Masaryk University
Faculty of Informatics

Performance testing of Virtual Data Optimizer storage layer

Master's Thesis

Samuel Petrovič

Brno, Spring 2020

Declaration

Samuel Petrovič

Advisor: Adam Rambousek

Acknowledgements

I would like to thank my self for tremendous help and guidance during writing of this thesis. I would also like to thank Red Hat for collaboration and provision of necessary testing equipment.

Abstract

Abstrakt sa pise nakoniec

Keywords

file system, performance, aging, benchmarking, fragmentation, storage, trim, fs-drift, XFS, VDO

Contents

List of Tables

List of Figures

1 Introduction

Storage requirements of modern society grows exponentialy, but even while cost of storage devices is decreasing, new solutions for optimal storage utilisation need to be implemented. In Linux there is a large family of storage administration tools called *LogicalVolumeManager* (i.e. *LVM*) that provides users with means of managing storage.

By using LVM and other tools available in Linux Kernel, users are able to create complex, layered abstraction which helps them utilise physical storage in the way that suits their needs. This complex structure of physical device management is generally called *storagestack*. In storage stack, different layers represent needed form of abstraction for working with the devices. These layers can have multitude of functions such as software RAID, encrypting, caching, back up, thin provisioning or space saving.

Space saving is delivered by installing the *VirtualDataOptimizer* (i.e. *VDO*) layer. This layer is able to deduplicate and compress user data on the fly, making it possible to create logical volumes larger than available physical space. This storage is therefore *thinlyprovisioned*, however is different from the *thinpool* technology, because (given the user data are compressible enough), it can actually hold more data.

There are two main techniques used by VDO to save space. The first one is *deduplication*, which is a way to identify and eliminate multiple copies of the same data, preventing them to occupy physical space. The second technique is *compression*, that shrinks already deduplicated data furthering the space saving ratio.

Space saving gives users serious advantage, since the cost of installing and maintaining a layer like this is much lower than the cost of purchasing more physical storage. However, on the fly manipulations of large quantities of data may seem costly in terms of usage of other resources like memory or CPU.

An important part of VDO technology is to make sure the costs of running VDO does not out-cost other approaches like purchasing more resources. VDO includes multitude of internal structures and logic, tunable by users, to ensure its advantages.

Because of aformentioned reasons, performance testing is an important part of developing and administering VDO. Users and developement teams need to ensure the final product meets industrial quality standards.

However, performance testing of such a complex technlogy requires extended knowledge of its internal structures and advanced expertise in benchmarking.

This thesis aims to lay a foundational work for performance testing of VDO, describing its workings, performance related structures and issues and describing ways of benchmarking VDO.

Chapter 2 is an introduction of VDO technology. Space saving mechnaisms will be described as well as VDO terminology, device organisation, system requirements, administration and relation to other layers in storage stack.

Chapter 3 presents used benchmarks, FIO and fs-drift, and their usage in testing VDO technology. New features for testing VDO were implemented for fs-drift and will be described in this chapter.

Chapter 5 is focused on performance testing of VDO. The testing is aimed on different VDO components as well as more complex deployment cases. Either way, results of the testing are presented giving recommendation for VDO usage and testin its performance.

In Chapter 6, high-level insight on performance testing of VDO is given with recommendation for further work.

2 Virtual Data Optimiser

2.1 Introduction

Virtual Data Optimizer (VDO) is a block layer virtualisation service in kernel storage stack. VDO enables user to operate with greater logical volume than it's physically available. This is achieved by using compression and deduplication.

Deduplication is a technique that, on a block level, dissalows multiple copies of the same data to be written to physical device. In VDO, duplicate blocks are detected but only one copy is physically stored. Subsequent copies then only reference the address of the stored block. Blocks that are deduplicated therefore share one physical address.

Compression is a technique that reduces usage of physical device by identifying and eliminating redundancy in data. In VDO, lossless HIOPS compression, based on a parallelized packaging algorithm is used to handle many compression operations at once. Compressed blocks are stored in a way that allows the most logical blocks to be stored in one physical block.

The actual VDO technology consists of two kernel modules. First module, called `kvdo`, loads into the Linux Device Mapper layer to provide a deduplicated, compressed, and thinly provisioned block storage volume. Second module called `uds` communicates with the Universal Deduplication Service (UDS) index on the volume and analyzes data for duplicates.

