Citrix XenServer ® 6.1.0 Storage Performance Guide

Published Thursday, 28 February 2013
1.0 Edition

Citrix XenServer ® 6.1.0 Storage Performance Guide

Version: 6.1.0

Citrix, Inc.
851 West Cypress Creek Road
Fort Lauderdale, FL 33309
United States of America

Disclaimers

Trademarks

Citrix ®
XenServer ®
XenCenter ®

Table of Contents

Overview

XenServer is an enterprise-ready, cloud-proven virtualization platform that contains all the capabilities required to create and manage a virtual infrastructure. Part of this infrastructure enables the mapping of storage targets (e.g. local disks, network storage) to virtual machines. Depending on virtual machine profile, storage type and concurrent workload, the performance of such subsystems can be affected differently.

Intended Audience

This document is aimed at XenServer administrators who would like to learn about how to investigate, monitor and fine tune their deployments in order to obtain the maximum storage performance that XenServer has to offer. It is based on XenServer 6.1.0, but much of it is applicable to older and possibly newer versions. Some general knowledge of the Linux operating system and XenServer itself is assumed.

This is the first edition of such a document and does not cover every storage-related aspect of XenServer. For example, specific advice about the usage of SSDs, how different RAID types interact with the system or technical details of StorageLink will not be addressed. Given customer and community feedback, future editions will be enhanced and directed appropriately.

About the author

Felipe Franciosi is a software performance engineer with the XenServer Performance Team at Citrix. His work focuses on researching and evaluating technologies for improving XenServer performance, mainly with regards to storage. He has a PhD in performance evaluation of virtualized storage systems from Imperial College London.

Acknowledgements

The author would like to thank Jonathan Davies, Marcus Granado, Andrew Bennieston, Stephen Turner, Lawrence Simpson, Sarah Vallieres, Usha Mandya, David Vrabel (along with the XenServer RING0 Engineers) and Keith Petley (along with the XenServer Storage Engineers) for their help, contribution and motivation towards the writing of this document.

Background

Before discussing the details of investigating, monitoring and tuning the XenServer storage subsystem, it is important to have a thorough understanding of the subsystem itself. For that, we recommend the reading of Chapter 5 (Storage) of the Citrix XenServer ® 6.1.0 Administrator's Guide. This chapter focuses on reviewing the terminology and concepts of the storage subsystem, with emphasis on what is relevant for this guide. Its reading is recommended even for those who are familiar with the subject.

Storage Subsystem Overview

Several different storage media types can be connected to a XenServer host. From local disks to network-attached storage, to name but a few examples, a XenServer host abstracts each connected storage infrastructure into a *Storage Repository* entity.

Each Storage Repository (SR) is connected to a host through a Physical Block Device (PBD). If multiple hosts exist in a pool, certain types of SRs can be shared amongst these hosts, causing one PBD to exist for each mapping <SR, Host>. This guide will focus on the cases where SRs are not shared, but the information here available can be easily adapted to cover both scenarios.

From an SR, an administrator can create *Virtual Disk Images* which can then be attached to a Virtual Machine (VM) through a Virtual Block Device (VBD). The Virtual Disk Image (VDI) representation on the SR will depend on the SR type. For example, it could be a file in an NFS SR or a Logical Volume on an LVM SR.

The illustration below represents a XenServer Host with three attached SRs (a Local Disk, an NFS Server and an iSCSI Target). The SRs have a series of VDIs created on them, and are attached to two different paravirtualized VMs through corresponding VBDs. The virtual devices appear to the VMs as local block devices (named xvda, xvdb, etc).

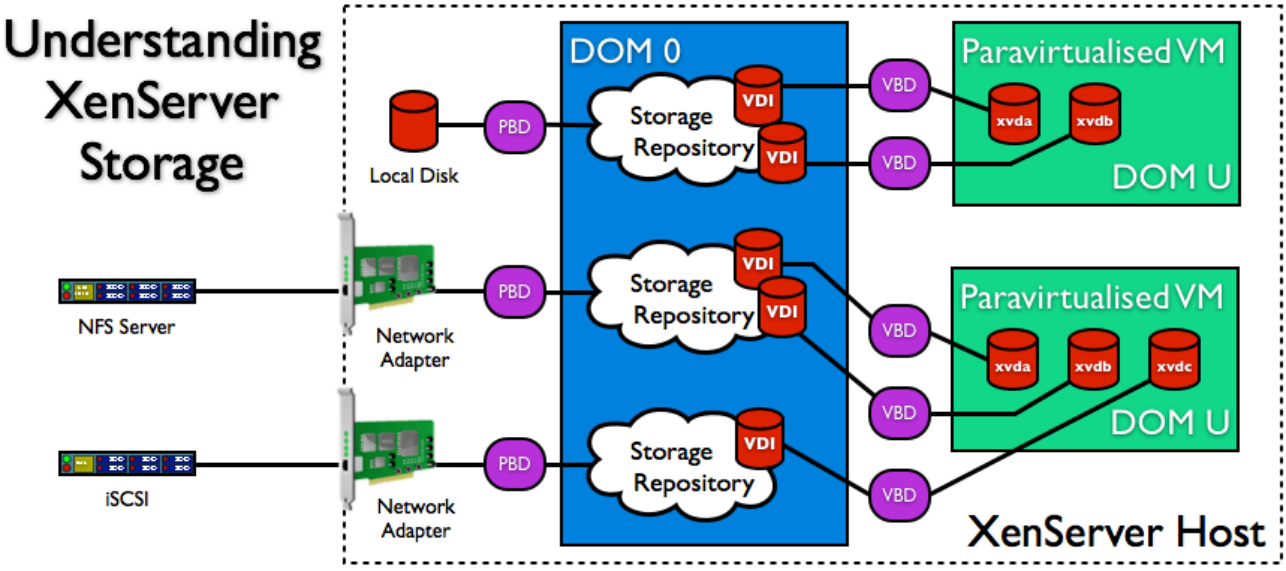

Figure 1. An overview of the XenServer storage subsystem.

Storage Repository Types and Implementation

As already discussed, different types of storage are abstracted by XenServer in the form of an SR. The table below lists the types of SRs covered by this guide, what constitutes a PBD and how VDIs are represented.

