Masaryk University
Faculty of Informatics


# Performance testing of Virtual Data Optimizer storage layer

Master's Thesis

**Samuel Petrovič**

Brno, Spring 2020

# 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

THX

# Abstract

Abstrakt sa pise nakoniec

## Keywords

VDO, deduplication, compression, storage, fs-drift

# Contents

# List of Tables

# List of Figures

# 1 Introduction

Storage requirements of modern IT industry 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 *Logical Volume Manager* (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 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, encryption, caching, back up, thin provisioning or space saving.

Space saving is delivered by installing the *Virtual Data Optimizer* (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 *thinly provisioned*, however is different from the thin pool 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 presents users with 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 de-

velopement 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 Linux storage stack. VDO enables user to operate with greater logical volume than is physically available. This is achieved by using deduplication, compression and elimination of zero-blocks.

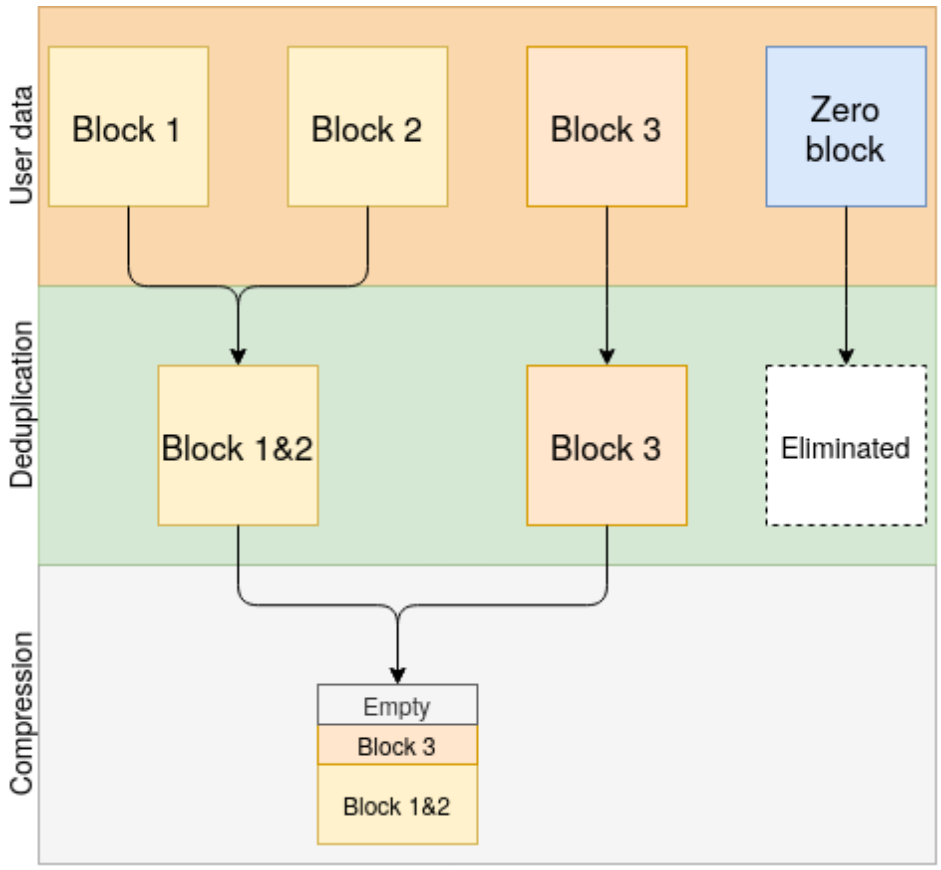


Figure 2.1: Diagram of space saving methods of VDO. In the first step, duplicate data and zero blocks are eliminated. In the second step, remaining unique blocks are compressed and stored as a single block.

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 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 searched 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 doesn'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 blocks 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 optimisation techniques 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 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 entirely of zeroes. These blocks are handled differently that normal 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 delete.

## 2.2 Constrains and requirements

VDO layer is in fact 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 for physically storing date 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 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. As mentioned, 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.

When trying to examine physical size, thee term Physical size stands for 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.

### 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 its logical size, 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

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 *deduplication window*. 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 CPU

## 2.3 Internal structures

Working VDO instance contains structures that handle the incomming events.

#### 2.3.1 Slabs

#### 2.3.2 Recovery journal

Recovery journal provides track of all block changes that has yet to be fully, reliably written to the physical device. It provides performance improvement with bothsynchronous and asynchronous writing policies.

When in synchronous mode, the completion request doesn't wait for the change to be made permanent on the device, it merely waits for the ackowledgement from the journal.

In asynchronous mode, the journal helps providing data loss window by ensuring the user will not lose data if the changes are commited to the journal before the window is expired.

The recovery journal has two parts. One is stored on the physical device and the other in memory to serve as a buffer. When entry is added to the journal, it is processed by the part in memory and is regarded as an active block. An attempt to commit the block to the device is made, however, the device might be locked by another commit thats in progress which makes the commit queue. Every successfull commit will wake others waiting after it is completed.

### 2.3.3 Block map cache

Block map exists in VDO volume to maintain the mapping of logical blocks to physcial blocks and managed as pages with each page holding 812 entries of logical blocks. The entire block map is kept on disk, since it can be rather large.

Block map cache is a subset of the entire block map that is kept in memory for performance increase. If request to access a block comes to VDO, it will check if that block is in the block map cache. In case it doesn't it needs to read the relevant page and store it in a cache. In case the request was write, it is written in the block map cache and only participates to the on-disk part later.

The block map is a cause of one of the requirements on physical space. It usually consumes about 1.25GB per 1 TB of saved stored physical data. The subset that's kept in memory is much smaller, 128MB by default and is tunable by –vdoBlockMapCacheSize.

The 128MB can cover about 100GB of logical blocks. However if users logical space is larger than 100GB and their workload is so random that it doesn't cached pages, user can observe great preformance hit. In case the block map cache is full, therefore runs out of free pages and a request comes for a page thats not contained in the cache, VDO needs to first discard some page from the cache, write it on disk and just after then load the desired page. It is obvious that this cycle is

very time expensive and therefore users are encouraged to increase block map cache size if the precoditions for cache tiering are met.

