Masaryk University
Faculty of Informatics

Performance testing of Virtual Data Optimizer storage layer

Master's Thesis

Bc.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.

Bc.Samuel Petrovič

Advisor: Adam Rambousek

Acknowledgements

I would like to express my deepest gratitude to Tea Dorminy for excellent and constant aid he provided during the time of writing this thesis.

I'm also thankful to my amazing fiance for her loving support and care.

Abstract

Abstrakt sa pise nakoniec

Keywords

VDO, deduplication, compression, storage, fs-drift, virtualization, LVM, Red, Hat

Contents

List of Tables

List of Figures

1 Introduction

Storage devices are becoming cheaper, but the storage requirements of the modern IT industry are growing faster, even exponentially. Innovation in storage utilization has compelling cost reduction potential, but many data reduction solutions are proprietary to a single company.

Linux, however, provides a framework for storage administration called *Logical Volume Manager* (LVM), providing users with easy means of managing storage. By using LVM in order to utilize storage optimization drivers available in the Linux Kernel, administrators can reduce their storage costs without having to use a proprietary, difficult-to-use solution.

The Linux Kernel provides several storage optimization drivers. Using LVM, these can be composed into a complex solution tailored to one's individual workload, combining them into a *storage stack*.

For instance, *thin provisioning* allows creating a virtual device larger than existing physical storage, reducing costs by delaying storage purchases until additional storage is actually needed for new data. Other layers include encryption, software RAID, caching, or backup/snapshot solutions.

One of the most exciting new projects in the storage optimization area is the *Virtual Data Optimizer* (VDO) driver. This layer uses deduplication (eliminating multiple copies of the same data) and compression (storing data in less space, if possible) to reduce the amount of storage required. Together, these techniques are called *data reduction*. While this driver provides thin provisioning like a thin-pool driver, using deduplication and compression means VDO can actually hold more data (if the data reduction is successful) than the storage devices.

Adding a data reduction solution to the storage stack is interesting since the cost of adding a new driver to a storage stack is much lower than the cost of purchasing of more physical storage. However, it does come with a cost of other resources such as memory or CPU, as performing deduplication or compression requires data processing that would otherwise not be needed.

Previous data reduction solutions have not seen wide adoption due to high costs relative to the storage savings, so VDO is carefully ar-

chitected to reduce the costs and provide tunables in order to optimize for specific workloads.

Poor VDO tuning can eliminate the potential cost savings. Therefore a performance testing is a crucial element for both VDO developers and administrators of storage stacks using VDO. VDO users and VDO developers all need to ensure that the resulting performance is maximized.

However, performance testing of such a complex technology requires extended knowledge of VDO's internal structures and expertise in benchmarking.

This thesis aims to lay a foundation for fast, efficient performance tuning of VDO: describing its workings, performance related structures and issues, and describing ways of benchmarking of VDO and the effects of the most important tunables.

Chapter 2 is an introduction of VDO technology, providing an in-depth explanation of its terminology, device organization, system requirements, and relation to other layers in a storage stack.

Chapter 3 presents a benchmarking tool `fs-drift`. Several new features were implemented in `fs-drift`in order to test VDO more efficiently.

Chapter 4 describes testing workflow design, methodology of measurements and data processing.

Chapter 5 is focused on a performance testing of VDO, including testing of different VDO components as well as more complex deployment cases. Results demonstrate maximizing VDO performance in different storage stacks, providing an example of testing and potential benefits from tuning VDO to a user's own hardware and workload.

In Chapter 6, high-level insight into 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 virtualization service in Linux storage stack. VDO enables users to operate with greater logical volume than is physically available. This is achieved by using deduplication, compression and elimination of zero-blocks.

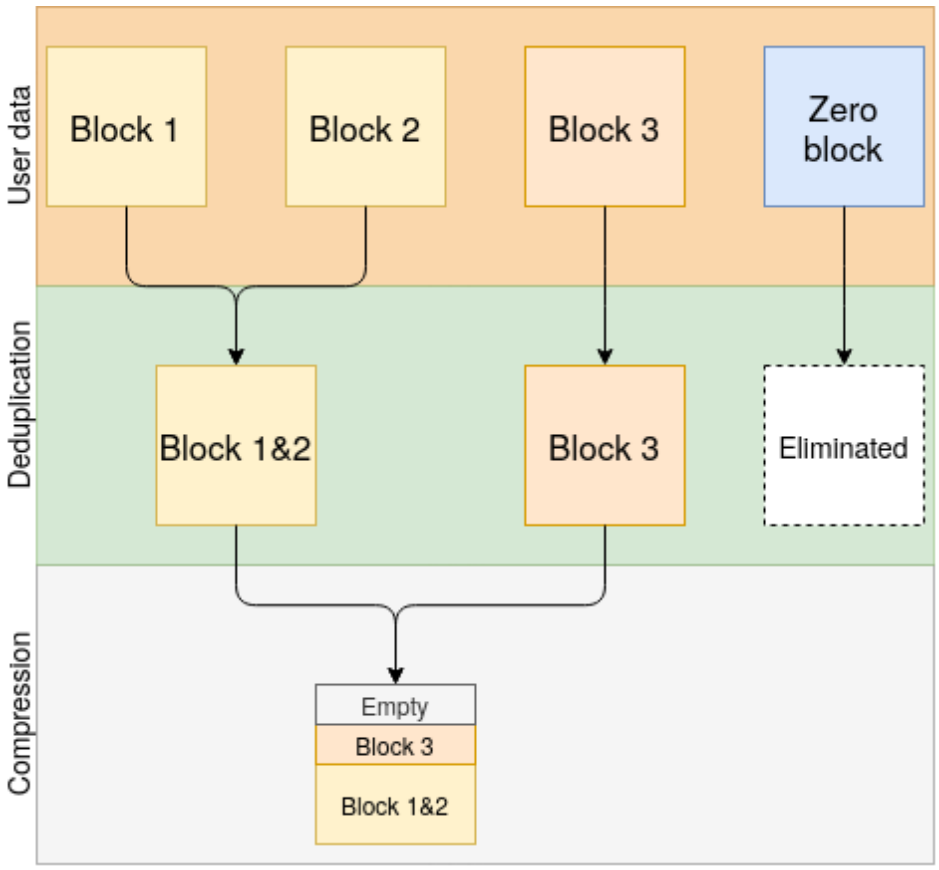


Figure 2.1: Space saving methods of VDO. In the first step, duplicate data and zero blocks are eliminated. In the second step, remaining unique blocks are compressed and stored as a single block.

Deduplication is a technique that, on a block level, disallows 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 incoming 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 identifies 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 optimization techniques is compression. By compressing already deduplicated blocks, VDO provides one more step to increase utilization of underlying device. Compression is also important for saving space in case the incoming data are not well deduplicable.

VDO uses standard LZ4 algorithm, for its compression, as it offers a good balance of speed and compression ratio. Compression work is distributed across the same threads which calculate hashes. Multiple compressed blocks are stored in a single physical block on a device.

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 underlying 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, using a block device for its storage and presenting a block device for upper layers. On creation of VDO volume, management tool divides the space into a region for UDS index, regions for VDO metadata, and regions for storing 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 cannot 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.

Slab size does not affect VDO performance, but VDO's per-slab metadata is of fixed size, so VDO has slightly more metadata for a given physical size with smaller slabs. VDO also requires several gigabytes for the UDS index, even on a small physical storage.

VDO module keeps two kinds of metadata which differ in the scale of required space. [1]

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, the term Physical size stands for overall size of underlying 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 supports 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 data sets 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 relevant 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.

2.3 Internal supporting structures

Working VDO instance contains supporting structures that handle the incoming events. Understanding their purpose as function is integral to proper performance tuning.

2.3.1 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 both synchronous 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 acknowledgment 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 committed 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 that's in progress which makes the commit queue. Every successful commit will wake others waiting after it is completed.

2.3.2 Block map

Block map is a structure used by VDO to handle logical to physical block mapping. It is implemented as a radix tree and works on a granularity of pages with every page holding mapping of 812 logical blocks.