SR Type	PBD Representation	VDI Representation
lvm (default)	LVM volume group over a local block device or partition	LVM Logical Volume
lvmoiscsi	LVM volume group over an iSCSI block device (open-iscsi)	LVM Logical Volume
nfs	NFS mount point from an NFS export	File within the mount point
ext	Linux ext3fs mounted from a local block device or partition	File within the mount point

The LVM SR recognizes a partition or disk as an LVM Physical Volume (PV) and configures an LVM Volume Group (VG) on top of it. Furthermore, an LVM Logical Volume (LV) whose name starts with “MGT” will be added to that VG, and it will contain metadata information about the SR. When a VDI is created on such a type of SR, an LV is added to the VG and it allocates the entire required space (thick-provisioning).

Similarly, for LVM over iSCSI SRs (lvmoiscsi), the same structure is used. When an SR is attached, XenServer will configure and handle a set of iscsiadm commands to initiate an iSCSI session. Once a SCSI block device is registered, the SM will add the LVM layers as outlined above. This is also thick-provisioned.

In contrast, file-based SRs such as NFS and EXT will use a different approach. For NFS, the corresponding SM backend will mount the NFS export in a mount point on the local file system. VDIs will be created in the form of files within this mount point, and by default will be thin-provisioned (they will use a small amount of space for meta data and allocate more space as the VDI grows).

For EXT based SRs, a local disk or partition is formatted with `ext3fs` and mounted on the local file system. Like the NFS SR, VDIs are created as files in this mount point. This SR will be used as the default if the checkbox “Enable thin provisioning (Optimized storage for XenDesktop)” is selected during the installation.

Understanding how different storage targets are connected to dom0 is fundamental prior to evaluating performance issues. Furthermore, it is important to understand how VDIs are implemented in the different types of SRs. As an example, consider the case where thin-provisioning (that is, on a file based SR) is selected for a host holding many VDIs that grow concurrently. Due to external fragmentation, the performance of such a system can start to degrade over time.

Virtual Block Device Implementation

In order to attach a VDI to a VM and expose it as a device, XenServer makes use of a number of components. This is independent of the SR type. The components involved in this process are abstracted as a VBD. It is particularly relevant for this guide to expose and explain these components and how they interact with each other.

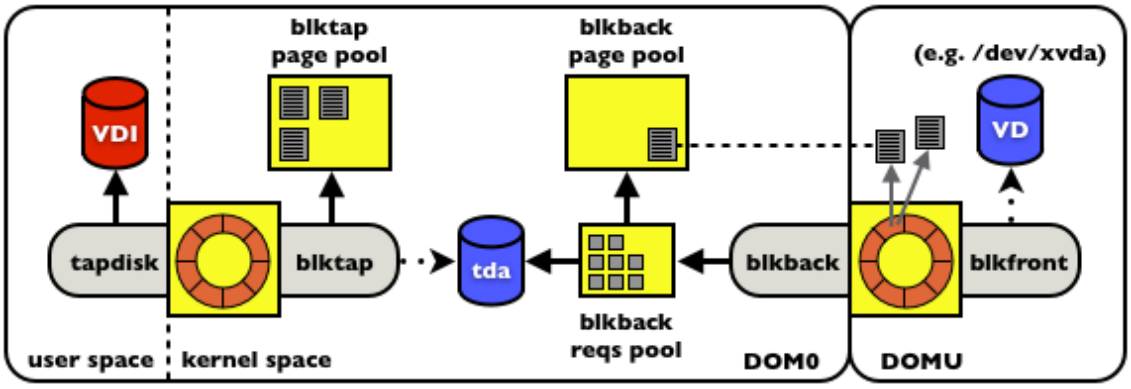

Figure 2. Components involved in the VBD implementation.

The figure above shows the components involved in the datapath for one attached VBD on a guest with paravirtualized storage drivers (for example, Linux's blkfront). On the domU side, we see the virtual device (/dev/xvda) implemented by the blkfront driver. This driver negotiates and allocates a communication ring with blkback. A default ring occupies one 4 kbyte memory page (by default) and can hold up to 32 outstanding requests between the two drivers. Each request, on the other hand, can address up to 11 memory pages of 4 kbytes each.

When a request reaches blkback (now in dom0 kernel space), it is placed on the blkback request pool. dom0 will also receive grants for the memory pages addressed by those requests. These grants are placed in the corresponding blkback page pool (as indicated by the dotted line between the pages in Figure 2). Starting in XenServer 6.1.0, one blkback page pool exists per SR.

The next step is to pass these outstanding requests to the corresponding VDI. VDIs are handled by a user space process called tapdisk. A few of this process's functionalities are error control, handling the VHD format and snapshots. Tapdisk communicates with blkback through a kernel module called blktap, which implements a block device in dom0. This block device is the representation of the VDI as a standard local disk (represented by "tda" in Figure 2). Tapdisk and blktap share a communication ring which can also hold 32 requests (similar to the ring between blkfront and blkback).

Finally, blktap also has a page pool where it keeps copies of the pages associated with outstanding requests. Both blkback and blktap page pools can hold, by default, 704 pages each (this corresponds to 64 requests addressing 11 pages each).

The next chapter of this guide explains how to identify these components and monitor their activity.

Storage Subsystem Components

Relevant components involving a VBD

Based on Figure 2 illustrated in the previous chapter, there are several components that are involved in the implementation of a VBD. The information contained in this section applies exclusively to XenServer 6.1.0, but is also accurate for XenServer 6.0 and XenServer 6.0.2 when all hotfixes have been applied. This section describes how to manually identify which processes these components correspond to.