2.1.1 Deduplication

Deduplicaton limits writing multiple copies of the same data by detecting duplicate blocks. Blocks that are duplicate of a block that VDO has already seen are stored as references for that block, which saves space on the underlying device.

Deduplication in VDO relies on growing UDS index. Hash from any incomming block, requested to be written, is serached for in the UDS index. In case the index contains an entry with the same hash, *kvdo* reads the block from physical device and compare it to the requested block byte-by-byte to ensure they are actually identical.

In case they are, block map is updated in a way that the logical block points to the block on underlying device that has been already written. If the index doens't contain the computed hash or the block-by-block comparison indetifies a difference in the blocks, *kvdo* updates the block map and stores the requested block.

Either way, the block's hash is written to the beginning of the index. This index is held in memory to present quick deduplication advice to the VDO volume.

Logical blocks that are copies and therefore share one physical block are called shared blocks.

2.1.2 Compression

Another part of VDO device optimisation is compression. By compressig already deduplicated blocks, VDO provides one more step to increase utilisation of underlying device. Compression is also importatnt for saving space in case the incoming data are not well deduplicable.

In VDO, lossless HIOPS compression, based on a parallelized packaging algorithm is used to handle many compression operations at once. Compressed blocks are stored in a way that allows the most logical blocks to be stored in one physical block.

2.1.3 Zero-block elimination

Zero-blocks are blocks that are composed completely of zeroes. These blocks are handled differently that normal and duplicate data blocks.

If the VDO layer detects a zero-block, it will treat it as a discard, thus VDO will not send a write request to the undelying layer. Instead, it will mark the block as free on a physical device (if it was not a shared block) and updates it's block map.

Because of this, if user wants to manually free some space, they can store a file filled with binary zeroes and deleting it.AAAAAAAAAAAAAAAAAAAAAAAAAAA

2.2 VDO Layer

VDO layer is actually another block device that can aggregate physical storage, partitions etc. On creation of VDO volume, management tool also creates volume for UDS index as well as volume to store actual data.

2.2.1 Physical Size

The VDO volume is divided into continuous regions of physical space of constant size. These regions are called *slabs* and maybe of size of any power of 2 multiple of 128 MB up to 32 GB. After creating VDO volume, the slab size cannont be changed. However, a single VDO volume can contain only up to 8096 slabs, so the configured size of slab at VDO volume creation determines its maximum allowed physical size. Since the maximum slab size is 32 GB and maximum number of slabs is 8096, the maximum volume of physical storage usable by VDO is therefore 256 TB. Important thing to notice is that at least one slab will be reserved for VDO metadata and wouldn't be used for storing data. Slab size does not affect VDO performance.

2.2.2 Logical Size

The concept of VDO offers a way for users to overprovision the underlying volume. At the time of creation of VDO volume, user can specify logical size of volume, which can be much larger than the size of physical underlying storage. The user should be able to predict the compressibility of future incoming data and set the logical volume accordingly. At maximum, VDO suports up to 254 times the size of physical volume which amounts to maximum logical size of 4 PB.

2.2.3 Memory requirements

The VDO module itself requires 370 MB and additional 268 MB per every 1 TB of used physical storage. Users are therefore expected to compute the needed memory volume and act accordingly.

Another module that consumes memory is the UDS index. However, several mechanisms are in place to ensure the memory consumption does not offset the advantages of VDO usage.

There are two parts to UDS memory usage. First is a compact representation in RAM that contains at most one entry per unique block, that is used for deduplication advice. Second is stored on-disk that keeps track of all blocks presented to UDS. The part stored in RAM tracks only most recent blocks and is called *deduplicationwindow*. Despite it being only index of recent data, most datasets with large levels of possible deduplication also show a high degree of temporal locality, according to developers. This allows for having only a fraction of the UDS index in memory, while still mantaining high levels of deduplication. Were not for this fact, memory requirements for UDS index would be so high that it would out-cost the advantages of VDO usage completely.

For better memory usage, UDS's Sparse Indexing feature was introduced to the uds module. This feature further exploits the temporal locality quality by holding only the most revelant index entries in the memory. Using this feature (which is recommended default for VDO) allows for maintaining up to ten times larger deduplication window while maintaining the same memory requirements.

2.2.4 VDO kernel module

The *kvdo* module provides mentioned techniques within Linux device mapper level. Device mapper serves as a framevork for storage and block device management. The *kvdo* module presents itself as a block device that can be accessed directly as a raw device or via installation of supported file systems (XFS/EXT4).