The full block map is stored on a physical device, in one of the last slabs reserved for metadata and it is a cause of one of the requirements on physical space. It usually consumes about 1.25 GB per 1 TB of stored physical data.

Since block mappings are accessed with high frequency and reading from physical device could be costly, part of the block map is stored in memory in a block map cache. When processing incoming request, the relevant page is pulled from the cache, or in case the cache doesn't contain it, it is read from physical device and pushed into the cache. Such cache misses could be costly in terms of performance, so it is important to set the size of a block map cache correctly, so it can be able to hold mappings for all the data being read or written in a short time period.

2.3.3 UDS index

VDO uses a high performance UDS index for data reduction. UDS index is a structure designed specifically to identify duplicate data using hash fingerprints of data blocks. It exists in VDO to provide deduplication advice for effective deduplication. [2].

VDO computes hashes of incoming blocks and checks them against an index, to retrieve potential matches. Any newly computed hash is placed at the beginning of the index.

The index itself consists of two parts. One part is stored in memory and the second part is stored on the physical device in slabs reserved for metadata. The part in memory is so called *deduplicationwindow*, which contains fingerprints of the most recent blocks VD processed.

Instance of UDS index is not vital for VDO block handling. In case of losing UDS index, 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 building a new one.

Updating index is costly, so VDO is trying to minimize the updates. That is a reason the index can contain references to blocks that are no longer present in the data.

2.4 Tunables

VDO provides user with many means of tuning. Tuning consist of parallelizing workload to more processor by changing number of different threads, choosing the right block map cache size, write policy or discard size.

2.4.1 VDO threads

One of the main means of tuning VDO performance is changing number of VDO threads that are completing various tasks. While running a VDO volume, users should monitor the thread usage and tune it accordingly. In case there is a high thread usage (>50 %), users should increase the number of relevant threads. There are six tunable threads in VDO. Figure 2.2 displays relationship between various threads and VDO internal structures.

Logical zones thread

Logical space presented to users of VDO device consists of Logical Block Numbers (LBNs).

LBNs are contained within pages, which are the main unit a block map cache is working with. Pages are further grouped into zones. Zones are assigned to logical zone threads, such as workload on multiple zones can be managed in parallel

Logical zone threads are active during read and write requests, since they are translating LBNs to PBNs.

Physical zones thread

Physical space VDO is working with consists of Physical Block Numbers (PBNs).

PBNs are divided into larger sections called slabs. Every slab is divided into zones. Physical zone threads are processing requests to physical zones in parallel.

Physical zones threads are active only during write phase, because their purpose is to update reference count. vdoPhysicalThreads is the option for setting physical threads.

I/O submission threads

I/O submission threads are submitting block I/O (bio) operations from VDO to the underlying physical device by passing requests from other VDO threads to the driver of the physical device.

The number of I/O submission threads can be tuned using vdoBioThreads option.

CPU and Hash zone threads

CPU threads and Hash zone threads in VDO help manage and balance intensive computing workload to multiple cores. The intensive operations such as computing hashes or compression are handled by these threads.

The number of CPU and Hash zone threads can be tuned using vdoCpuThreads and vdoHashZoneThreads options respectively.

I/O acknowledgment threads

This type of thread is managing acknowledgment operations to an application above VDO after I/O request completion.

The number of acknowledgment threads can be tuned using vdoAck-Threads option.

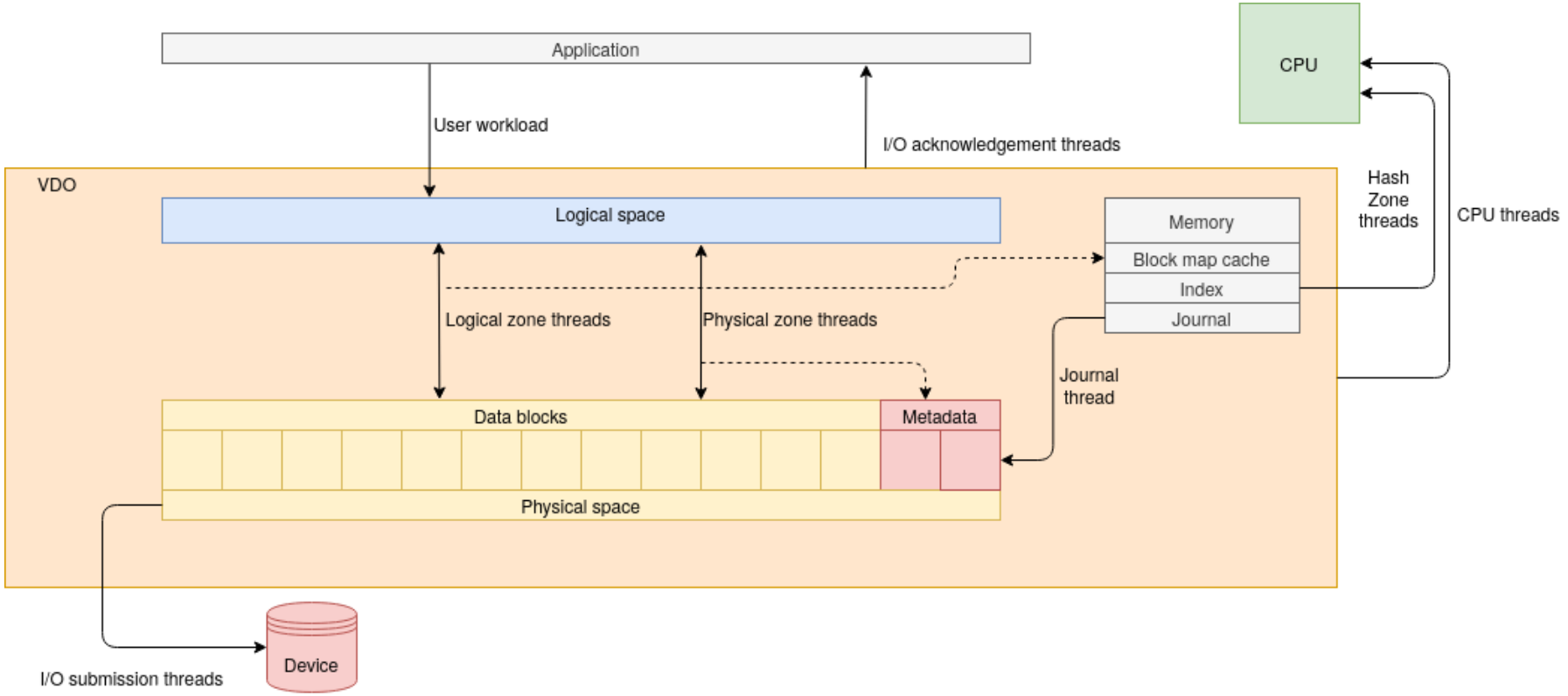


Figure 2.2: VDO internal structures and their relationship with VDO threads.

2.4.2 VDO write policies

VDO can operate in either synchronous or asynchronous mode.[3] 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 whether 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 acknowledgment, it attempts to deduplicate 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 is 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.