### 2.3.4 UDS index

UDS index is a structure designed specifically to indentify duplicate data using hash fingerprints of data blocks. It exists in VDO to provide deduplication advice for effective deduplication. Instance of UDS index is not vital for VDO block handling. In case of losing the index, the VDO still manages blocks and stores and compress data, only without deduplication. Even in the event of index becoming corrupted, there are different mechanisms in place to assure data correctness, so user can discard the index and start bulding a new one.

Also, updating index is costly, so VDO is trying to minimise the updates. That is a reason the index can contain references to blocks that are no longer present in data. Those blocks are called stale blocks.

## 2.4 Tunables

### 2.4.1 VDO threads

### 2.4.2 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 use. The main difference is wether or on is the write request written immediately or not. In case of system failure while using asynchronous mode, data can be lost.

Synchronous mode

In synchronous mode, VDO temporarily writes the block to the device 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 iss already written and releases the previously written temporary block. In case the block is not a duplicate, *kvdo* updates the block map to make the temporary physical block permanent.

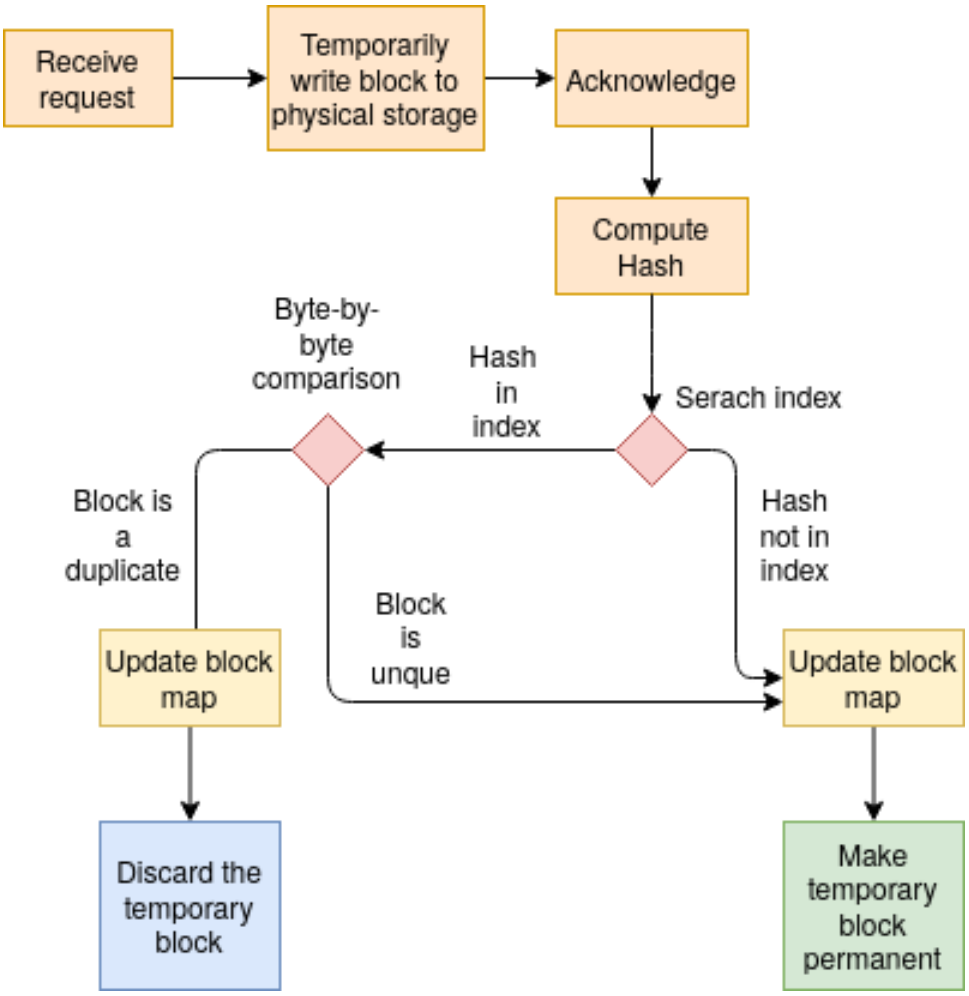


Figure 2.2: Sync diagram

Asynchronous mode

In asynchronous mode, instead of writing the data immediately, physical block is only allocated and acknowledgement of 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 is be unique, block map is updated and the data is written to the allocated block.