• Blkfront and blkback communication ring

When a request is generated within a VM, blkfront will find the next free slot in its communication ring with blkback and place the request there. Blkback will be notified of a change in the ring and process the new request. At any time, it is possible to observe the status of the ring from dom0 by examining the `sysfs` entry name `/sys/devices/xen-backend`. This can be done as follows:

```
[root@host ~]# cd /sys/devices/xen-backend
[root@host xen-backend]# ls -d vbd*
vbd-1-51712  vbd-1-51728  vbd-2-51712  vbd-2-51728
[root@host xen-backend]#
```

Within the `sysfs` directory `/sys/devices/xen-backend`, it is possible to find a series of entries named according to the format “vbd-DOMID-VBDID”. The “DOMID” corresponds to the domain ID of the running VM and “VBDID” corresponds to the ID of that VBD. In the example above, there are two running domains (1 and 2), each associated with two VBDs (51712 and 51728). It is possible to match domain IDs to VMs by querying XAPI or by running the program `/opt/xensource/bin/list_domains`.

```
[root@host ~]# cd /sys/devices/xen-backend/vbd-1-51712
[root@host vbd-1-51712]# cat io_ring
nr_ents 32
req prod 23392 cons 23392 event 23393
rsp prod 23392 pvt 23392 event 23393
[root@host vbd-1-51712]#
```

From a VBD entry in `/sys/devices/xen-backend/`, it is possible to examine the contents of the `io_ring` entry as illustrated above. The first line will show the size of the ring (in this case, 32 requests). Following that, it is possible to see how many requests have been produced by blkfront and consumed by blkback, as well as responses produced by blkback and consumed by blkfront. The value indicated as `event` is an internal counter and can be disregarded for the purposes of this guide.

• Blkback outstanding request pool

Starting with XenServer 6.1.0, the outstanding request pool will grow as much as needed to accommodate virtually any number of requests. This is used by blkback to keep track of the requests that are currently inflight. Previous releases of XenServer had a fixed size pool for this purpose and allowed the size of the pool to be configured at boot time through the `blkbk.reqs` option in the `/boot/extlinux.conf` file.

To prevent I/O requests from consuming too much memory in dom0, however, the storage subsystem will limit the number of pages that are addressed by outstanding requests. The next subsection explains this topic in further detail.

• Blkback page pool

When requests are inflight, dom0 requires access to the corresponding memory pages that reside in guest memory regions. This access is realized through Xen grants. Blkback allocates a memory region (indicated in Figure 2) as the blkback page pool for keeping track of these grants.

As mentioned in the previous chapter, this pool can hold, by default, grants for 704 pages. This number corresponds to 64 requests (or the traffic of two full communication rings) where all requests address the maximum amount of data allowed (11 pages each). When the pool is full, requests are not removed from the communication rings until responses are processed and grants are available.

However, it is important to notice that there is one pool per SR by default. This means that traffic within one SR will not affect the datapath of VBDs in other SRs. It is also relevant to mention that the page pools are only created when the first VBD is plugged for that SR.

```
[root@host ~]# cd /sys/kernel/blkback-pools/
[root@host blkback-pools]# ls
9910631a-9c95-fbb1-cc02-cef665962aa5
[root@host blkback-pools]# ls 9910631a-9c95-fbb1-cc02-cef665962aa5/
free  size
[root@host blkback-pools]#
```

As illustrated above, it is possible to see all blkback page pools currently allocated by listing the contents of the `/sys/kernel/blkback-pools` directory. Each pool will be represented by a subdirectory and is named after the SR UUID it corresponds to. Inside each subdirectory, two entries called `size` (representing the total number of pages) and `free` (representing the number of free pages) can be opened for reading. They can be verified as shown below:

```
[root@host blkback-pools]# cd 9910631a-9c95-fbb1-cc02-cef665962aa5/
[root@host 9910631a-9c95-fbb1-cc02-cef665962aa5]# cat free
704
[root@host 9910631a-9c95-fbb1-cc02-cef665962aa5]#
```

• Blktap block device and page pool

Blktap is a kernel component in dom0 that implements block devices representing the VDIs for plugged VBDs in that host. In XenServer 6.1.0, any operation that requires access to VDIs (such as VDI Copy, VM Import/Export or even Storage Migration) will access disk contents through a blktap device.

In its current implementation, blktap also requires a pool to keep copies of guest memory pages during the servicing of requests. Similarly to blkback, the existence and usage of these pools are viewable through `sysfs` (in the `/sys/class/misc/blktap-control/pools` directory) as shown below.

```
[root@host ~]# cd /sys/class/misc/blktap-control/pools
[root@host pools]# ls
9910631a-9c95-fbb1-cc02-cef665962aa5  default
[root@host pools]#
[root@host pools]# ls 9910631a-9c95-fbb1-cc02-cef665962aa5/
free  size
[root@host pools]#
```

Again similarly to blkback, each pool is named after its corresponding SR. They are allocated when the first VBD for that SR is plugged (and released when the last VBD for that SR is unplugged). To examine its size or amount of free pages in the pool, the `size` and `free` entries can be opened for reading (as illustrated in the previous section for blkback).

Although the communication ring between blktap and tapdisk is not exposed in `sysfs`, there are other interesting debugging mechanisms available. To access them, it is first necessary to discover the minor number for the blktap instance corresponding to the VBD at hand. This can be found by examining the `physical_device` entry in a VBD subdirectory in the aforementioned device list at `/sys/devices/xen-backend`. The `physical_device` entry is readable and contains the blktap block device minor number.

```
[root@host ~]# cd /sys/devices/xen-backend/vbd-1-51712
[root@host vbd-1-51712]# cat physical_device
fd:3
[root@host vbd-1-51712]#
```

It is important to notice that the minor number exposed in the `physical_device` entry (as shown above) is expressed in hexadecimal. Once it is acquired and converted to decimal, the minor number can be used to visit another set of entries in `sysfs`. These are located at `/sys/class/blktap2/blktapMINOR` (where blktapMINOR would be "blktap3" in the example above). As XenServer 6.1.0 uses the second generation of the blktap implementation, the `sysfs` class is named "blktap2".