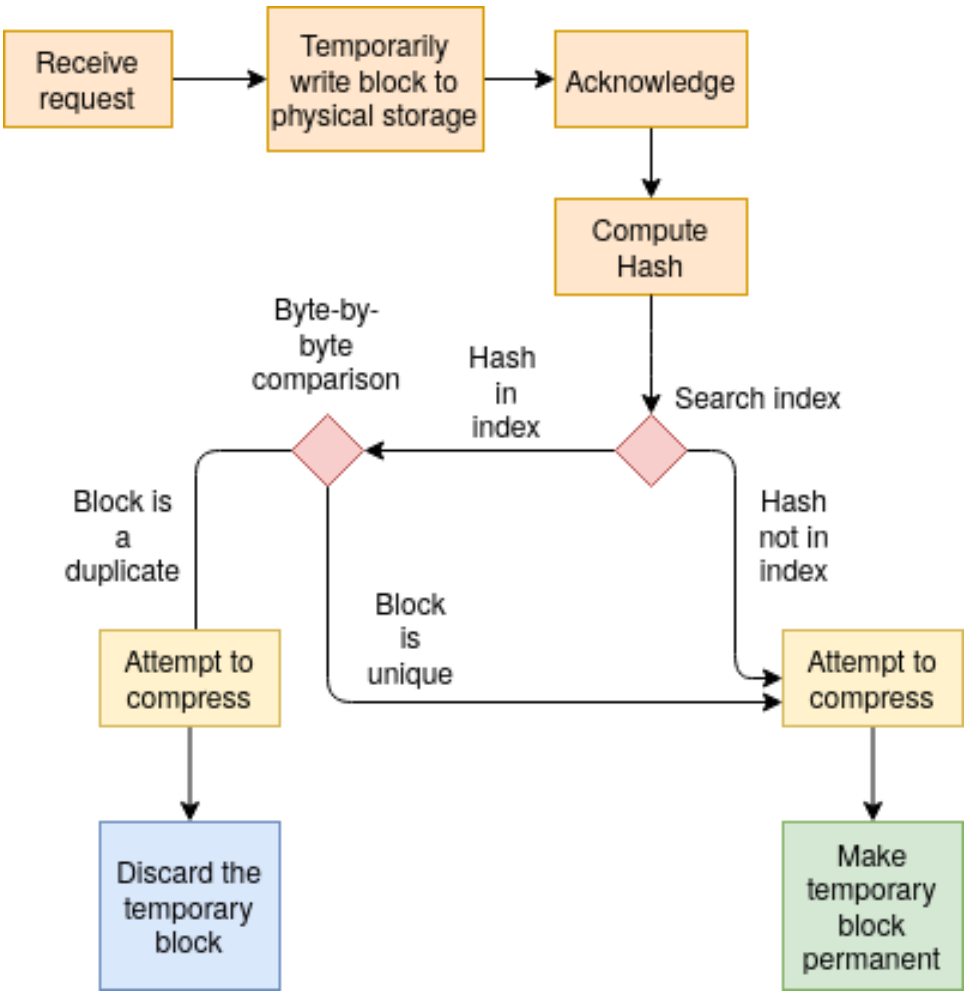


Figure 2.3: Flowchart of asynchronous diagram processing

Asynchronous mode

In asynchronous mode, instead of writing the data immediately, physical block is only allocated and acknowledgment 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.

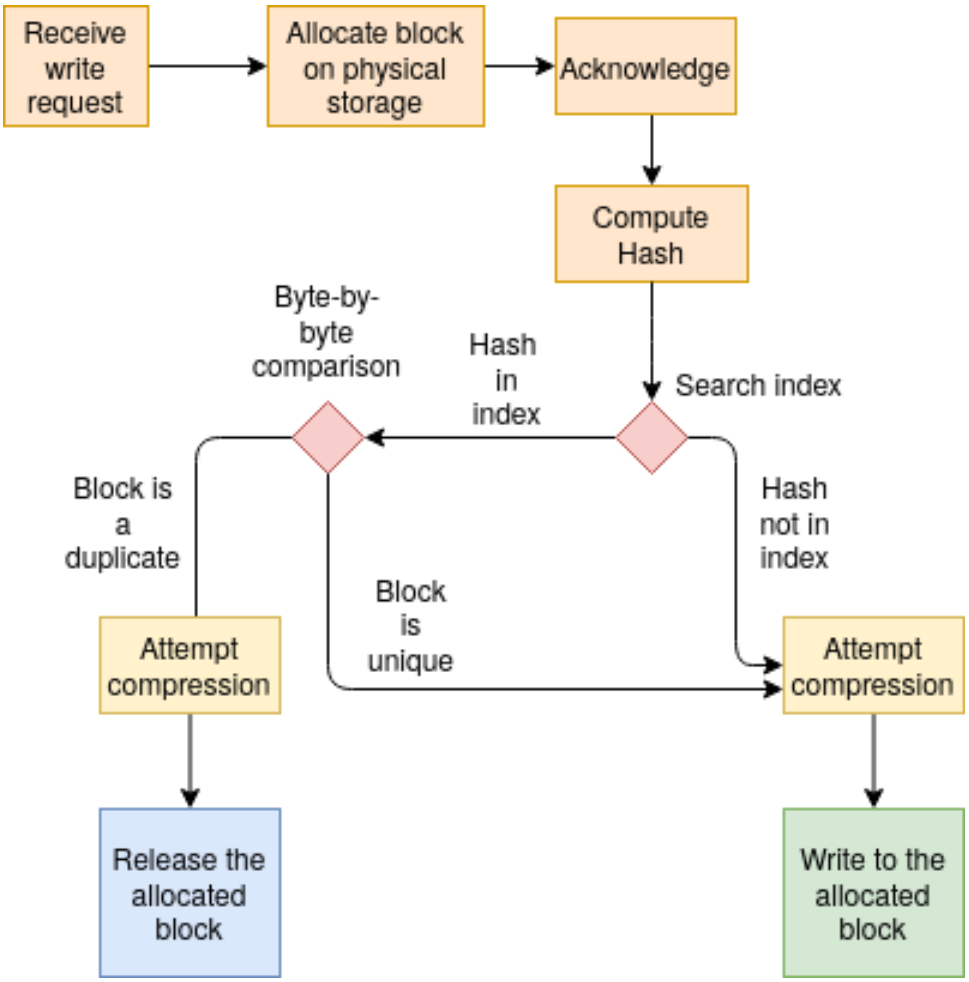


Figure 2.4: Flowchart of asynchronous diagram processing

Asynchronous unsafe mode

In older versions of VDO, asynchronous write mode was not compliant with ACID [4] guarantees. In case an application attempted to asynchronously write data into a given block while the system failed, the block's contents could the block's contents could be permanently lost under certain conditions, violating the atomicity element of ACID. Most of the applications expect the underlying devices to be ACID compliant and can end up corrupted after a recovery.

For this reason, asynchronous mode was re-engineered to be ACID-compliant, however, old implementation is being kept in VDO under the name async-unsafe. [5]

2.4.3 Block map cache size

Block map exists in VDO volume to maintain the mapping of logical blocks to physical 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. When a request to read or write block from VDO occurs, the block mapping for that block must be read or updated. If the mapping is not already in the block map cache, the cache must load the relevant block mapping page, possibly at the expense of writing out a previously-modified block mapping page currently in the cache.

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

The 128 MB can cover about 100 GB of logical blocks. However if VDO logical space is larger than 100 GB and the workload is so random that the requests doesn't hit cached pages, user can observe great performance hit. In case the block map cache is full, therefore runs out of free pages and a request comes for a page that's 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 preconditions for cache tiring are met.

3 Fs-drift

Fs-drift is an open-source benchmark developed specifically for testing heavy workload and aging performance. Being implemented in Python 3, it is very easy for users or contributors to add new features or change the benchmark behavior to their needs.

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 Compressible data generator

While testing compression and deduplication technology as VDO, data generated for testing need to be specifically shaped. The benchmark needs to be able to generate the testing data in a way that corresponds to examined cases. Since VDO performs deduplication and zero block elimination, testing data must be generated so that each block is properly compressible on itself rather than just padding the whole size with zeroes. For this reason, new parameter was added to fs-drift, that enables the user to produce buffers with specified compressibility.

Compressible blocks are achieved by filling the desired number of block's bytes with random data and filling the rest with zeroes. This way, any generated block is always compressible in desired manner.

Performance of this step is very important, since the workload needs to be able to output enough data in a given time to properly benchmark the target.

Usage:

- -c|–compression-ratio, number that is a desired compress ratio, e.g. 4.0 is a compressibility of 75%, so the compressed block occupies 25% of the original space (default 0.0)

3.2 Deduplication

Sometimes it is needed to produce deduplicable data for correct workload shaping. Fs-drift provides this option to users by repeating gen-

erated random blocks. Outputed data is therefore deduplicable to a specified amount.

