MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Performance testing of Virtual Data Optimizer storage layer

MASTER'S THESIS

Samuel Petrovič

Brno, Fall 2019

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

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 effective data storing need to be implemented. In Linux, large family of storage administration tools called Logical Volume Manager (i.e. LVM) is available to users.

By using LVM tools and other tools available in kernel, users are able to create complex logical structures above physical devices. This complex structure of physical device management is generally called storage stack. 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 as of now compression.

Compression is delivered by installing the Virtual Data Optimizer (or 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 approach seems very advantageous to users, since the cost of installing such layer is much smaller than the cost of buying more physical storage. However, obvious questions arise in terms of performance impact.

In this thesis, such questions will be addresed by thorogh performance testing of VDO layer as well as covering VDO functionality, usage and administration.

This thesis is a follow up on the tesis The Effects of Age on File System Performance and will use similar approaches and testing methods.

In Chapter 3, VDO will be introduced. The reader will get valuable information on VDO terminology, compression and deduplication techniques, device organisation, system requirements, administration, positioning with other layers etc.

In Chapter 4, new features in the benchmark fs-drift will be introduced. The fs-drift benchmark had to undergo some changes so it can be able to test compression layer better. Some changes to data measurements and reporting will also be introduced.

In Chapter 5, proposed testing techniques and configurations will be explained.

In Chapter 6, results and performance impact of administering VDO layeer will be discussed.

2 Related work

In this chapter I present different approaches of file system aging and fragmentation research described and implemented in the past. The first section discuss usage of collected data to create aging work-load. The second section discuss possibilities of aging the file system artificially, without pre-collected data.

3 Virtual Data Optimiser

3.1 Introduction

Virtual Data Optimizer (VDO) is a compression layer in kernel storage stack. Using block virtualisation, it allows for users to operate with a logic volume much greater than physically available. This sort of overprovision is achieved by using block deduplication an block compression.

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

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.

3.1.1 Deduplication

Deduplication in VDO relies on growing UDS index. Any incomming block requested to be written is hashed. The hash is then serached for in the UDS index. In case it is found, reference on the written block is retrieved and returned, succesfully deduplicating the incomming block. The hash entry is then moved to the beginning of the index. In case the hash is not found in the index, the entry with hash is written at the beggining of the index and the block is passed to the compression phase. This index is held in memory to present quick deduplication advice to the VDO volume, therefore there is a minimum requirement of 250 MB of DRAM to be available.

3.1.2 Compression

Compression is an important part of VDO data processing. By compressig already deduplicated blocks, the volume saves even more space. Compression phase is also importatnt for saving space in case the user data aren't don't deduplicate well.

3.2 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 for the actual VDO.

3.2.1 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 contain only up to 8096 slabs, so the configured size of slab at VDO volume creation determines its maximum allowed physical storage. 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.

When trying to examine physical size, two terms are offered. The term Physical size stands for the overall size of undelying device. Available physical size stands for the portion of physical size, that can actually hold user data. The part that does not hold user data is used for storing VDO metadata, f.e. UDS index.

3.2.2 Logical Size

The concept of VDO offers the user means for overprovisioning the physical volume. During 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.

3.2.3 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 reccomended default for VDO) allows for maintaining up to ten times larger deduplication window whlie maintaining the same memory requirements.

3.2.4 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, TRIP or a block of only zeroes, *kvdo* only acllocates a physical block for the request.

3.2.5 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 by computing the hash and searching it in the VDO 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 upadted in a way that the logical block points to the physical block that's already written and releases the physical block allocated at the beginning. 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 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 written to the allocated block and block map is upadted.

3.2.6 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.

3.2.7 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

3.3 Administering VDO

3.3.1 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.

3.3.2 Creating VDO volume

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

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

- –vdoSlabSize=slabsize

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 3.2.7.

3.3.3 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.

4 Fs-drift

Fs-drift is a benchmark developed specifically for testing file system aging performance. It is currently developed as an open source project by engineer Ben England and Samuel Petrovič. During the years of collaboration, both sides usually add features that are important for the individuals line of work, therefore there are currently two working branches. One for testing of shared file systems as Ceph and the like and second for testing local layers that require more advanced testig as VDO or Thin Provisioning testing.

4.1 New features

Fs-drift was used as a main tool in a previous study of file system aging so only new features will be presented in this thesis.

4.1.1 Compressible data generator

For testing a compression layer as VDO, completely random data cannot be used. The benchmark needs to be able to shape the incomming data in a way that simulates compressibility that is expected in real-life user data. For this reason, new parameter was added to fs-drift, which specifies what compress ratio will the incomming data have.

The parameter in question 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 simples use cases, user only specifies the desired size and compress ration and aquires data with the exact qualities.

Usage is:

- ./lzdatagen –size SIZE –ratio RATIO

The new approach in the buffer therefore retreives the desired compression ratio, if it's set on 0.0, normal data are generated as in previous versions. If there is a compression ratio set, the buffer is filled using lzdatagen call with the desired compression ratio.

To address concerns of lzdatagen affecting performance measurements, the generator is filled with data before the measurement is started.

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 layer with a compression feature such as ZFS or VDO.

4.1.2 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 units unsuitable for local performance testing. The unit in question was response time which may be important when measuring performance in distributed file systems but in the kind of testing the previous work and this work is trying to ...dosiahnut..., measuring throughtput would yield much more stable data points.

Therefore, the option to store bandwidth was added as a new feature. Bandwidth is measuerd as a size of request divided by the time of request completion. This way, variability of request size will not be affecting the data points as we could see while using response times where usually larger IO requests will take more time to complete.