Within the `/sys/class/blktap2/blktap3` directory (in this example), there are a number of entries that will be relevant for the purpose of debugging the system and identifying the processes that connect to each other. For example, `pool` reveals the blktap pool ID this thread belongs to and `task` indicates the PID of the corresponding tapdisk user process.

```
[root@host ~]# cd /sys/class/blktap2/blktap3
[root@host blktap3]# ls
debug  dev  name  pool  power  remove  subsystem  task  uevent
[root@host blktap3]# cat debug
tap 253:3 name:'' flags:0x000010
pool:9910631a-9c95-fbb1-cc02-cef665962aa5 pages:704 free:696
disk capacity:16777216 sector size:512
queue flags:0x1a800 plugged:0 stopped:0 empty:1
bdev openers:1 closed:0
begin pending:1
00: usr_idx:00 op:R nr_pages:08 time:1347902536.000825859
end pending
[root@host blktap3]#
```

As shown above, perhaps the most comprehensive entry is called `debug`. Upon reading from it, it is possible to visualize important debugging information. Firstly, we can confirm the blktap page pool which this device is associated to, as well as its size and number of free pages. Further down, between `begin pending` and `end pending`, it is possible to see how many outstanding requests were present at that time. In the example, there was one read request of 8 pages. Finally, it is possible to infer that no other I/O operation was taking place as the number of free pages was 696 (exactly 8 short of the total 704).

• Tapdisk

Last in the storage subsystem stack is the user space process responsible for handling the storage backend corresponding to a VDI. There will be one tapdisk process per VBD. This process will exclusively open the VDI and handle all access to it (based on the requests it receives from the communication ring with blktap).

It is possible to list all running tapdisk processes in a host:

```
[root@host ~]# tap-ctl list
   22425    0    0  vhd /dev/VG_XenStorage-<SR_UUID>/VHD-<VHD_UUID>
   22688    1    0  vhd /dev/VG_XenStorage-<SR_UUID>/VHD-<VHD_UUID>
   22923    2    0  vhd /var/run/sr-mount/<SR_UUID>/<VHD_UUID>.vhd
   23048    3    0  aio /var/run/sr-mount/<SR_UUID>/<VHD_UUID>.raw
[root@host ~]#
```

The above shows that there are four plugged VBDs in the host. The tapdisk processes that attach them have PIDs 22425, 22688, 22923 and 23048, corresponding to blktap devices with minor numbers 0, 1, 2 and 3 respectively. It is also possible to observe if the VDI type is VHD (indicated by "vhd") or RAW (indicated by "aio"). Finally, the last column indicates the location of the VDI: the first two VBDs belong to a block-based SR, while the last two are allocated in a file-based SR (this is deduced from the pathname).

Tapdisk processes have a control unix socket allocated in dom0. The `tap-ctl` tool can connect to this socket and issue a series of commands, including the retrieval of useful statistics regarding the VBD. The following example illustrates that command:

```
[root@host ~]# tap-ctl stats -p 22688 -m 1
{ "name": "vhd:/dev/VG_XenStorage-<SR_UUID>/VHD-<VDI_UUID>", "secs": [ 34942,
2320 ], "images": [ { "name": "/dev/VG_XenStorage-<SR_UUID>/VHD-<VDI_UUID>",
"hits": [ 34942, 2320 ], "fail": [ 0, 0 ], "driver": { "type": 4, "name":
"vhd", "status": null } } ], "tap": { "minor": 1, "reqs": [ 1401, 1401 ],
"kicks": [ 1166, 1259 ] }, "FIXME_enospc_redirect_count": 0, "nbd_mir-
ror_failed": 0 }
[root@host ~]#
```

While each of the above entries are interesting, the two most relevant performance-related data are `secs` and `reqs`. The first indicates the number of 512 byte sectors read and written to the VDI. By reading this value repeatedly over timed intervals, it is possible to obtain the throughput for the VBD. The latter indicates the number of requests sent to the VDI and the number of responses processed. Similarly to the reading of `secs` values, it is possible to infer the number of IOPS by observing how these numbers increase over time. Furthermore, subtracting the number of responses from the number of requests indicates how many requests are inflight at that given moment in time.

Monitoring components usage with xsiostat

To facilitate the collection and processing of all performance-related data from the components listed in the previous sections, XenServer 6.1.0 provides a new tool called `xsiostat`. This tool can be called from dom0's command line and will output at every second (unless specified otherwise), displaying performance statistics grouped by VBD.

```
[root@host ~]# /opt/xensource/debug/xsiostat -h
-----------------------------
 XenServer Storage I/O Stats
-----------------------------
Usage: /opt/xensource/debug/xsiostat [ -h ] [ -d <domain_id>
        [ -v <vbd_id> ] ] [ -i <interval> ] [ -o <out_file> ]
       -h               Print this help message and quit.
       -d               Domain ID (run list_domains for a list).
       -v               VBD ID (run xenstore-ls
                        /local/domain/<dom_id>/device/vbd
                        for a list).
       -i interval      Interval between outputs in milliseconds
                        (1000 = 1s, default=1000).
       -o out_file      File to write the output to
                        (in binary format).
[root@host ~]#
```

It is possible to change the output frequency by specifying a new time interval with the "-i" command line option. It is also possible to filter domains and VBDs by using the "-d" and "-v" options respectively, as shown in the help. When running `xsiostat` without parameters, it outputs a set of data every second. The output groups VBDs according to the blkback page pools to which they belong. It is expressed as follows:

```
--------------------------------------------------------------------------
pool: 164d3080-00a6-2b08-69f7-8c3fc75cba07 ( 704,  374)
      DOM   VBD  BLKBKRING  INFLIGHT     BLKTAP REQS/s     BLKTAP MB/s
                  TOT USE    RD  WR        RD      WR       RD     WR
 vbd:   5,51728: ( 32,  2) (  2,  0) (  56.00,   0.00) (  2.52,  0.00)
 vbd:   6,51728: ( 32, 32) (  0, 32) (   0.00,1736.95) (  0.00, 78.26)

pool: 9e98b6e2-d2d3-7dc5-43ee-5499b5e514d9 ( 704,    0)
      DOM   VBD  BLKBKRING  INFLIGHT     BLKTAP REQS/s     BLKTAP MB/s
                  TOT USE    RD  WR        RD      WR       RD     WR
 vbd:   6,51712: ( 32,  0) (  0,  0) (   0.00,   0.00) (  0.00,  0.00)
 vbd:   5,51712: ( 32,  0) (  0,  0) (   0.00,   0.00) (  0.00,  0.00)
```

The example shows a system with two attached pools. This implies that there are VBDs attached for two different SRs. It also shows that there are two VMs (with domain IDs 5 and 6), each with two VBDs (one in each pool). For each VBD, several different statistics and performance data can be observed:

- BLKBKRING TOT: Total number of entries in the blkback ring.
- BLKBKRING USE: Total number of requests currently in the blkback ring.
- INFLIGHT RD: Number of read requests currently inflight.
- INFLIGHT WR: Number of write requests currently inflight.
- BLKTAP REQS/s RD: Number of read requests processed per second.
- BLKTAP REQS/s WR: Number of write requests processed per second.
- BLKTAP MB/s RD: Current read throughput in MB/s.
- BLKTAP MB/s WR: Current write throughput in MB/s.