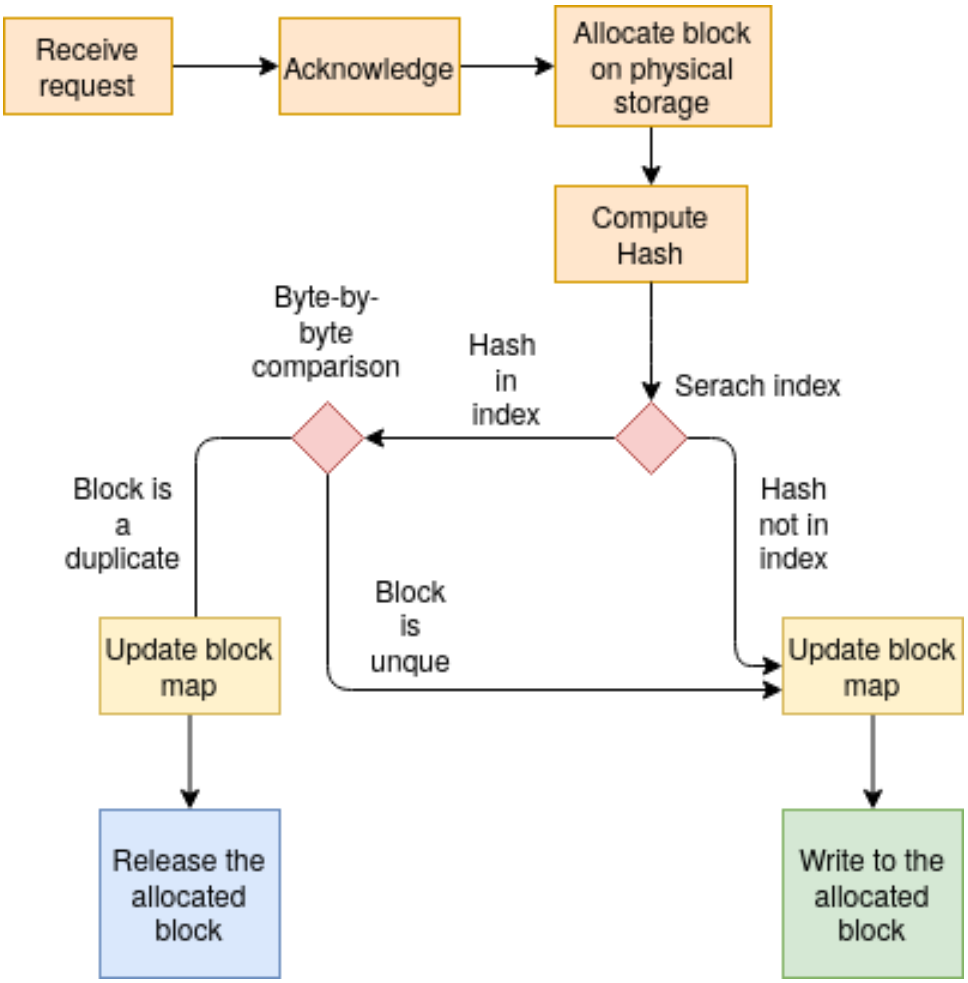


Figure 2.3: Async diagram

Asynchronous unsafe mode

The option async-unsafe is there in case the user wants to keep the original non-ACID async write policy.

### 2.4.3 Block map cache size

### 2.4.4 Discard size

# 3 Fs-drift

## 3.1 Introduction

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.

## 3.2 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.

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. The user can specify compression ratio, size and output file and obtains data with those qualities. Seed for repeatable output can be also specified, which is useful for producing deduplicable workload.

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.

Usage:

- -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)

## 3.3 Deduplication

While some of the data from lzdatagen will be naturally deduplicable, more precise control of deduplicibility of the data could result in more precise performance measuerment.

As mentioned in the section above, user can input seed value to lzdatagen to produce the same output. Fs-drift uses this parameter to produce deduplicable data. Based on the inputed probability, fs-drift will either produce new data by obtaining new seed from os.urandom() function and saving it for later. In case fs-drift needs to produce data that will deduplicate, it uses one of the stored seed from the previously generated data.

Usage:

- -+D|–dedupe-percentage, percentage of data chunks or files that will be deduplicable (default 0)

## 3.4 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.

Usage:

- -D|–direct, if true, use os.O_DIRECT flag when opening files. Data alignment of 4096 bytes will be used. (default False)

## 3.5 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. File systems can have considerable impact on performance and can skew results in ways that make it hard to clearly observe VDO impact on overall performance.

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.

Usage:

- -R|–rawdevice, set path of the device to use it for rawdevice testing (default '')

## 3.6 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.

The pilot idea was to call blkdiscard command using the subprocess module. However, discard operations are very fast and the speed of calling subprocess module has become a significant bottleneck.

Howver, Python provides a framework to directly call ioctls directly from the Python code. First, the code for BLKDISCARD operation needs to be computed. Parameter for BLKDISCARD ioctl is an array of two u_int64 numbers which represent the beginning offset and length

to discard. This structure can be created by using Pythons module struct.

Example 3.1: Using BLKDISCARD ioctl to discard first 4096 bytes from the beginning of the device

```
import os
import struct
from fnctl import ioctl
offset = 0
length = 4096

#opening a device that supports BLKDISCARD
fd = os.open('/dev/sde', os.O_WRONLY)

#computing command for ioctl,
#the value from documentation is _IO(12, 119)
BLKDISCARD =  0x12 << (4*2) | 119

#Creating C-like array of two uint_64 numbers
args = struct.pack('QQ', offset, length)

#Finally, ioctl call with the prepared parameters
ioctl(fd, BLKDISCARD, args, 0)

os.close(fd)
```

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

## 3.7 Random map

When computing offset for random operations, it might not enough to just generate random number. Sometimes it is benefical to administer IOs only to unused blocks, ensuring no overwriting takes place and all the free blocks will eventually be used.

For this purposes a feature to keep track of unused offests was added to fs-drift. The random map is generated before the test as a shuffled list of possible indices, to save time while the workload is in progress.

The user should be wary of the fact that if the test runs out of the random map, it is recomputed again and will start to overwrite data.

Usage:

- -r|–randommap, if true, use random map to get random offsets (default False)

## 3.8 Multithread

Multithreaded workloads are essential for researching performance of ——.

Option to run fs-drift a multithreaded bechmark was added. User will just specify the number of threads that is to be used and fs-drift will spawn that many. For this option, large parts of fs-drift needed to be reengineered so the threads are not corrupting eachother data structures, buffers, etc.

Important fact to notice is that every thread is maintaining its own performance throughput report so it is expected that the user will know how to aggregate and interpret the result.

- -T|–threads, fs-drift will spawn this many threads to run a workload (default False)

## 3.9 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 generate buffers, setting offset to file descriptors. 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 $time()$.

## 3.10 Data reporting

In previous versions, data was stored in the programs memory which increased RAM consumption during long tests. Also in case of OS or benchmark failure all the results were lost. This problem was removed by having the output files open and continually appending new entries.

In fs-drift, gathering only response (completion) times introduced noise to the datapoints, since there can be variability of file or data size, that variability will directly project into the data, since working with larger files can take longer than working with smaller files.

With this in mind, an option to store bandwidth was added as a new feature. Bandwidth is measuerd as a total completed size divided by the time of completion. This way, variability of data size is not affecting the measurements.

# 4 Testing methodology

This chapter presents used testing hardware, setup 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 with fs-drift

Fs-drift is a powerfull tool for administering IOs, simulating many types of workload. To obtain correct performance measurements, it's important to setup benchmark correctly.

### 4.2.1 File size

File size is another parameter that need to be carefully thought of. File size is used both for testing on file system and testing with rawdevice.

It represents the size of data that will be administered block by block to the device or file. Again, since there is some delay between actually invoking the writing process, larger file size will mean more aggresive workload for the device or file system. It is also a granularity at which fs-drift collect data, one record for a given file, so seting file size too high can mean lowering the data points in results.