After receiving read request from an above structure, *kvdo* maps the requested logical address to the actual physical block and retrieves the data.

When *kvdo* receives a write reqest, it updates its block map and acknowledges the request. If the received request is either DISCARD, TRIM or a block of only zeroes, *kvdo* only acllocates a physical block for the request.

2.2.5 VDO write policies

VDO can operate in either synchronous or asynchronous mode. By default VDO write policy is set to *auto* which means the the module decides automatically which write policy to enquire. The main difference is in the approach if the block is written immediately or not.

The obvious consequence is that if a system fails while VDO performs asynchronous write, user can lose data.

In synchronous mode, the block is temporarily written to an allocated block and acknowledges the request. After completing the acknowledgement, it attempts to deduplciate the block. In case it's a duplicate, block map is updated in a way that the logical block points to the physical block that's already written and releases the temporal block. If the block is not a duplicate, *kvdo* updates the block map to make the temporarily allocated physical block permanent.

In asynchronous mode, instead of writing the data immediately, only physical block allocation and acknowledgement of the request is performed. Next, VDO will attempt to deduplicate the block. If the block is a duplicate, the module only updates it's block map and releases the allocated block. If the block turns out to be unique, it is then written to the allocated block and the block map is updated.

2.2.6 Storage requirements

As mentioned earlier, at least one *slab* is reserved for VDO metadata and UDS on-disk index.

VDO module keeps two kinds of metadata which differ in the scale of required space.

1. type scales with physical size and uses about 1 MB per every 4 GB of managed physical storage and also additional 1 MB per *slab*.
2. type scales with logical size and uses approximately 1.25 MB for every 1 GB of logical storage, rounded up to the nearest slab.

2.2.7 VDO in Storage Stack

Generally it is importatnt for users to realise that some of the storage layers work better when above or under the VDO layer in the storage stack.

Technology that is recommended to be installed under VDO layer:

- dm-multipath
- dm-crypt, i.e. layer for data encryption

- mdraid, i.e. software raid
- LVM (as software raid)

Technology that is recommended to be installed above VDO layer:

- LVM cache, i.e. possibility to mark part of a block device as a cache to be used by LVM
- LVM Snapshots
- LVM Thin Provisioning

Unsupported configurations are the ones that break those rules plus a few others:

- VDO on top of VDO volume
- VDO on top of LVM Snapshots
- VDO on top of LVM Cache
- VDO on top of LVM Thin pool
- VDO on top of a loopback device
- VDO under an encrypted device
- Partitions on VDO volume
- RAIDs on top of VDO volume

2.3 Administering VDO

2.3.1 Installation

Since VDO is now part of Kernel, it can be installed usign native packaging system. VDO relies on two RPM packages to be installed:

- vdo
- kmod-kvdo

After succesfull installation of these two packages, user can create a VDO volume.

2.3.2 Creating VDO volume

VDO can be created using VDO manager through command line by invoking *vdocreate*. The required parameters are:

- –name=vdoname
- –device=blockdevice
- –vdoLogicalSize=logicalsize

When specifying block device, it is reccommended to use persistent device name. Otherwise, VDO might fail to start properly in case the name of the device changes.

While specifying the logical space, user should be aware what kind of data will be written into the VDO block device and set the logical size accordingly. If heavily compressible data are expected, user can specify logical size as large as ten times the physical size. If the data are expected to be less compressible, it is reccomended to lower the ratio accordingly.

After succesfull VDO creation, the layer is prepared to be used as an ordinary block device. That means, either file system can be created on top of it, or a more complex structure can be installed above. All within the contstrains specified in section 2.2.7.

2.3.3 Monitoring VDO

VDO works as a thinly provisioned volume. Therefore applications or file systems that use it will only see a logical space that is provided by VDO. To monitor VDO space usage, physical space left or compression ratio and much more, *vdostats* utility provides neccessary inspection of VDO volume.

3 Benchmarks

3.1 fs-drift

Fs-drift is an open-source benchmark developed specifically for testing heavy workload and aging performance. Being implemented in Python, it is very easy for users or contributors to add new features or change the benchmark behavior to their needs. For testing performance of VDO, several new features were added.

For performance testing of VDO, new features and behavior needed to be implemented to fs-drift to enable more precise control over IOs and testing process.