As will be discussed in the next chapter, the size of the blkback ring can be larger than one page (therefore capable of supporting more than 32 requests). Therefore the first item of data exhibited by `xsiostat` is the size of that communication ring (BLKBKRING TOT). Next, it shows how many entries are currently occupied in the ring (BLKBKRING USE).

The next column displays the breakdown of the outstanding requests at the blktap level. It shows how many are read requests (INFLIGHT RD) and how many are write requests (INFLIGHT WR). Because this information is collected from blktap, it is not unusual that the sum of these two values differs slightly from BLKBKRING USE. Also, because the statistics about ring usage and inflight requests is obtained only once for every `xsiostat` iteration (they are instantaneous), it is normal for them to appear zeroed under light loads while the number of IOPS and the throughput are showing activity (they are cumulative).

The pair of numbers that appear next to the pool UUID are the total number of pages available in the pool and the total number of pages currently in use (also an instantaneous metric). There are situations where the blkback page pool is full and requests that could be inflight are being withheld. How to identify and address these scenarios will be discussed in the next chapter.

Next, `xsiostat` shows the IOPS for that VBD. It breaks down the number of requests being processed per second by read (BLKTAP REQS/s RD) and write (BLKTAP REQS/s WR) operations. Finally, the throughput for each VBD is expressed in MB/s and also broken down by read (BLKTAP MB/s RD) and write (BLKTAP MB/s WR). These two metrics are important in identifying cases where the disk scheduler for a particular SR is unfair in serving a VBD that has a larger number of outstanding requests than others. Such cases will be explained and addressed in the next chapter.

This version of `xsiostat` does not support dynamic plugging or unplugging of VBDs. This also means that starting or shutting down VMs while the tool is running will result in incorrect metrics being produced or cause `xsiostat` to quit unexpectedly.

Performance Tuning

Overview

There are cases where certain configuration changes can be applied towards improving the storage performance of particular deployments. Most of these recommendations are not general rules and must be evaluated on a case-by-case basis. Reading the previous chapter is essential in order to completely understand the recommendations stated in this chapter.

Fine-tuning blkback pools sizes

When using `xsiostat` (see section “Monitoring components usage with `xsiostat`”), it is possible to observe cases where blkback will not consume requests from the communication ring with blkfront because the corresponding page pool is full. This usually means that the host could possibly benefit from a larger number of inflight requests on that SR. The following example shows the `xsiostat` output for a blkback pool with five attached VBDs to five different VMs. Each VM is issuing 32 concurrent write requests to its VBD.

```
--------------------------------------------------------------------------
pool: 164d3080-00a6-2b08-69f7-8c3fc75cba07 ( 704,  704)
      DOM    VBD   BLKBKRING  INFLIGHT     BLKTAP REQS/s      BLKTAP MB/s
                    TOT USE    RD  WR         RD       WR        RD      WR
 vbd:  11,51728: ( 32, 32) (  0, 13) (   0.00, 320.99) (  0.00, 14.46)
 vbd:  13,51728: ( 32, 32) (  0, 13) (   0.00, 317.99) (  0.00, 14.33)
 vbd:  12,51728: ( 32, 32) (  0, 13) (   0.00, 336.99) (  0.00, 15.18)
 vbd:  14,51728: ( 32, 32) (  0, 13) (   0.00, 337.99) (  0.00, 15.23)
 vbd:   6,51728: ( 32, 32) (  0, 12) (   0.00, 336.99) (  0.00, 15.18)
```

The workload above was generated by issuing a series of 1408 KB write requests from each VM to its corresponding VBD. Because a request can only address 11 pages (44 KB) of data, blkfront will break each 1408 KB request into 32 requests of 44 KB, filling its ring. Due to the large nature of these requests, the blkback page pool will be full when 64 requests are inflight. This is observable by noticing that all blkback rings are full, but only 64 requests are inflight at any moment in time, limiting the throughput of each VBD.

By default, a blkback page pool is set to support up to two full rings of 32 requests each. This value is configurable through XAPI and can be set to four or eight full rings per SR. If set to four rings, the page pool for that SR will hold up to 1408 page grants. Similarly, if set to eight rings, the page pool for that SR will hold up to 2816 page grants. It is important to note, however, that the page pool size is set during its creation (when the first VBD for an SR is plugged). Changing the size of existing page pools is not supported and it is necessary to unplug all VBDs for that SR for the changes to take effect. The size of the page pool for an SR can be set as follows:

```
[root@host ~]# xe sr-param-set uuid=<SR_UUID> \
                 other-config:blkback-mem-pool-size-rings=<NR_RINGS>
[root@host ~]#
```

The *<SR_UUID>* value must correspond to the UUID of the SR to be changed. The *<NR_RINGS>* value can be set to 2, 4 or 8. The table below can be used as a guideline for the number of page grants obtained with each value:

NR_RINGS	Page Pool Size
2 (default)	704
4	1408
8	2816

Finally, it is important to note that while the actual pages reside in the guest memory space, the total amount of low memory available to dom0 is reduced when a blkback page pool is allocated (due to page addressing restrictions). Increasing the size of page pools will reduce the number of SRs supported in a host and can result in other performance and stability issues.

Fine-tuning blktap page pools sizes

The previous tuning advice showed how to increase the size of the blkback page pools. This, however, does not affect the blktap page pools (which also exist per SR and are also set to 704 pages by default). **Note that increasing the size of blktap page pools is not currently supported**. However, it can be achieved by editing the Storage Manager backend. This is done by editing the file `/opt/xensource/sm/blktap2.py`.