### 4.2.2 Block size

Choosing the correct block size for the workload is a very important step. For most cases, the native VDO blocksize of 4k is used. However, it is important to understand what does choosing different blocksize mean for the workload.

The way fs-drift works, block size is the smallest granularity of IOs that will the workload achieve, which means that it is a smallest chunk of data that will be submitted to the device or file. Since there is some work to be done between the subsequent submisions, the device that services the requests can have a small bit of time to process the requests. By choosing larger blocksize the workload becomes more aggressive, since there is more data submited at the same time.

However, by increasing block size, we're increasing the smallest continuous chunk of data that will be worked with and some devices (such as HDDs) may benefit from that.

### 4.2.3 Compress ratio

Compress ratio is a very imporant factor while testing VDO. Generally, it is good to use some compression, so the VDO can excercise all its components. However there can be cases where lowering the compression ratio would benefit the user, f.e. if the goal of a test run is to fill as much of physical space as possible, it would be counter-productive to let a portion of the data be deduplicated and compressed.

While testing VDO, it is important to remember that it's using packaging compression and storing compressed fragments in one block. In case the compression ratio of written blocks is lower than 50%, therefore resulting compressed blocks would occupy more than 50% of their original size, there will be no apparent compression

during the test, because the packaging algorithm would not be able to fit multiple blocks into one.

#### 4.2.4 Deduplication

#### 4.2.5 Random map

Most testing workloads that aim to test limits of VDO technology are random writes. This also excercises the ability of VDO to serialise workload for the device underneath. By default, fs-drift is randomly choosing the offset to write the next block of data.

However, this approach may not be best, since some blocks can be used more than once and some blocks may never be used. The act of overwriting blocks can introduce noise to the results, so in case the test is not specifically aimed at reusing blocks, it is reccomended to use randommap.

## 4.3 Testing package

To manage the test results and metadata as well as testing environment and to be able to include tests in more complex or automised workflows, it is beneficial to encapsulate the benchmark into a testing package. For fs-drift, there is drift_job package.

Drift_job accepts several parameters such as used device, command for preallocation or parameters for fs-drift. When run, it prepares the testing environment, gathers data about the system, runs the test and package results into an easily managable tar file. The results can be then automatically sent to a data gathering server.

If specified, the package also asynchronously gathers information while the test is running such as statistics about VDO volume.

## 4.4 Data processing

Data processing is accomplished by using library written for processing drift_job packages. The main object Report will process data from all the specified result packages and creates easy to view html report.

The html report consists of two parts. First part is presenting individual report for given results. The second part is comparing the inputed results in an easily digestable manner.

The report output can be tuned with several parameters:

- list of paths to individual tar packages
- path to store the output
- offset, tuple to control which part of X axis to view
- log window, for approximating data points
- smooth, to let the object know if interpolation and filtering should be used
- chart_vdostats, list of vdostats atributes to plot
- lim_Y to set the upper limit of Y axis
- test_label, to label outputed comparing charts

### 4.4.1 VDO chart

In case the test was run on a VDO volume, the resulting tarball will contain a file with vdostats logs. The data processing script will find all the statistics the user inputs and produces a chart. This chart therefore shows the state of VDO in time, while the test was running. We use it mainly to view how many logical and physical blocks were used. But with testing specific components like block map cache or journal, we can view their statistics easlily on this chart

### 4.4.2 Throughput progression chart

This chart is a simple representation of a measued throughput during the test. Since the data can be sometimes noisy, filtering and interpolation is used to obtain a smooth curve. The way filtering with Savitzky-Golay filter works, if there are revolting data points on the extremes of the X axis, it will cause the curve to turn up or down in hyperbolical maners. If this happens and it hinders the visibility of

the results, offset can be set accordingly so the revolting values are excluded.

This chart is mainly used to confirm there was no event that would speed up or slow down the test. If there is a dramatic change in a throughput progression that was unexpected, the test might be faulty.

Hower, it is very usefull, when there is an expected change in behavior of the underlying device such as Empty VDO or Half-Full VDO test

### 4.4.3 Histograms

### 4.4.4 Boxplots

Boxplots are the main tool used to visually compare performance of different tests. The part in focus is the median, which is the main metrics considered when comparing multiple test runs.