Usage:

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

3.3 Direct IO

When using fs-drift to measure performance of random operations, the Linux page cache can heavily skew measurements.

Writes of data to the page cache can be very fast since the requests are processed in memory to be submitted to the device later. Read performance is also affected if the desired block is found in the cache.

There are two ways to combat this effect. First is calling fsync() after writing some amount of data, which only works for write operation. Second option is to use O_DIRECT flag when opening target to signal the system to read and write while bypassing the page cache.

Users can specify to fs-drift if they want to use direct IO by a parameter direct.

Usage:

- -D|–direct, if 1, use os.O_DIRECT flag when opening files. Data alignment of 4096 bytes will be used. (default 0)

3.4 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 measurements. 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 '')

3.5 Random discard operation

Because more and more devices have an internal block mapping layer, such as thinly provisioned devices or hardware devices, sending devices discard requests for deleted data locations is becoming more common. Many systems run 'fstrim' on a schedule, while others mount file systems with the discard option which sends discards when data is deleted. Because of this, discard performance i becoming increasingly important.

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.

However, discard operations are very fast (about 1 GB/s, Figure5.11) and the speed of invoking blkdiscard through the subprocess module was relatively slow (about 500 calls second), so for discarding less than a megabyte at a time, the overhead of invoking blkdiscard took longer than the actual discard.

However, Python provides a framework to execute ioctl calls 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 shows exactly how to issue block discard ioctl.

If the users of fs-drift wants to test discard speed, they 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 instead of producing buffer and writing data, it's using discards with random offset and specified block size to call BLKDISCARD.

Example 3.1: Using BLKDISCARD ioctl to discard first 4096 B of a device /dev/sde

```
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)
```

3.6 Random map

When computing offset for random operations, it might not enough to just generate random number. Sometimes it is beneficial 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 offsets 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)

3.7 Multithread

Multithreaded workloads are essential for researching high-performing technology.

Option to run fs-drift a multi threaded benchmark 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 re-engineered so the threads are not corrupting each other 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.

If multi threaded fs-drift is used for testing, user should limit the number of parallel IOs using iodepth parameter.

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

3.8 IOdepth

Fs-drift threads will submit their IO requests to the device or files in parallel. By default, this behavior is unmanaged and can result in overloading the target.

By using iodepth parameter, user can specify maximum amount of 4 kB IO units that will be in progress at the same time. This behavior is implemented by managing a counter shared between threads. The counter contains information on how many IOs are in progress. If this number is higher or the same as the specified iodepth, threads will wait randomly between one to ten miliseconds for the counter to be lowered.

- -i|–iodepth, number of 4 kB blocks that can be executed at the same time (default 0, unmanaged)

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 measurement, 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 inconsistent, 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 data points, 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 measured 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. [6] 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 preferred to ensure no performance impacts caused OS aging, memory shortage, etc.

For this thesis, testing was conducted on instances of RHEL-8.1 [7] and RHEL-8.2 [8] 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 sequence of LVM commands. For the simplest 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 device, 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, the framework calls sync and drop caches.

4.1.1 VDO allocation

Reproducibility of results is an important aspect of any kind of testing. As mentioned, new test environment is prepared before every test to achieve stable, reproducible results. However, when VDO instance just started, it's mapping information is not fully allocated yet. VDO stores its mapping information in a tree, which is allocated as needed.

This could pose a problem for performance testing, since the allocation takes some to complete and therefore some amount of testing time will be testing the allocation latency. We can observe this effect

on a Figure 4.1. This test was conducted on a empty instance of VDO on top of HDD device. As could be observed, it took about 150 s for VDO to reach stable performance.

In case we don't want to specifically test VDO allocation latency, by testing on unallocated VDO a large of testing time is effectively wasted. This could be avoided by forcing VDO to allocate all of its mapping tree before the test. VDO logical space is divided into regions of 812 4k blocks. Every region is covered by the same set of mapping blocks. Therefore executing a write request to each region will force VDO to allocate the whole needed mapping tree.

We can achieve the allocation f.e. by writing zero byte every 812*4096 bytes through all the logical space before test. Figure 4.2 shows performance of VDO measured after executing the preallocatig sequence. We can observe, that the performance is stabilized from the start of the test unlike the previous test.

This technique will be by default used prior to all further tests, unless explicitly stated otherwise. The operation is a component of the testing package and can be controlled by parameter -a.

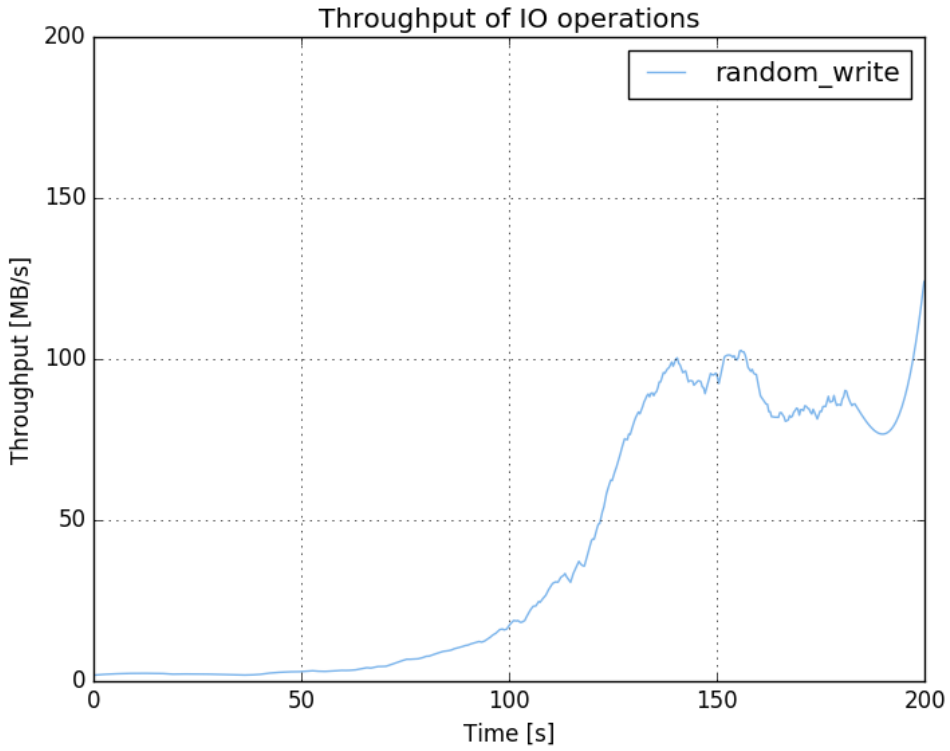


Figure 4.1: Performance of allocated VDO storage in time. After all mapping space is allocated, the performance stabilizes.

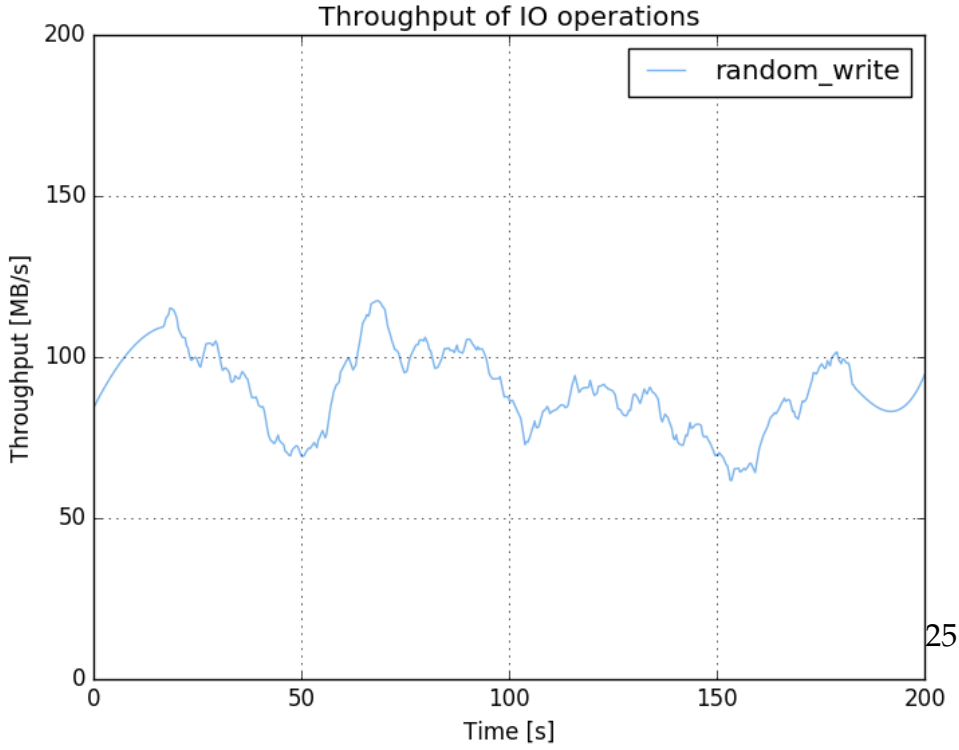


Figure 4.2: Performance of empty, but previously allocated VDO storage. Measurements are stable through the whole test.