On line 1187, it is possible to add the following statement (marked in bold):

```
try:
    blktap.set_pool_name(sr_uuid)
    blktap.set_pool_size(2816)
except OSError, e:
    if e.errno == errno.EEXIST:
```

After inserting the statement above, all newly allocated blktap pools will support up to 2816 pages. It is also possible to use any of the values suggested for blkback (704 or 1408). **As this is an unsupported tuning**, other changes can be made to this Python script in order for this tuning to affect only certain pools.

Fine-tuning communication ring sizes

While most workloads perform as expected with up to 1408 KB of outstanding data, some require support for larger capacities. The blkback module in XenServer 6.1.0 supports the negotiation of communication rings larger than a single page with blkfront. It is possible to configure blkback to negotiate rings as large as 8 pages. The table below illustrates how many requests can fit into different sized rings.

Ring Order (power of 2)	Ring Size (pages)	Number of Requests	Inflight Data Capacity
0 (default)	1	32	1408 KB
1	2	64	2816 KB
2	4	128	5632 KB
3	8	256	11264 KB

There are four important points to observe when considering the increase of the size of the communication ring between blkback and blkfront:

1. **Blkfront support**: it is necessary that blkfront also support the negotiation and handling of multi-page communication rings. The PV drivers distributed with XenServer 6.1.0 for Windows guests will support rings of up to 4 pages (ring order set to 2). Most Linux guests will only support rings of 1 page. Some implementations, however, may support up to 8 pages and it is necessary to check individual blkfront details for confirmation.

2. **Host-wide settings**: the configuration of blkback to negotiate larger rings is host-wide. This means that it is not possible to enable support of larger rings only for certain VBDs. Once configured, blkback will offer all new negotiations with the possibility of using multiple rings. This also means that it is possible to have inconsistent settings in distinct hosts within the same pool. Migrating a VM to a different host will cause the rings to be renegotiated, possibly affecting the system behavior.

3. **Specified as order**: the blkback setting for the maximum number of pages to negotiate is given in terms of powers of two. As per the table above, the default of zero uses one page. The maximum allowed value is three, which uses up to eight pages (supporting 256 inflight requests).

4. **Blktap rings**: this configuration only applies to the communication ring between blkback and blkfront, not affecting the communication ring between tapdisk and blktap which is not configurable (and always use one page, supporting 32 outstanding requests). While this limits the amount of inflight data at a different level of the datapath, there are reports that some workloads enjoy higher throughput when using larger rings in blkfront.

To enable multi-page rings in a host, the `sysfs` entry below must be altered as follows:

```
[root@host ~]# cd /sys/module/blkbk/parameters/
[root@host parameters]# cat max_ring_page_order
0
[root@host parameters]# echo 3 > max_ring_page_order
[root@host parameters]# cat max_ring_page_order
3
[root@host parameters]#
```

The above example illustrates how to configure blkback to support up to eight pages when negotiating new communication rings with blkfront. In order to make this setting persistent across reboots, it is also possible to realize this configuration on the boot configuration file. This is done by editing `/boot/extlinux.conf` and adding the following statement (marked in bold) to the appropriate boot section (in this example, the sectioned labeled as "xe"):

```
label xe
  # XenServer
  kernel mboot.c32
  append /boot/xen.gz mem=1024G dom0_mem=752M,max:752M <...>
  --- /boot/vmlinuz-2.6-xen blkbk.max_ring_page_order=3 <...>
  --- /boot/initrd-2.6-xen.img
```

The parts marked as <...> have been omitted for the sake of simplifying the example.

Choosing an appropriate I/O scheduler

When using block-based SRs, different I/O schedulers (also called *elevator algorithms*) are applied to the corresponding block devices. These algorithms implement techniques that aim to better utilize the underlying storage infrastructure. As examples, they may group requests together according to locality, reducing seek times on mechanical disks or be specifically configured not to do so when being applied to SSD arrays.

XenServer will apply different I/O schedulers depending on the SR type. These defaults are based on best practices for general cases. For example, it will use the CFQ scheduler for the default "Local Storage" SR, assuming it is the host's local disk. Analogously, it will apply NOOP for iSCSI SRs, assuming iSCSI targets would use many disks in RAID and normally being capable of scheduling the requests by themselves.

To check which scheduler is being used for a particular SR, it is first necessary to identify the corresponding block device for that device (e.g. /dev/sda, /dev/sdb, etc). Then, the following command can be issued (in this example, for "sda"):

```
[root@host ~]# cat /sys/block/sda/queue/scheduler
noop anticipatory deadline [cfq]
[root@host ~]#
```

The output will indicate all available schedulers, marking the one in use with brackets (in the example above, CFQ). It is possible to change the scheduler on the fly by writing on top of the corresponding `sysfs` entry. As an example, the following command change the current scheduler to NOOP.

```
[root@host ~]# echo noop > /sys/block/sda/queue/scheduler
[root@host ~]# cat /sys/block/sda/queue/scheduler
[noop] anticipatory deadline cfq
[root@host ~]#
```

It is also relevant to note that most schedulers offer a range of configurable parameters. These can affect the behavior of each scheduler considerably and should be evaluated when possible. Although it is beyond the scope of this guide to address individual options for every scheduler, plenty of publications, discussions and recommendations are available on the Internet regarding specific tuning for different storage infrastructure. As an example, the following command lists the options available for the CFQ algorithm:

```
[root@host ~]# cd /sys/block/sda/queue/iosched/
[root@host iosched]# ls
back_seek_max      fifo_expire_sync  slice_async     slice_sync
back_seek_penalty  low_latency       slice_async_rq
fifo_expire_async  quantum           slice_idle
[root@host iosched]#
```

Finally, to illustrate further the important role of I/O schedulers, it is possible to consider a case where a host has an extra local SCSI disk named “sdb”. When creating an LVM SR on this disk, XenServer will, by default, assign the NOOP I/O scheduler. If two VDIs are created on this SR and assigned to two different Linux VMs, it is possible for one of the VMs to monopolize the disk usage by issuing a very large amount of requests if compared to more usual workloads.