## 4.5 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.

# 5 Performance of VDO

## 5.1 VDO threads

## 5.2 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.

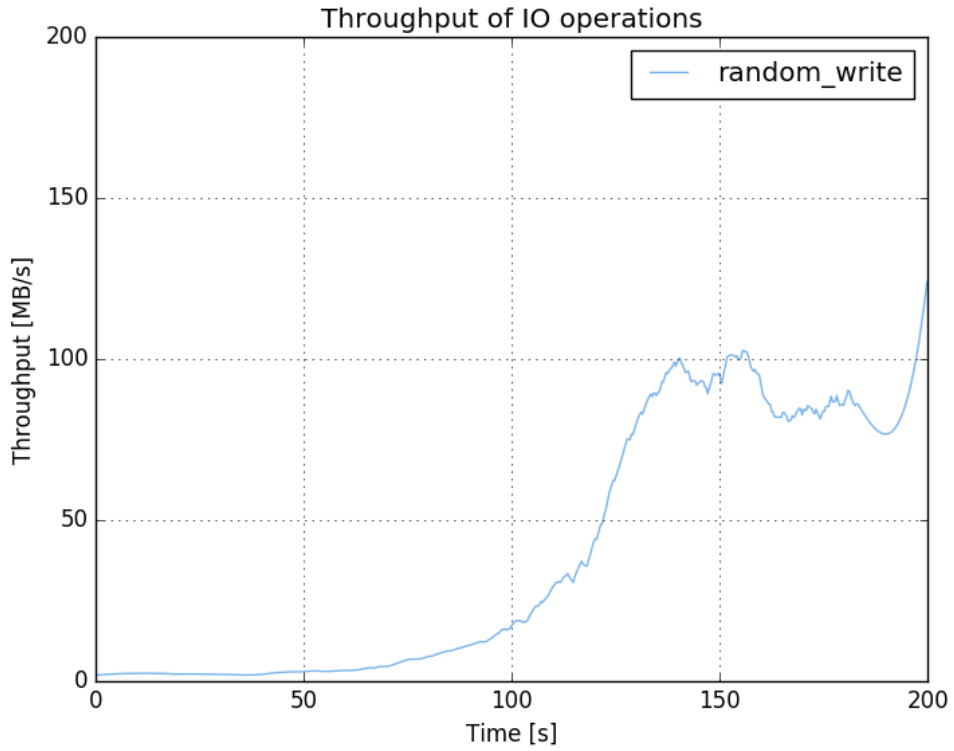


Figure 5.1: EmptyVDO

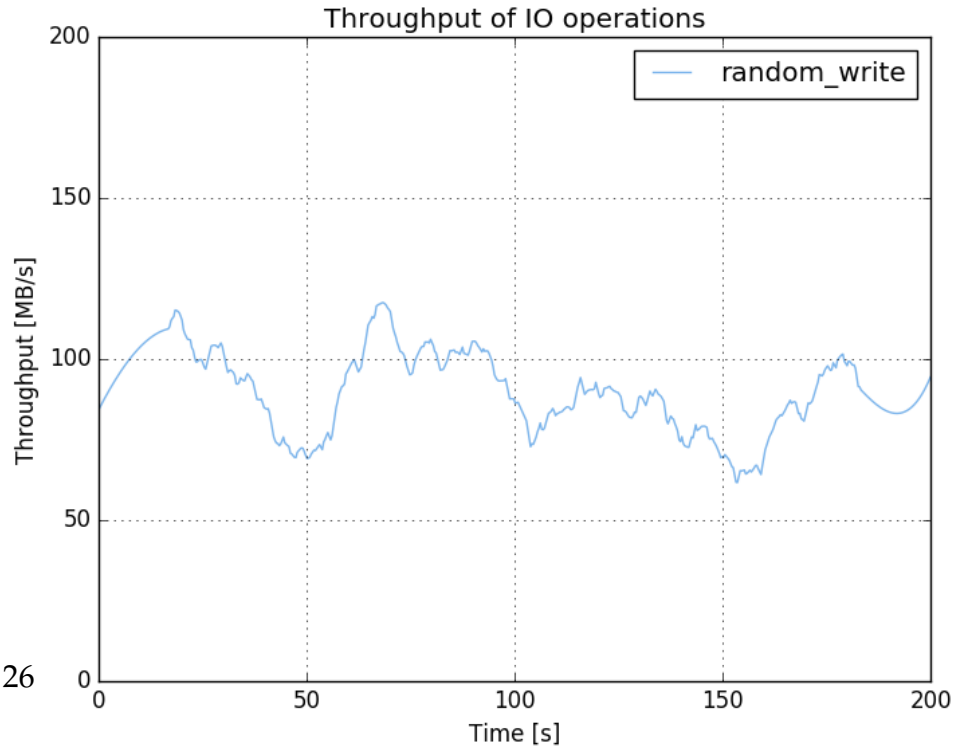


Figure 5.2: PreallocatedVDO

## 5.3 Half empty VDO

## 5.4 Steady state testing

## 5.5 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 fills up, (runs out of free pages), and there is a write request to a non-cached page, the cache is forced to discard one page from its pool to make room for the relevant page. 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.

To ensure the workload is truly aggresive, large file size and completely random write is used so the test makes lots of requests in a short amount of time and the effect of caching pages could be seen.

## 5.6 Maximum discard size

Performance of discard operation is important to consider, since users may want to discard large quantities of blocks to free space.

VDO offers an option to change the maximum allowed discard size with a parameter –maxDiscardSize. VDO will process discards one VDO block at the time to ensure metadata consintency. VDO manual states that lower discard sizes could work better since, VDO can process them in parallel, assuming low IO traffic.

The tests of maximum discard size throughput were conducted on four different VDO volumes created with differend –maxDiscardSize parameter. This progression was tested on empty VDO, full VDO and empty VDO with preallocation.

The results from unallocated empty VDO are not considered, since it is not expected for users to work (and perform discards) on unallocated space.

While processing the data, the interesting statistic to look for from vdostats is `bios in discard`. By observing bios in discard together with logical blocks used and with the progession of throughput, we can see the relationship between performance of discarding data and performance of discarding empty blocks.

As we can see on the Figure **??**, while having normal VDO setup and regular discard workload, the speed of discard operation is decreasing with using larger discard sizes. This test was conducted after a previous run of fs-drift filled all the space with random writes. Filling the storage with random data before every test takes a considerably long time. To shorten the time to test discard operation, we could try testing with empty VDO.

It is important to notice, that while filling the volume with random data might not be necessary, the tests should be done on fully allocated VDO volume. If there is a discard request for a block tha has not yet been allocated, the discard handling is much faster, since no work has been done. This effect could be observed by running the same test on an unallocated storage as presented in Figure **??**.

This is once again a reason for why preallocation of the tested storage is important. The results from empty, but previously allocated VDO show the same behavior as the reults from the test where VDO was filled with random workload at the beggining.