4.2 Benchmark settings

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

4.2.1 IO operations

Fs-drift offers wide range of IO operations to be executed against a target. Most of the operations come in sequential and random variant.

Sequential operations are guided by an offset variable shared between threads, so that they can cooperate in processing the target. Random operations are either getting a random offset, or request one from the random map, so no overwrites can happen during the test. On top of that, random operations can wait if the target is overloaded.

When testing VDO, the most interesting behavior is triggered when something changes within the VDO block mapping tree. This is the reason, mainly write and occasionally discard operations are used in the tests conducted for this thesis. Randomized operations are used to mimic real-life access patterns

4.2.2 File size

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 target. Since some delay can occur between actually invoking the writing process, larger file size will mean more aggressive workload for the device or file system.

4.2.3 Block size

The way fs-drift works, block size is the smallest granularity of IOs that the workload achieves, 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 submissions, the device that services the requests can have a small bit of time to process the requests. By choosing larger block size the workload becomes more aggressive, since there is more data submitted at the same time.

However, by increasing block size, we're increasing the smallest continuous chunk of data that will be worked with, so a random workload may become slightly more sequential.

Blocksize is a granularity at which fs-drift collect data, one record for one administered IO, so setting it too high can mean lowering the data points in results.

4.2.4 Compression ratio

Compression ratio is an important factor in testing VDO. Generally, the test should use some compression, so the VDO can exercise 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.

4.2.5 Deduplication

Similar to compression, it is expected that user data will be to some degree deduplicable and the testing workload should reflect that. Fs-drift is producing deduplicable data on block size granularity.

Some of the tests that will try to mimic real-life usage should increase dedupulication properties of their workload. In case the test is meant to be of a higher intensity, deduplication should be decreased, so the VDO needs to handle more data physically.

4.2.6 Random map

Most testing workloads that aim to test limits of VDO technology are random writes. This also exercises the ability of VDO to serialize

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 bet the best solution, 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 recommended to use randommap.

4.2.7 Multithreading

VDO is a high performing layer aimed to serve intensive, multi threaded applications. As a block device, VDO serves incoming IO stream through a queue of 2000 IOs. In the test aimed to exercise upper limits of VDO performance, the generated workloads need to be intensive enough so the full potential of VDO can be achieved.

To generate such workload, we need to use multi threaded fs-drift with batch submission controlled by iodepth. Iodepth in fs-drift controls exactly how many 4 kB blocks will be in progress at any given time. It is not only lower, but also upper limit of the batch size.

By setting these parameters correctly, we aim at the right VDO queue exercises. If the VDO isn't engaged enough, the test is limiting itself in the performance. On the other hand, if the test has no upper limit on IO submission, the queue would be overflown at all times, which will slow down performance.

Following formula references a way to compute maximal batch size with given number of threads and block size.

$$threads * (blocksize\ 4) = Maximalbatchsize$$

In case the maximal batch size of the test with given parameters is much larger than 2000 (size of VDO queue), the test will overwhelm VDO and measured performance would be too low. In case the maximal batch size of a test is much smaller than 2000, the test will not exercise VDO fully and the performance would also appear low.

Fs-drift parameter iodepth limits the batch size from both size. If the user specifies iodepth of 2000, fs-drift will be submitting exactly 2000 4 kB blocks at the same time.

However, even limiting batch size with iodepth isn't a final solution. In case the physical device under VDO is very slow, VDO might not

be able to empty the queue between batches which would again cause delays and worse performance. Testers should always make sure the queue was properly exercised by inspecting vdostats during and after the tests.

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 automized 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 manageable 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. However, since starting phase of fs-drift may take considerable amount of time (allocation, computing random map), the gathering thread waits for fs-drift start file. This functionality is used mainly to gather statistics about VDO using vdostats, however other points of interest may be stored for later examination using this feature.

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. HTML code is generated by an external library. [9]

The HTML report consists of two parts. First part is presenting individual report for given results. The second part is comparing the inputted results in an easily digestible 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 attributes to plot
- lim_Y to set the upper limit of Y axis
- test_label, to label outputted 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 easily on this chart

4.4.2 Throughput progression chart

This chart is a simple representation of a measured 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[1] 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 hyperbolic manners. 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.

However, it is very useful, when there is an expected change in behavior of the targeted device.

1. Available in SciPy. [10]

4.4.3 Histograms

Histogram of measured throughput is used to evaluate possible deviations in the measurements that could not be observed on other charts. Caching effects are usually easily visible on histograms as well different behavior patterns of a tested target.

4.4.4 Boxplots

Box plots 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 equipped with. These machines are stable systems used by Red Hat Kernel Performance team for regular testing.

5 Performance of VDO

This chapter presents testing of different components of VDO and performance tuning of some possible user cases. Presented results were obtained using methodology from Chapter 4.

5.1 Performance of VDO after prolonged usage

VDO will try to serialize the the incoming workload if possible by sequentially filling slabs to about half of their capacity. If more than half of slab is used, or if VDO is running out of free physical blocks, it will start to search for free blocks in the slab, which will appear as a change in performance.

When VDO is empty, the performance can look as of performance of underlying device under sequential load. After using about 50% of the physical capacity, performance will lower and starts to resemble a performance of underlying device under random workload. Figures 5.1 and Figure 5.2 show change of access pattern of VDO to an underlying device.

We could test this effect by setting up fresh VDO volume, preallocating it and running random write workload without randommap. This means some of the blocks might be overwritten during the workload, which can free some physical blocks from the sequentially filled slab, that the VDO can search for.

While running this test, performance of VDO should be reasonably stable until it uses about half of its data blocks. Performance should sharply decrease after that point.

On Figure5.1 and Figure5.2, results from such test can be examined. Physical space for this test was set to 5 GB with slab size of 2 GB, therefore this instance of VDO has one slab of data blocks. We can observe on the Figure that the workload caused the VDO to use half of its data blocks in about 12 s. While looking at the Throughput graph, we can see that is the turning point for performance decrease.[1]

1. Figures were created using block trace visualisator Seekwatcher [11]

Evolution of performance can be observed on a Figure 5.3. It can be clearly seen, that the performance started to drop at approximately 150 s mark. This test was conducted on a VDO with physical volume of 12 GB. Space for the user data has 4 slabs, which is 8 GB. By examining Figure 5.4, we can observe, that after 150 s,the test passed half of VDO's physical volume.