In this example, the VMs are issuing the following synthetic workloads:

Domain ID	Workload
1	dd if=/dev/zero of=/dev/xvdb **bs=1408K** oflag=direct
2	dd if=/dev/zero of=/dev/xvdb **bs=88K** oflag=direct

The VM with Domain ID 1 will be filling its communication ring by issuing 32 concurrent requests of 44 KB of data each. The other VM will only issue 2 concurrent requests of 44 KB of data each. Because the I/O scheduler for the SCSI disk underlying that SR is set to NOOP, no intelligent scheduling is done. This means that the tapdisk process corresponding to the VBD of the second VM will not be able to issue as many requests as the one for the first VM. The output of `xsiostat` for this scenario is as follows:

```
--------------------------------------------------------------------------
pool: 164d3080-00a6-2b08-69f7-8c3fc75cba07 ( 704,  374)
      DOM   VBD  BLKBKRING  INFLIGHT     BLKTAP REQS/s     BLKTAP MB/s
                  TOT USE    RD  WR        RD      WR       RD     WR
 vbd:   1,51728: ( 32, 32) (  0, 32) (   0.00,1782.97) (  0.00, 80.33)
 vbd:   2,51728: ( 32,  2) (  0,  2) (   0.00, 198.00) (  0.00,  8.92)
```

It is possible to observe that the aggregate disk throughput in this case is around 90 MB/s and that 90% of that throughput is dedicated to a single VBD. When changing the I/O scheduler for that SCSI disk to CFQ, which attempts to be fair towards different (tapdisk) processes issuing requests, the following occurs:

```
--------------------------------------------------------------------------
pool: 164d3080-00a6-2b08-69f7-8c3fc75cba07 ( 704,  374)
      DOM   VBD  BLKBKRING  INFLIGHT     BLKTAP REQS/s     BLKTAP MB/s
                  TOT USE    RD  WR        RD      WR       RD     WR
 vbd:   1,51728: ( 32, 32) (  0, 32) (   0.00,1019.13) (  0.00, 45.92)
 vbd:   2,51728: ( 32,  2) (  0,  2) (   0.00, 740.46) (  0.00, 33.36)
```

While the aggregate throughput has dropped slightly, the throughput for each VBD has changed significantly. The CFQ scheduler waits for a certain amount of time while user space processes are issuing requests. It can therefore be "fairer" in terms of passing these requests to the SCSI disk. Which scheduler to use and which effect is desired is a decision to be made by administrators for each of their deployments.

CPU load in dom0, request latency and vCPU pinning

To achieve high storage throughput, two factors are important: low latency (or a high number of IOPS) and high bandwidth (or a large amount of inflight data). Previous tuning advice in this section showed how to allow for more data to be inflight, therefore allowing for larger bandwidth. Request latency, on the other hand, is a more complicated matter.

Because the processing of a request is done by several components inside dom0, it is imperative that XenServer is able to schedule physical CPUs for dom0 to run whenever it requires processing time for its vCPUs. If the hypervisor is unable to allocate physical CPUs to dom0 (in cases where all cores are busy), the latency for serving requests will inevitably increase, causing throughput to drop and page pools' utilization to rise.

Given that modern servers often provide a large number of physical cores, one practical solution is to dedicate some of these cores exclusively to dom0 (not allowing other VMs running on the host to use them). Given that other processes in dom0 may also require CPU power (such as network-related threads or processes for handling the virtualization of HVM guests such as Windows), it is often a good performance recommendation to implement these exclusive pinning techniques when hosts have proper core availability.

A simple way of experimenting with exclusive pinning is to use the "xl" command-line tool to pin dom0 vCPUs to a set of physical cores and all remaining VMs to other cores. This is done automatically by a script called "exclusive-pinning.sh" which can be downloaded from the following URL:
https://raw.github.com/perf101/scripts/master/exclusive-pin.sh.

Apart from pinning, it is also important to assess the load imposed on dom0 by the VMs running on a host and consider increasing the number of vCPUs allocated to dom0. Tests conducted by XenServer Engineering suggest that, given core availability, a host may perform better with 6 or 8 vCPUs assigned to dom0. The following commands show how to change the default number of dom0 vCPUs and cause them to be pinned at boot time:

```
[root@host ~]# cd /opt/xensource/libexec/
[root@host libexec]# ./xen-cmdline --set-xen dom0_max_vcpus=1-6
[root@host libexec]# ./xen-cmdline --set-xen dom0_vcpus_pin
```

The commands illustrated above will automatically modify the `/boot/extlinux.conf` configuration file and take effect on the next host reboot. Pinning dom0 vCPUs at boot time is usually preferred due to cache locality, especially on machines with Non Uniform Memory Access (NUMA). The following command prints the CPU topology of a host:

```
[root@host ~]# xenpm get-cpu-topology
CPU      core     socket   node
CPU0      0        0        0
CPU1      1        0        0
CPU2      2        0        0
CPU3      3        0        0
CPU4      4        0        0
CPU5      5        0        0
CPU6      6        0        1
CPU7      7        0        1
CPU8      8        0        1
CPU9      9        0        1
CPU10     10       0        1
CPU11     11       0        1
CPU12     0        1        3
CPU13     1        1        3
CPU14     2        1        3
CPU15     3        1        3
CPU16     4        1        3
CPU17     5        1        3
CPU18     6        1        2
CPU19     7        1        2
CPU20     8        1        2
CPU21     9        1        2
CPU22     10       1        2
CPU23     11       1        2
[root@host ~]#
```

The example above displays the CPU topology for a NUMA AMD server where the memory access from different cores can have different cost. Looking at the output of `xenpm`, it is possible to infer that the system has two sockets with two NUMA nodes each. Each node has six physical cores.

In practical terms, when a dom0 vCPU runs on CPU23 and allocates memory on that core, the access to that data will be more expensive when the hypervisor reschedules that same dom0 vCPU to run on CPU0. Considering this architecture and the options discussed in this section, it would arguably be good practice to assign 6 vCPUs to dom0 and pin them to cores 0 to 5.