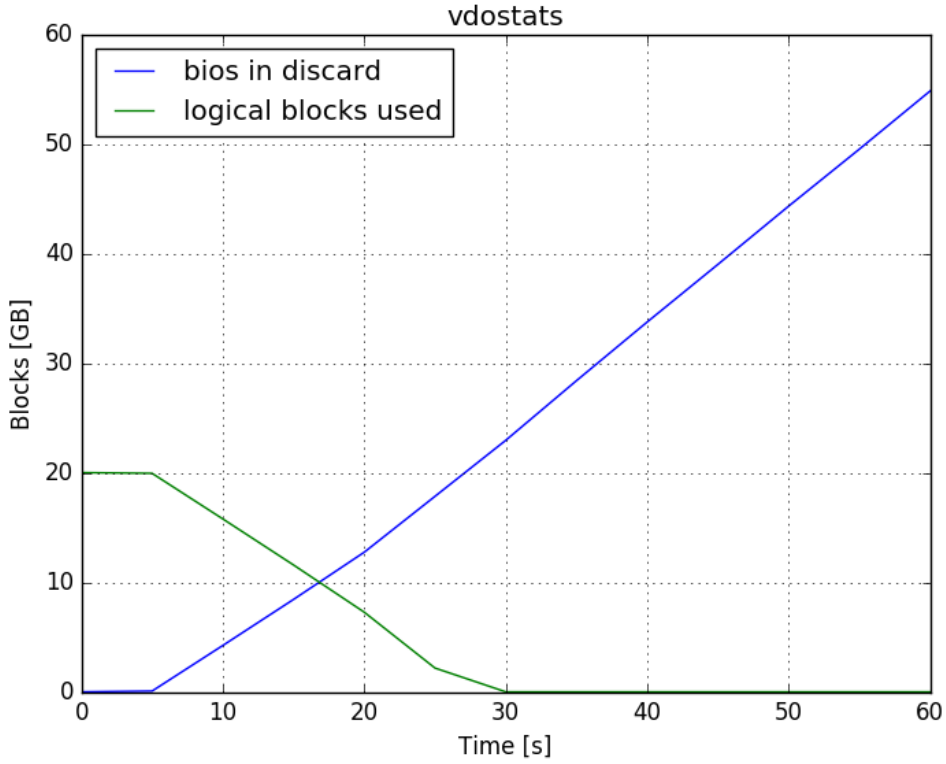


Figure 5.3: Inspecting VDO stats

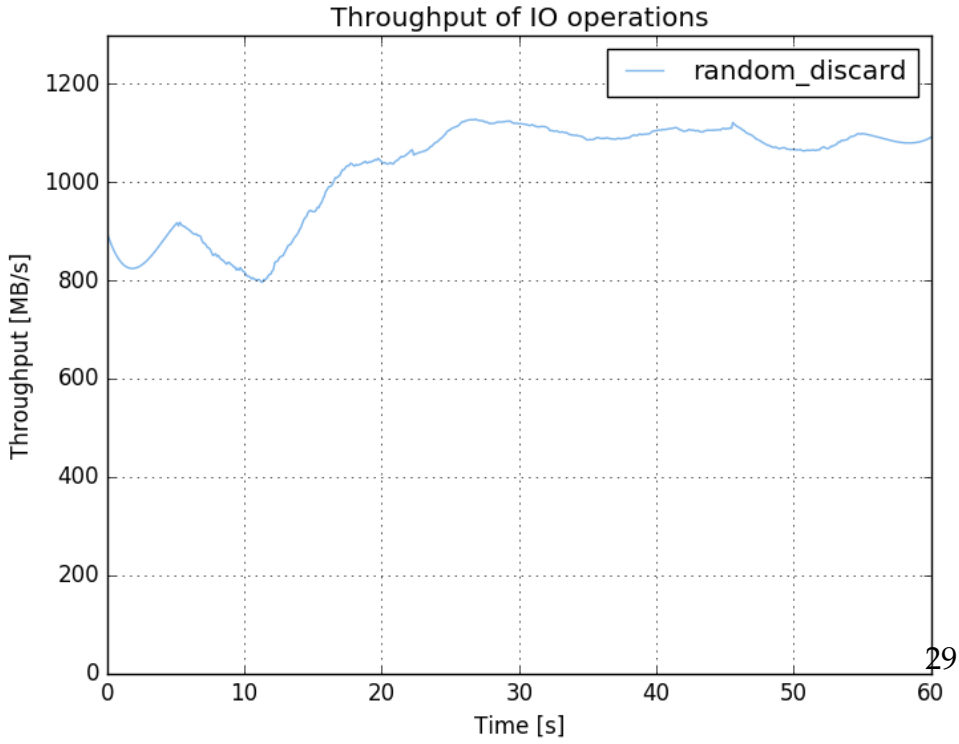


Figure 5.4: Inspecting throughput

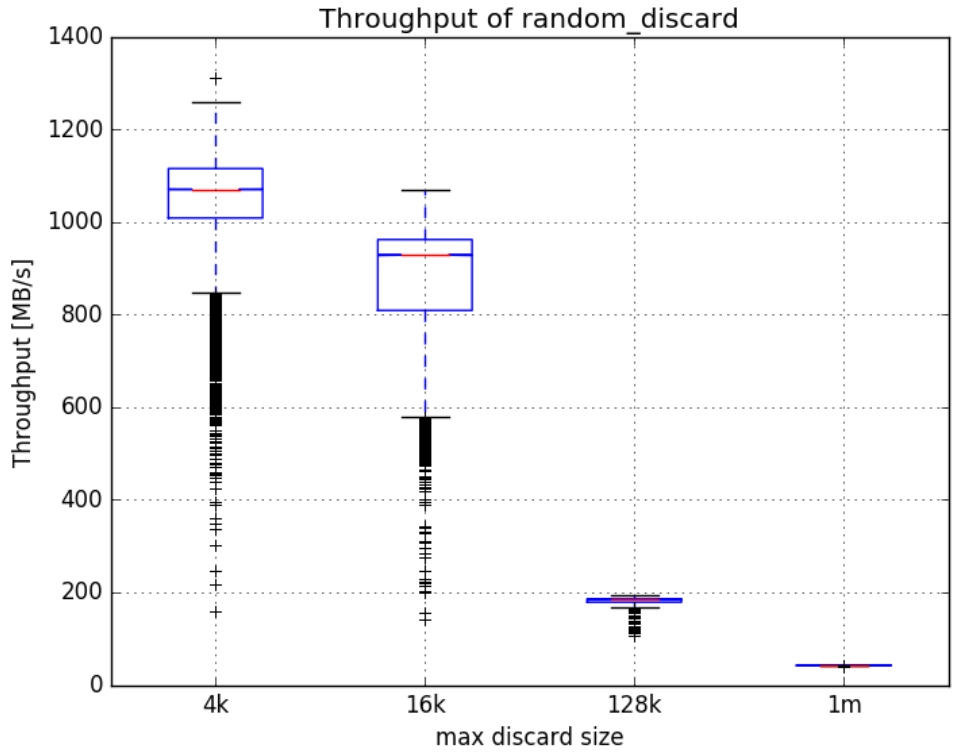


Figure 5.5: Testing of VDO volume created with various maxDiscardSize parameters. Prior to the test, the device was filled with random write workload