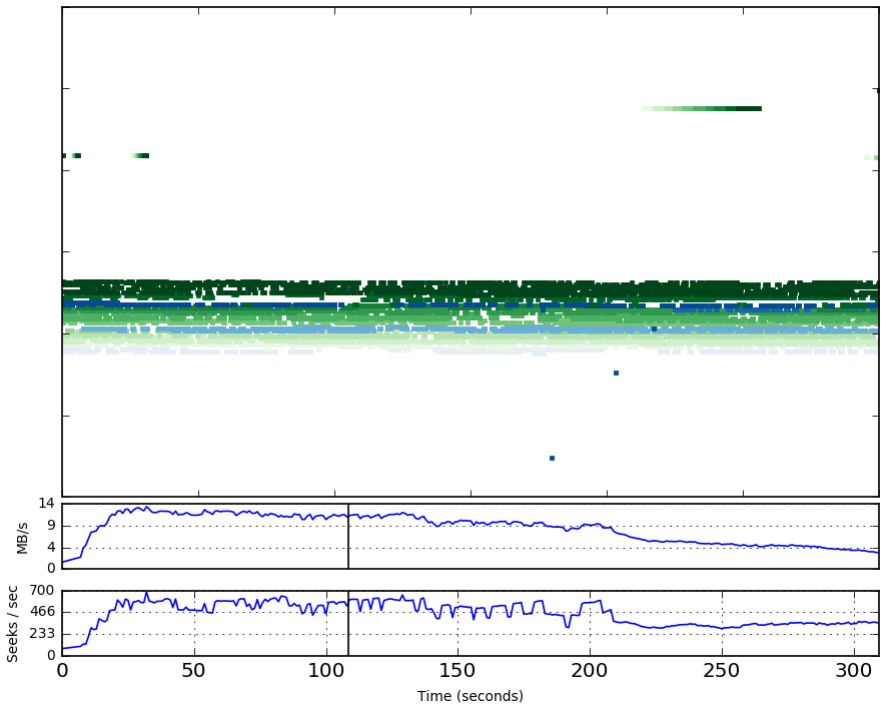


Figure 5.1: Access pattern of VDO to the underlying device while VDO is empty. VDO is successfully serializing incoming workload.

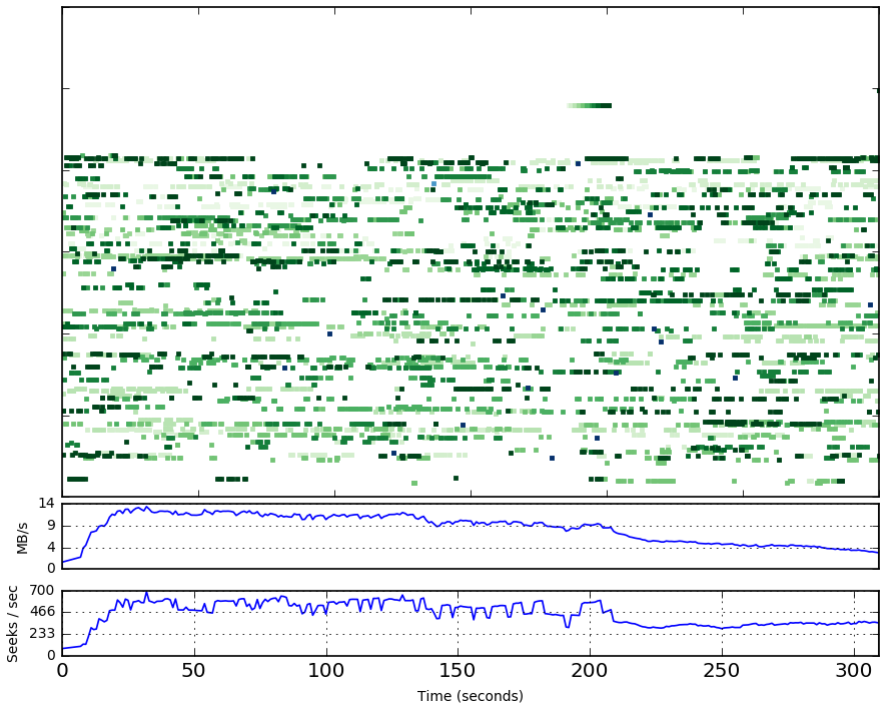


Figure 5.2: Access pattern of VDO to the underlying device while VDO is almost full. VDO can't serialise incomming workload anymore.

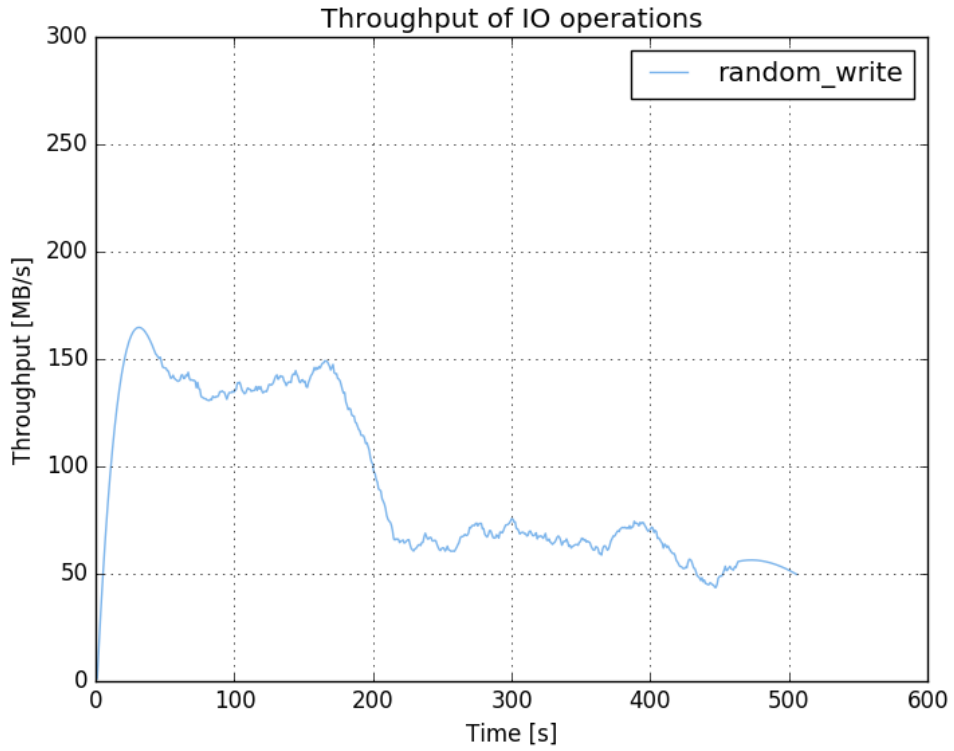


Figure 5.3: Evolution of VDO performance while filling the medium. The performance decreases after writing 50% of data blocks.

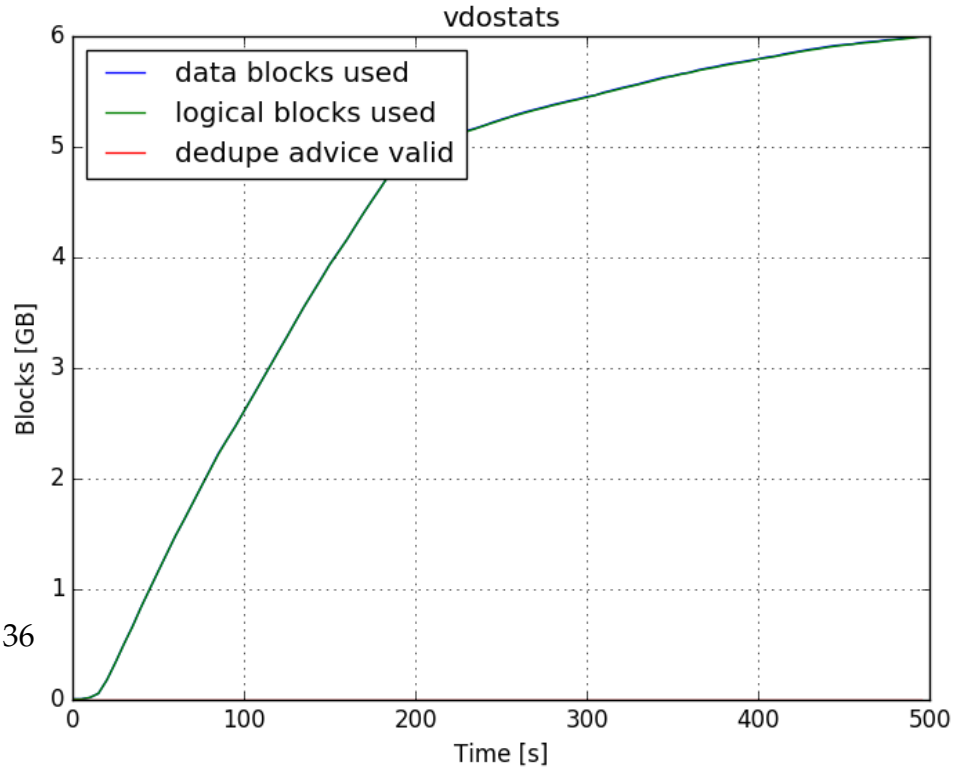


Figure 5.4: Evolution of logical, physical and deduplicated blocks while filling the medium.