Finally, it is important to note that when dom0 vCPUs are pinned, the hypervisor will never schedule those vCPUs to run on other physical cores than those assigned. This, however, does not prevent the hypervisor from assigning the same physical cores to other guests in the same host. When such a scenario happens, dom0 could be prevented from executing, causing the performance of the whole host to be affected.

To prevent such situations, the VMs should always be pinned to the remaining cores of a host when dom0 pinning is in effect. For the same reasons of memory locality explained above, it is preferred that VM pinning is also done before the VM starts. This can be done by configuring vCPU masking for guests in XAPI. The documentation for these procedures are referred to as “VCPUs-params:mask” and can be found in the “VM Parameters” section (Appendix A.4.27.2) in the Citrix XenServer ® 6.1.0 Administrator’s Guide.

Understanding and profiling storage targets

One aspect of storage that is often overlooked is the performance characteristics offered by different infrastructures such as RAID controllers and disks with ZCAV. The combination of these properties will cause different address ranges to deliver varying throughput for I/O operations mainly due to latency and seek times in processing requests.

Most mechanical hard disk drives today will offer ZCAV properties. This means that there are more sectors towards the edge on the media than in the centre. Because these disks rotate with constant angular velocity, transferring data from the edge of the disk is much quicker than from the centre. As an example, measuring

the time to read data from a 500 GB Seagate ST3500630NS HDD while using 1 MB operations would result in the following throughputs (each datapoint represents 100 read operations of 1 MB):

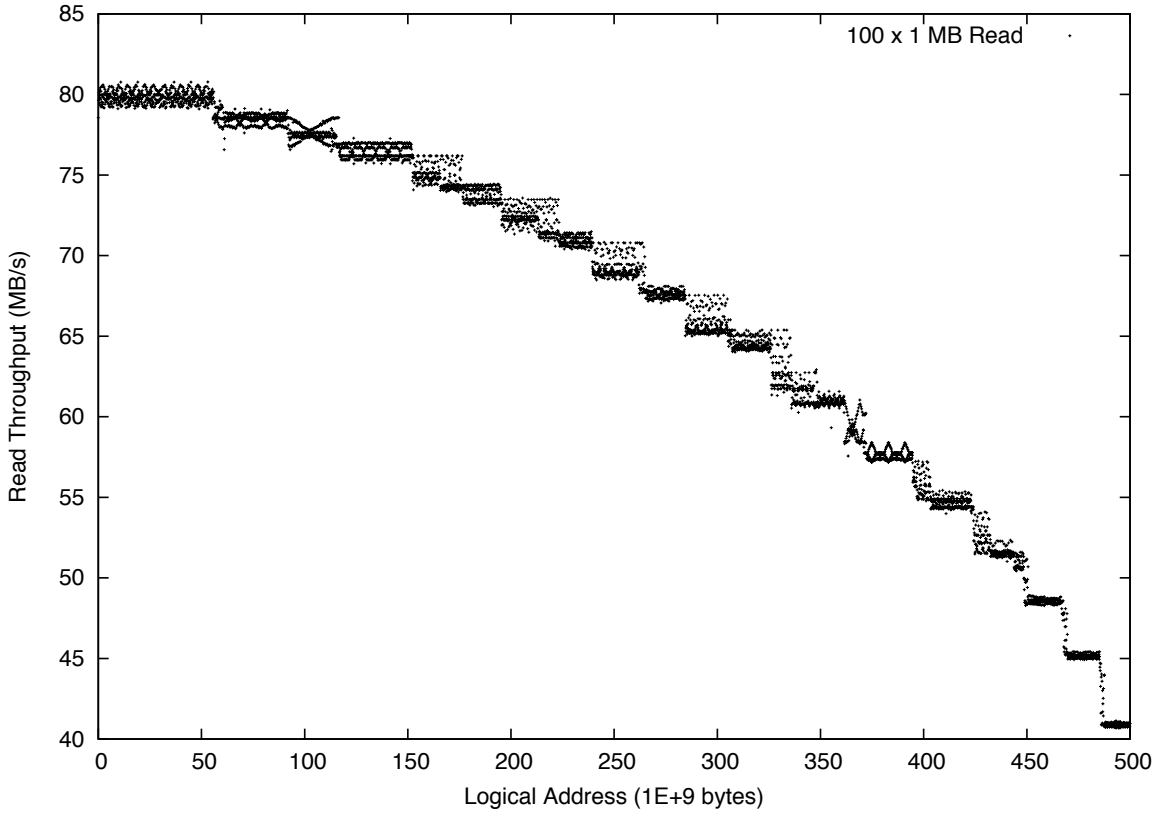

Figure 3. Read throughput across the logical addresses of a ST3500630NS.

The plot shows that the throughput at the edge of the disk (first logical addresses) is about twice as fast in comparison to the centre of the disk (last logical addresses). It is also possible to observe all the zones that this disk possesses. Precisely defining zones, however, might be complicated without an accurate workload (the example above also illustrates that by presenting fuzzy throughput between zones).

This effect can be amplified by grouping disks of this type in RAID. For example, a simple RAID0 could cause the array to perform at around 320 MB/s for reading in the first logical addresses. Similarly, the same array would perform at around 160 MB/s for the last ones. Different controllers (or different configuration sets) could cause the array to be assembled differently, mixing address ranges from different disks to provide a constant throughput of around 240 MB/s.

Before deploying a new SR, we strongly recommend the profiling of the infrastructure. This should be done for single disks or arrays. Understanding performance discrepancies for different workloads between logical addresses can assist, for example, on the design of LUNs. If LUNs (and therefore SRs) are created in accordance to performance profiles, it is possible to assign VDIs that require higher performance to particular SRs, moving them to other SRs as required with features such as Storage XenMotion.

Conclusion

This document describes the storage subsystem of XenServer 6.1.0, focusing on the performance aspects of its components. It shows how different storage backends are organized in VDIs and the datapath structures that allow Virtual Machines exclusive and secure access to their data. It also explains how a set of tools can be used to analyze, diagnose and assess the usage of the subsystem itself, allowing for the identification of bottlenecks and other issues related to performance. Finally, it discusses a series of tuning options that can assist administrators to increase the performance of their deployments.