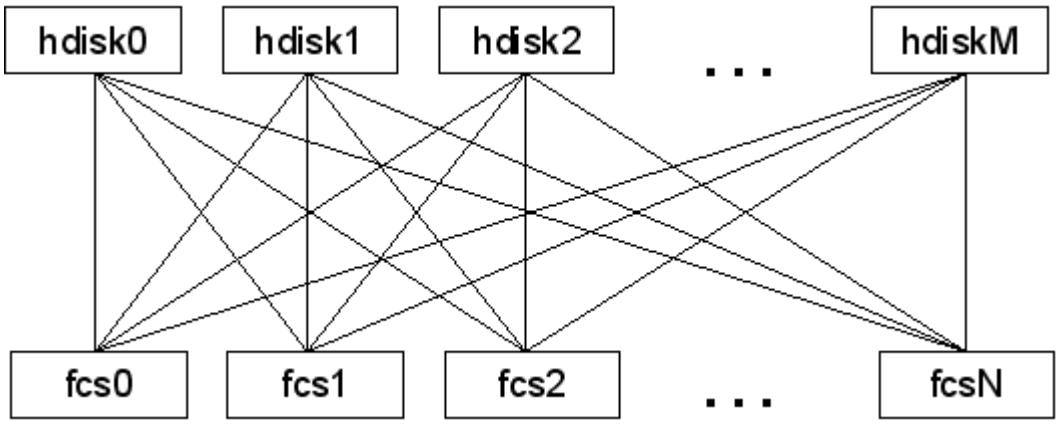

Figure 189 – Adapter and port hdisk queue connections

Thus, it's possible to ensure that we never fill the adapter port queues, by making SUM(hdisk0 queue_depth, hdisk1 queue_depth, ... hdiskM queue_depth) <= SUM (fcs0 num_cmd_elems, fcs1 num_cmd_elems, ... fcsN num_cmd_elems). This assumes that IOs are evenly spread across the adapters. And most multi-path code does balance IOs across the adapters (or at least can).

Though environments often have many more hdisks than FC ports, and ensuring we won't fill the adapter drivers can lead to small values for queue_depth, and full queues on the hdisk drivers.

So there is the dedicated resource approach, the shared resource approach, and in between dedicated and shared. Taking this simple example where Q represents the queue size for the device driver:

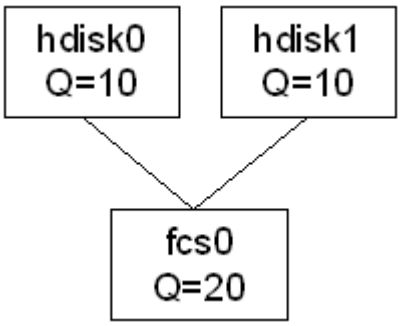

Figure 190 – Dedicated adapter port queue slot resource

This would be considered a dedicated resource approach, where 10 of the adapter driver queue slots are dedicated to each hdisk driver. Here we know we'll never submit an IO to a full queue on the adapter driver.

Alternatively:

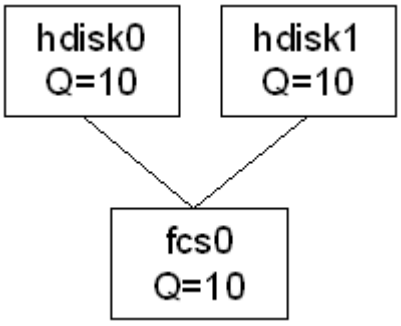

Figure 201 – Shared adapter port queue slot resource

This would be considered a shared resource approach where the 10 adapter queue slots could be filled up from a single hdisk driver.

And here's an example of something in between:

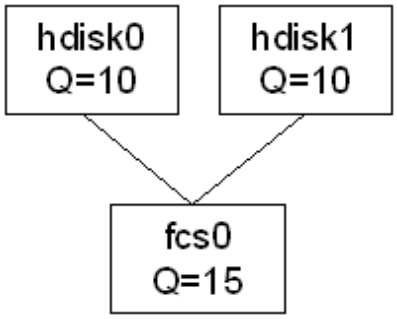

Figure 22 – Adapter port queue slot resource – partly dedicated, partly shared

Here, there will always be at least 5 queue slots available in the adapter driver for either hdisk driver.

There are pros and cons to each approach. The benefit of the dedicated resource approach is that the resources allocated will always be available but typically there will be fewer resources available to each user of the resource (here the resource we're considering is the adapter queue slots, and the users of the resource are the hdisk drivers sending IO requests from the hdisk queues). The benefit of the shared resource approach is that we'll have more resources for an individual user of the resource when it needs it and it will be able to get greater thruput than in the dedicated resource approach. The author generally prefers a shared resource approach, as it generally provides the best overall thruput and price performance.