5.1.1 Steady state testing

As shown in Sections 5.1 and 4.1.1 performance testing of VDO is susceptible to various testing preconditions. Most of the VDO instances created by users won't be unallocated nor will they be only half empty most of the time. To find out, how could VDO perform while being used in real-life, we should be able to create a testing instance that would show qualities of used VDO.

The aim of this test is to bring VDO to a hypothetical steady state where its partially fragmented and almost full. This can be done by dividing the testing to two stages. First stage will be aimed to prepare the VDO volume. Second stage will finally gather performance measurements.

The pre-writing should fulfill these conditions:

- fill up more than half of physical space
- use random write across the logical space so the content is cached
- using only non-compressible and non-deduplicable data

This test, based on principles of aging testing may equip users with efficient means to predict the average performance of VDO volumes,

5.2 VDO threads

Setting the correct number of VDO threads is the main method for users to increase performance. Incorrect amount of threads can result in unwanted performance penalty.

To show the effect of VDO threads tuning, we'll design a workload that could exercise VDO in a way it will benefit from thread count increase. The designed workload will mimic high-traffic, multi threaded usage. We'll use high thread count, large block size, random write workload without randommap, so the data can be randomly overwritten. To generate more traffic, occasional large discard will be triggered so the physical space becomes more fragmented.

We can see results from VDO threads testing below. The tests were conducted on Machine 2 with VDO installed on an SSD. Each test, number of threads was increased to show the optimal performance. The tests were conducted on VDO volumes with following settings of VDO threads:

Conducted tests:

- default: Logical: 1, Physical: 1, CPU: 2, Hash: 1, Ack: 1, Bio: 4
- 1: Logical: 1, Physical: 1, CPU: 1, Hash: 1, Ack: 1, Bio: 1
- 2: Logical: 2, Physical: 2, CPU: 2, Hash: 2, Ack: 2, Bio: 4
- 3: Logical: 4, Physical: 3, CPU: 3, Hash: 3, Ack: 4, Bio: 6
- 4: Logical: 6, Physical: 3, CPU: 4, Hash: 3, Ack: 6, Bio: 8
- 5: Logical: 8, Physical: 6, CPU: 6, Hash: 3, Ack: 6, Bio: 8

On Figure 5.5 it is apparent that the thread count is insufficient. Not only does the one Logical thread use up to 80% of CPU, usage of some other threads is nearing 40%. This is mainly Physical threads and CPU and Hash threads (The complete results can be found in electronic appendix).

Figure 5.6 it is observable the CPU usage of individual threads was lowered after increasing the threads number. We can examine the performance impact in Figure 5.7.

By examining the exact values in Table 5.1, we can see that the correct thread tuning improved throughput by approximately 100 MBps.

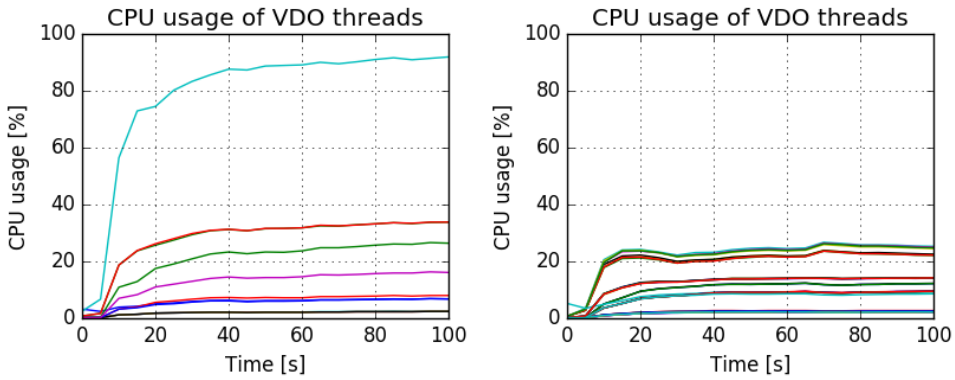


Figure 5.5: Thread load before increasing number of threads (Test: default)

Figure 5.6: Thread load after increasing number of threads (Test: 5)

5.3 Block map cache

When VDO receives a request, it needs to find mapping between the logical and physical address in the block map. First, it looks for the needed entry in the pages stored in block map cache. If it's not in the cache, VDO will retrieve the correct page from the part of block map that is stored on the disk and writes the page into the block map cache. If the request was a write request, page is updated only in memory and the change will be written to the on-disk part when VDO decides to discard the page from the cache.

VDO decides to discard pages from the cache either when it wasn't used for some time of if there is no space for a new page to be cached. If there is no space left in the cache, VDO needs to discard some page, writing it on the device and load the requested page. This round trip of two I/Os to the physical device is expensive and could cause a performance problem.

The default size of block map cache is 128 MB, which is 32768 pages, that covers about 100 GB worth of data. In case the logical space is larger than what could be managed by the block map cache and the access pattern of the workload is unpredictable enough, block map cache can easily run out of free pages and would need to write and load a page to and from the physical device on every request. That

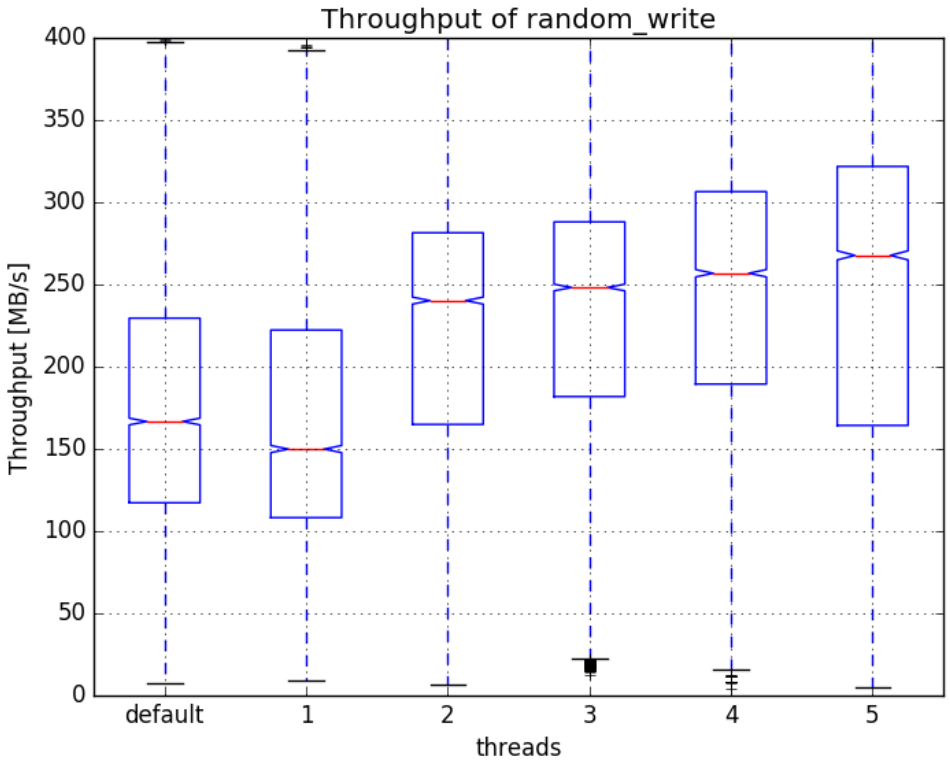


Figure 5.7: Testing of VDO volume created with increasing numbers of VDO threads

could be a bottleneck for VDO performance and is the reason why users can increase the size of a block map if they believe the standard size will not suffice.