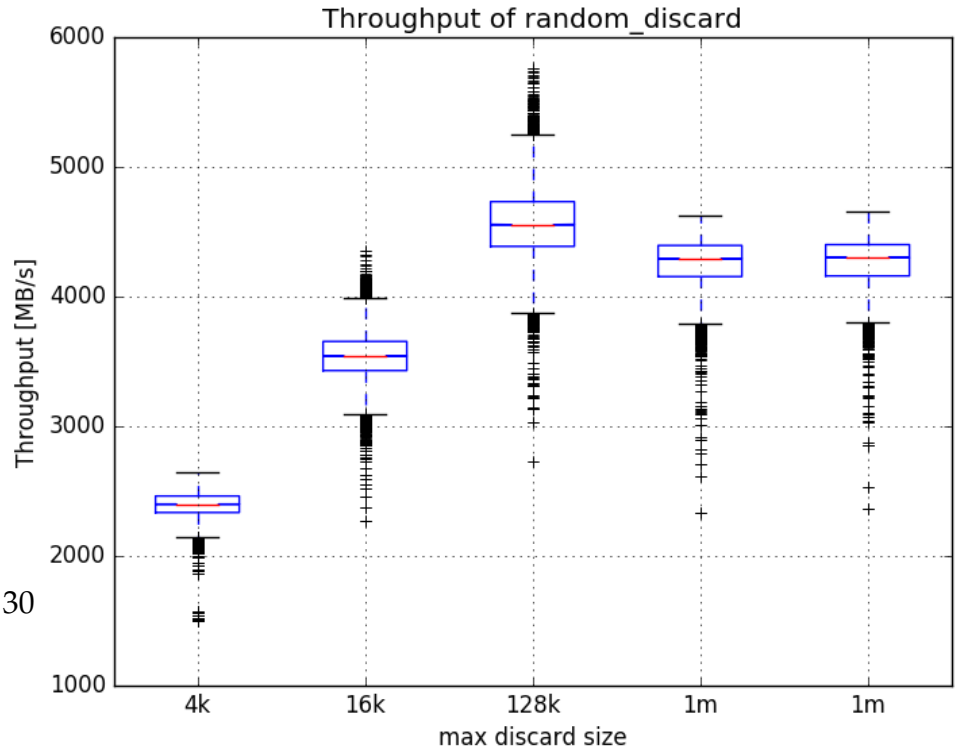


Figure 5.6: Testing of VDO volume created with various maxDiscardSize parameters. This test was conducted on a fresh unallocated instance of VDO

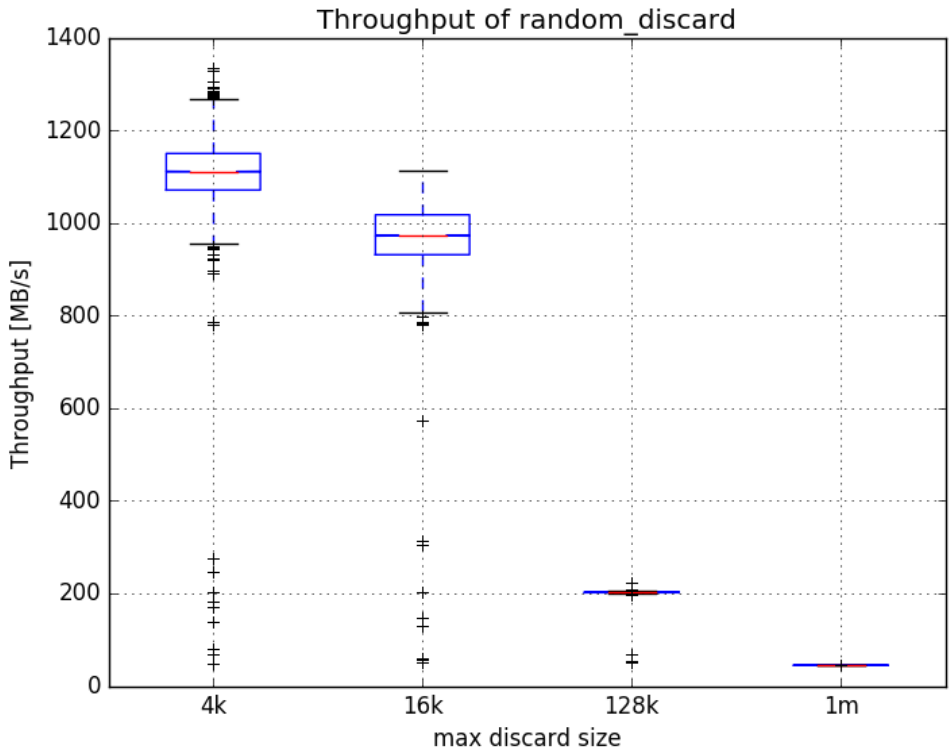


Figure 5.7: Testing of VDO volume created with various maxDiscardSize parameters. This test was conducted on a preallocated VDO without any data written

## 5.7 Write policies

Write policies are an important aspect of working with VDO.