3.1.1 Compressible data generator

While testing compression and deduplication technology as VDO, data generated for testing need to be specificaly shaped. The benchmark needs to be able to generate the testing data in a way that corresponds to examined cases. A restriction for VDO testing is that the repeating patterns in compressible data cannot be composed entirely of zeroes, because the way VDO deals with zero blocks (see, Zero blocks elimination). For this reason, new parameter was added to fs-drift, that enables the user to produce buffers using *lzdatagenerator* with specified compressibility.

The parameter works as follows:

- -c | –compression-ratio, number that is a desired compress ratio, e.g. 4.0 is 1/4.0, therefore compressibility will be 75% (default 0.0)

This feature was implemented using alternative IO buffer behavior which works with another open source project called *lzdatagen*.

LZ data generator (i.e. lzdatagen) is a simple but powerfull tool for generating data with desired compressibility. In the simplest use cases, user only specifies the desired size and compress ration and obtains data with the exact qualities.

Usage is:

- ./lzdatagen –size SIZE –ratio RATIO

If there is a compression ratio set other than 0.0, the buffer is filled using lzdatagen call with the desired compression ratio. Otherwise the buffers are filled the same way as in previous versions of fs-drift.

This simple but powerfull feature now empowers the benchmark user to specify the compressibility of written data and therefore makes the user able to measure performance of any technology that deals with compression feature such as ZFS or VDO.

3.1.2 DirectIO

When researching behavior of random operations in fs-drift, file system cache can often skew considerable amount od measurements.

The system cache considereably affects both writes and reads, and the effect is mostly visible with random operations. In fs-drift, $fsync$ time was taken into the measurement to get the real request completion time. However, the system cache succesfully serialises the requests, so at the time of fsync, they are written sequentially.

To battle this effect, option to use direct IO was added. When passing os.O_DIRECT flag to the os.open() call, the UNIX based systems require memory alignemnt to the underlying device so all the operations had to be refactored into being aligned.

3.1.3 Rawdevice

While file system can be installed on a VDO layer (and there is an increasing number of users which do), it is important to have an option to test VDO block device also without the file system to get more precise measueremts.

Rawdevice mode was added, that enables fs-drift to run block-wise on a given device instead of working with files on a file system.

This option is triggered as follows:

The parameter works as follows:

- -R|–rawdevice, (default ”)

3.1.4 Random discard operation

With thinly provisioned technology being increasingly relevant, it is interesting for researchers to also test performance of a block discards.

Discard command could be passed either from the file system or an application to let the underlying device know that the particular block is no longer occupied and can be returned to the block pool.

In Linux, available command for block discard is *blkdiscard*. User can specify the offset and length to be discarded, making it very easy to write an operation type for fs-drift.

If the user of fs-drift wants to test discard speed, he can specify so in the configuration file along with its probability. When the event of discard is triggered, fs-drift works similar as with random writes, but intead of producing buffer and writing, it's using *blkdiscard* with random offset and specified request size to discard blocks.

3.1.5 Data reporting

In previous versions, several problems in measurement reporting had to be addressed.

One of the problems was the data storage. In older versions, data were stored in-memory which posed two problems when running very long tests:

- RAM consumption
- Lose of data after benchmark failure

The new approach that was installed uses a file predestined on a stable device (i.e. system device). Every time the measurement is made, an entry is appended to the file. This means that even after many days of testing, the memory of a system si not cluttered. On top of that, if the system or benchmark fails, there is still some data to be retreived from the file.

Second problem that caused noisy performance data in the previous versions was the use of response times as units of performance measurement. However, since there could be a large range of file sizes, this unit introduces noise, because larger files will take longer to process in what could seem like a decreased performance.

Therefore, the option to store bandwidth was added as a new feature. Bandwidth is measuerd as a total IO size divided by the time of completion. This way, variability of request size will not be affecting the data points.

3.1.6 Performance measurement

In fs-drift, performance is measured by saving a time stamp before invoking the IO operation and storing the time stamp difference after the operation was finished. However, with this type of measurment, it is important to make sure the time stamping is as close to the actual IO operations as possible.

In previous versions, the time measurement was the same for every IO operation, which made some of the measurement incosistent, e.g. taking into the measurement the time to fill buffer etc.

This problem was removed by moving the time stamps inside the functions, providing more precise control over the timing of events.