Note that this situation of shared resources occurs in several possible ways beyond hdisk drivers using adapter drivers. It is also involved when:

- Several LV vSCSI hdisks for a single hdisk on a VIOS
- Several vFC adapters using a single real FC adapter
- Several LPARs using the same disk subsystem
- Several systems using SSPs
- vSCSI hdisks using a vscsi adapter

So similar approaches can be used for sharing or dedicated those resources as well.

We've taken two approaches to tuning queues here. One approach uses IO statistics to determine how big the queues should be, while the other approach looks at the potential number of IOs that might arrive to a queue. The later approach will be more conservative in avoiding queue wait, and will also have larger average number of requests in the service queue.

In practice one should generally size the servers and storage based on the application resource needs, setup the environment, test its performance, and set the queues appropriately then, taking into account planned growth.

Mathematically, the IO queues here are closest to an M/D/c queue. M represents IO arrivals and is assumed to have a Poisson distribution. D refers the fact that IOs arriving to the queue will need some time to service them, and c referring to the number of queue slots whereby up to c IOs can be serviced in paral-

lel. An M/D/*c* queue model is defined as a stochastic process whose state space is the set {0,1,2,3,...} where the value corresponds to the number of customers in the system, including any currently in service. In queuing theory, an M/D/c queue represents the queue length in a system having *c* servers, where arrivals are determined by a Poisson process and job service times are fixed (deterministic). See http://en.wikipedia.org/wiki/M/D/k_queue.

4. Estimating Application Performance Improvement

Projecting application performance improvement based on disk latency improvement, from the application's point of view, has challenges. Some of the challenges include:

- Disk IOs are often done in parallel
- Many IOs are done asynchronously and don't block the application, while the application is blocked for many other IOs
- Writes tend to block applications more than reads
- IO latency improvements for reads and writes differ

Nevertheless, by making some assumptions, we can create an estimate, or a range of potential improvement, using Amdahl's law.

Assuming we can improve IO latency L% and the disk part of the job is D%, then the potential performance improvement (for a batch job, or for application response) is DxL. E.G., assuming IO service time averages 7 ms, wait time in the queue averages 3 ms, and that we can eliminate time in the queue, then L= 3/(7+3) = 30%; thus improving latency 30% from the application's point of view. Assuming the disk work is 50% of the job time, then we can run the job in 50% x 30% = 15% less time.

Without knowing the portion of the job that is disk related, we don't know the actual improvement, but it puts the potential improvement into the range of 0-L, and a couple of data points at different latencies (as seen by the application) can provide an estimate of D.

Performance improvement to a batch job is reflected in the run time. For OLTP applications, reductions in IO latency are reflected in application response time to a query/update, or to maximum application thruput. Since the number of in-flight transactions is often limited by memory, the ability to do transactions faster means we essentially increase the max TPS for an LPAR by reducing IO latency (or alternatively we could reduce system resources and keep the same TPS with faster IOs).

Another metric we can calculate is how much IO time we can save by eliminating the queue wait time:

IOPS x <avg time in the queue per IO> = IO latency savings per second

Just be aware that this isn't time savings for the application. E.G., if we're waiting in the queue on average 2 ms for each IO, and we're doing 10,000 IOPS, the savings in IO latency would be 20 seconds per second. However we can't save more than 1 second every second, and because the IOs are done in parallel; thus, the real application savings is something less than one second every second. But this metric is an indicator whether tuning will help performance significantly and whether the tuning effort is worth the time.

Appendix: Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this white paper.

- “AIX 5.3 Performance Management Guide”, IBM
- “AIX 6.1 Performance Management Guide”, IBM
- “AIX 7.1 Performance Management Guide”, IBM
- "PowerVM Virtualization on IBM System p: Introduction and Configuration", IBM Redbooks
    - http://www.redbooks.ibm.com/redpieces/abstracts/sg247940.html
- "PowerVM Virtualization on IBM System p: Managing and Monitoring", IBM Redbooks
    - http://www.redbooks.ibm.com/redpieces/abstracts/sg247590.html

- Fibre-channel I/O Performance Tuning on AIX using fcstat; A How-to and Usage Guide
  http://w3-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/TD106122
- IBM System p and AIX Information Center
  http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp (AIX 5.3)
  http://publib.boulder.ibm.com/infocenter/systems/scope/aix/index.jsp (AIX 6.1)
  http://publib.boulder.ibm.com/infocenter/aix/v7r1/index.jsp (AIX 7.1)