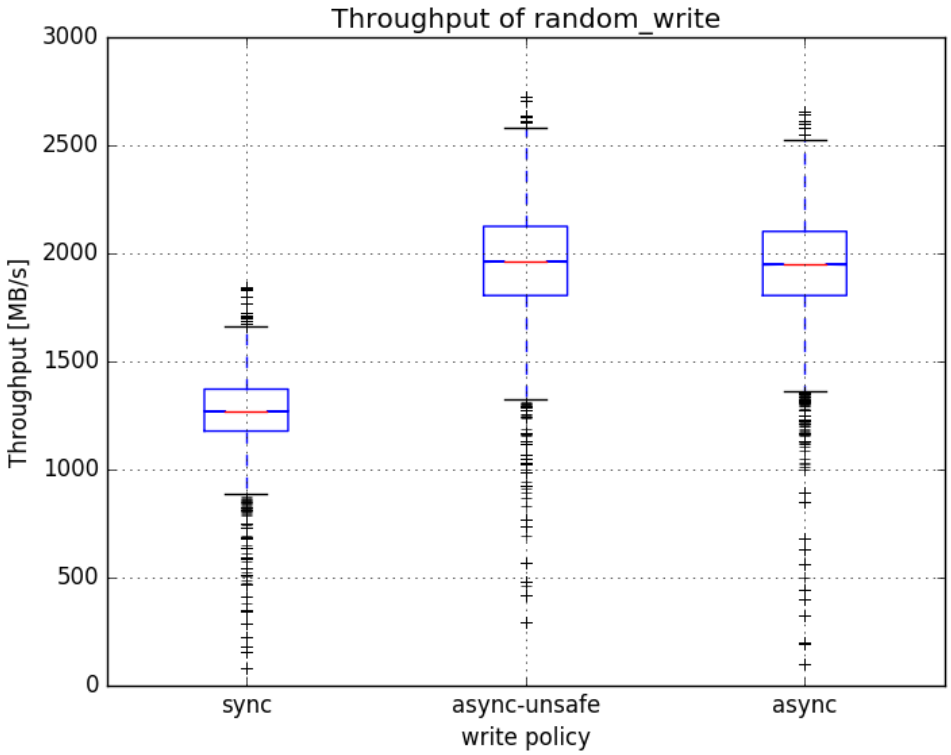


Figure 5.8: Testing of VDO volume created with various write policies

## 5.8 Testing on different architectures

## 5.9 Journal performance

Recovery journal is an important aspect of VDO for assuring safety of user data. It is physically present in the last slabs of VDO storage, in the metadata part. Updating journal might not be an expensive operation, if the supporting device under VDO volume is fast enough. However, with increasing number of VDO users, use cases where VDO will be placed on a badly performing devices can emerge.

This experimend is aimed at exploring a use case where writing to a journal can be a bottleneck for the VDO performance.

The test was conducted using slow rotational hard drive on a Machine 2. Performance of this drive is very poor and installing VDO on top of it increases the performance. However, it would be interesting to observe if journaling is posing as a bottleneck and the VDO volume could be even faster.

VDO stores its metadata on the last slab (or multiple last slabs) of the physical device. By having the last slabs be actually on a completely different, fast device could show how much speeding the journal increases the overall performance of VDO.

We will start by creating a partition on the rotational drive. After putting this partition and the fast SSD into one volume group, we will create an LV on the partition and grow it, so the last blocks are allocated from the second device. Finally, VDO can be put on top of this configuration and the test can be conducted.

To make absolutely sure only the journal updates went to the fast device and not actual data, the test was run using seekwatcher, to examine block seeks. As can be seen on charts, data were kept strictly on the slow device part of the LV and only journal updates were handled by fast device.

## 5.10 Testing with distributed file system

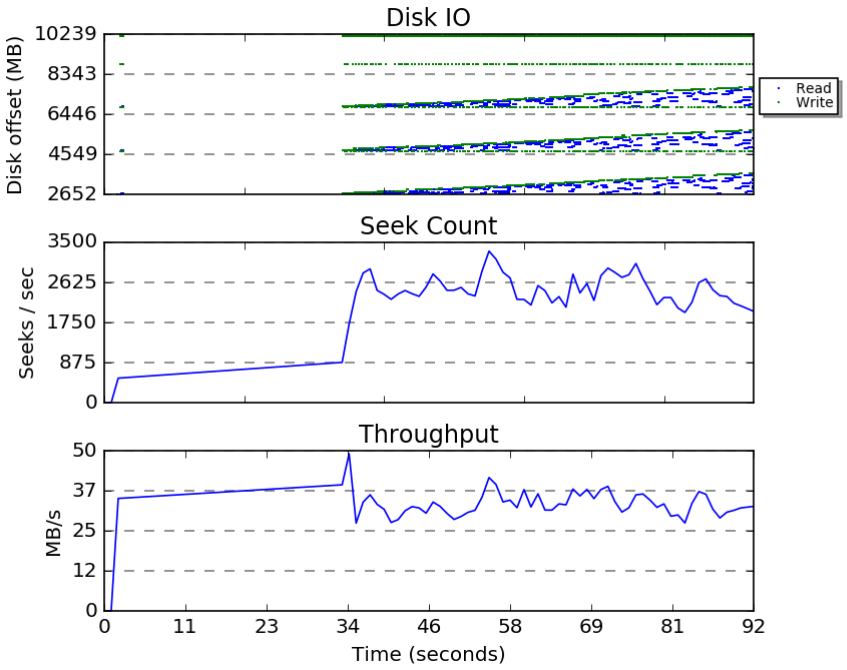


Figure 5.9: seekwatcher

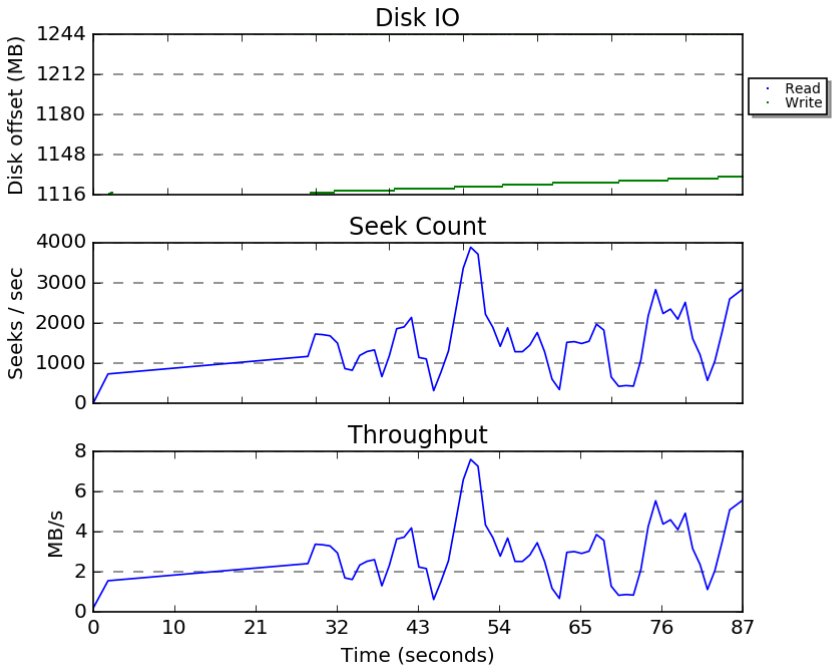




Figure 5.10: Seekwatcher

# 6 Conclusion

## 6.1 Future work

# 7 Appendix

## 7.1 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 |