Another small feature added by switching to $perf\_counter()$ as a tool to log time instead of $time()$.

3.2 FIO

Flexible Input/Output tester is a workload generator known for its flexibility and large base of users and contributors. It is an exellent benchmark for testing block devices and file systems since it gives users very precise control over the workload.

3.2.1 for testing VDO

4 Testing methodology

This chapter presents used testing hardware, preparation of testing environment and performance measuring methodology.

4.1 Testing environment

While testing performance, usage of clean testing environment is strongly encouraged. In time of testing, no other applications should run in the environment to ensure low noise levels. Running tests on clean installation of OS is prefered to ensure no performance impacts caused OS aging, memory shortage, etc.

For this thesis, testing was conducted on instances of RHEL-8.1 and RHEL-8.2 to gain access to the most recent features. Tuning options for the systems were set to $throughput - performance$. Other options for the OS are left to be default.

Storage stack for testing purposes is always prepared by executing a seuqence of LVM commands. For the simples tests, one volume group and one logical volume was used to be a block device for VDO layer.

The storage stack can be tested either as a raw deveice, or file system can be installed on top of it when working with files, or relationship between stack and file system needs to be examined.

It is important to create a fresh instance of stack before the test to ensure stable testing conditions. Also, before every test, we $sync$ and $dropcaches$.

4.2 Testing hardware

Testing for this thesis was conducted on multiple machines provided by Red Hat company. I will introduce testing systems which will be used for multitude of tests with or without the VDO layer installed. These machines were chosen by their computing power, provided memory and by useful storage hardware they are equiped with. These machines are stable systems used by Red Hat Kernel Performance team for regular testing.

4.3 Supermicro X11SPL-F

This machine is used for regular testing of VDO and other complex technologies in Red Hat Kernel Performance team. It's equiped with 4 10 TB SAS rotational drives and one 220 GB SATA SSD for tests that require LVM Cache. The system is always installed on an additional SSD. This machine is equiped with enough memory to handle large VDO volumes.

Machine 1	
Model	Supermicro X11SPL-F
Processor	Intel Xeon Silver 4110
Clock speed	2.10 GHz (8 cores)
Memory	49 152 MB
Testing Hard Drives (4x)	
Model	WD HGST Ultrastar
Capacity	1 TB
Interface	SAS 12 GB
Type	Rotational HDD
Logical sector size	4096 B
Physical sector size	4096 B
Testiting SSD (for LVM cache)	
Model	Micron 5100 MTFD
Capacity	240 GB
Interface	SATA 6 GB
Type	SSD
Logical sector size	512 B
Physical sector size	4096 B
System disk	
Model	SuperMicro SSD
Capacity	126 GB
Interface	PCIe Gen3 x4 Lanes
Type	SSD
Logical sector size	512 B
Physical sector size	512 B

4.4 Data processing

5 Performance of VDO

5.1 Empty VDO

When preparing VDO volume for testing it is important to notice that a logical space of a fresh instance of VDO is not yet fully allocated. Only after writing data, the volume becomes progressively more allocated. Every new subsequent block of data that is written to VDO has to allocate one to four new blocks. This could affect writing performance of new VDO volume.

To examine this effect, random write workload can be used on a new instance of VDO. Results can be compared to a performance of a fully allocated VDO volume or to a device without VDO.

To obtain fully allocated, but empty VDO volume, it is sufficient to just write one zero byte to every VDO sector. One VDO sector is 812 VDO blocks (default block size in VDO is 4096), so we'll write a zero byte every 812*4096 bytes.

5.2 Half empty VDO

5.3 Block map cache

As mentioned in chapter 2, block map cache is an internal VDO structure that keeps part of the journal in memory to provide better performance. Its default size is 128 MB, which is XXX pages, that covers about 100 GB worth of data.

This could pose a performance problem in case the logical space is large enough and the workload access the space in a way the block map cache is tired out. If the block map cache runs out of free pages, it is forced to discard one page and load the need one (which is written to) from disk. This operation is very costly and this could become a bottleneck for VDO performance.

To test this effect, we can use random write workload on an instance with sufficient and insufficient cache size. The parameter to tune cache size in VDO is –blockMapCacheSize=BYTES.

5.4 VDO threads

5.5 Discard size

5.6 Steady state testing

5.7 Testing on different architectures

5.8 Journal performance

5.9 Testing with distributed file system

6 Conclusion