To test this effect, an unpredictable, aggressive random write workload can be run on VDO instances with various sizes and various block map cache sizes.

The test results show performance of random write workload on three VDO instances. All parameters of VDO were kept the same except logical size, and block map cache size:

- Test1: logical size: 80 GB, block map cache size: 128 MB(default)
- Test2: logical size: 400 GB, block map cache size: 128 MB(default)
- Test3: logical size: 400 GB, block map cache size: 512 MB

Throughput of random write (MB/s)						
	median	1st q.	3rd q.	min	max	stdev
default	166.24	116.82	229.1	6.99	953.87	89
1	149.48	107.79	221.91	9.08	1130.06	94
2	239.78	164.48	281.05	6.48	1241.21	125
3	247.74	181.42	287.65	11.74	1366.39	125
4	256.41	189.02	306.05	4.12	1664.54	141
5	267.29	163.87	321.38	4.89	1400.84	153

Table 5.1: Results of performance tests of multiple tests with increasing number of load balancing VDO threads

Throughput of random write (MB/s)						
	median	1th q.	3rd q.	min	max	stdev
80g	67.32	46.16	151.54	10.57	188.82	55
400g defaul	1.13	0.94	1.32	0.85	35.98	1
400g increased cache	74.66	58.18	174.05	12.69	187.58	55

Table 5.2: Testing of VDO volume created with various with sufficient and insufficient block map

The expected results will be that on the VDO volume with insufficient cache size, performance will be significantly worse than on other two tests. The test with increased cache size will demonstrate, that the standard performance is restored to the baseline levels.

We can observe the comparison of performance on Figure 5.8 and Table 5.2 for exact numbers.

5.4 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

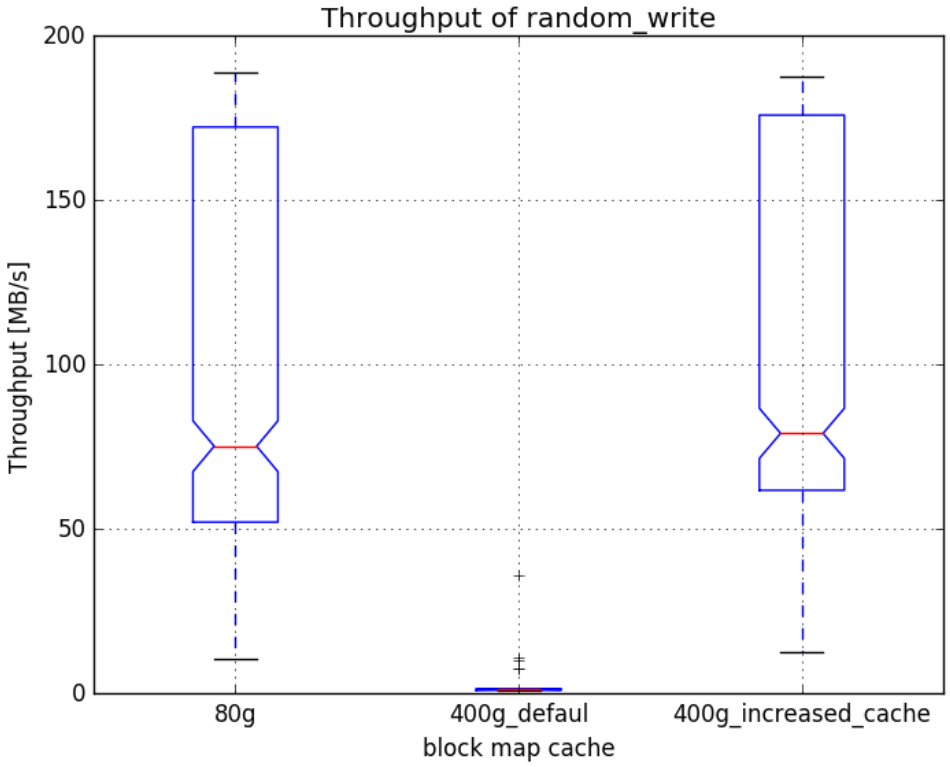


Figure 5.8: Testing of VDO volume created with sufficient and insufficient block map cache size

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 various –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 progression 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 without pre-writing the 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 that 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 **??**.

It is apparent, the results from empty, but previously allocated VDO show the same behavior as the results from the test where VDO was filled with random workload at the beginning, unlike the VDO without allocation, that shows heightened performance.

Throughput of random discard (MB/s) on full VDO						
	median	1st q.	3rd q.	min	max	stdev
4k	1077.65	1025.58	1119.79	158.12	1310.17	123
16k	935.4	863.6	966.42	140.93	1069.8	120
128k	184.27	179.58	185.94	105.26	195.25	9
1m	42.4	42.27	42.57	38.79	43.11	0
Throughput of random discard (MB/s) on empty but fully allocated VDO						
	median	1st q.	3rd q.	min	max	stdev
4k	1114.18	1075.4	1152.95	48.4	1334.86	67
16k	977.27	935.99	1019.96	52.48	1111.36	62
128k	202.49	201.6	203.29	51.29	223.63	8
1m	44.87	44.75	45.03	44.31	46.67	0

Table 5.3: Performance of discard operation on a VDO volumes with various state of utilization

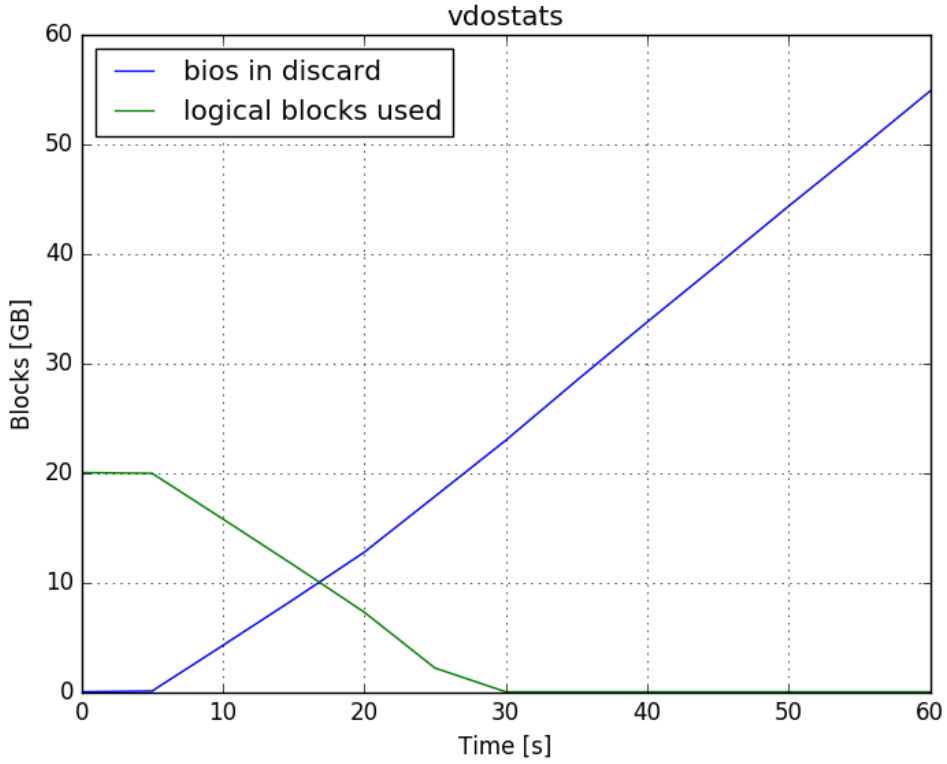


Figure 5.9: Evolution of VDO stats during random discard workload

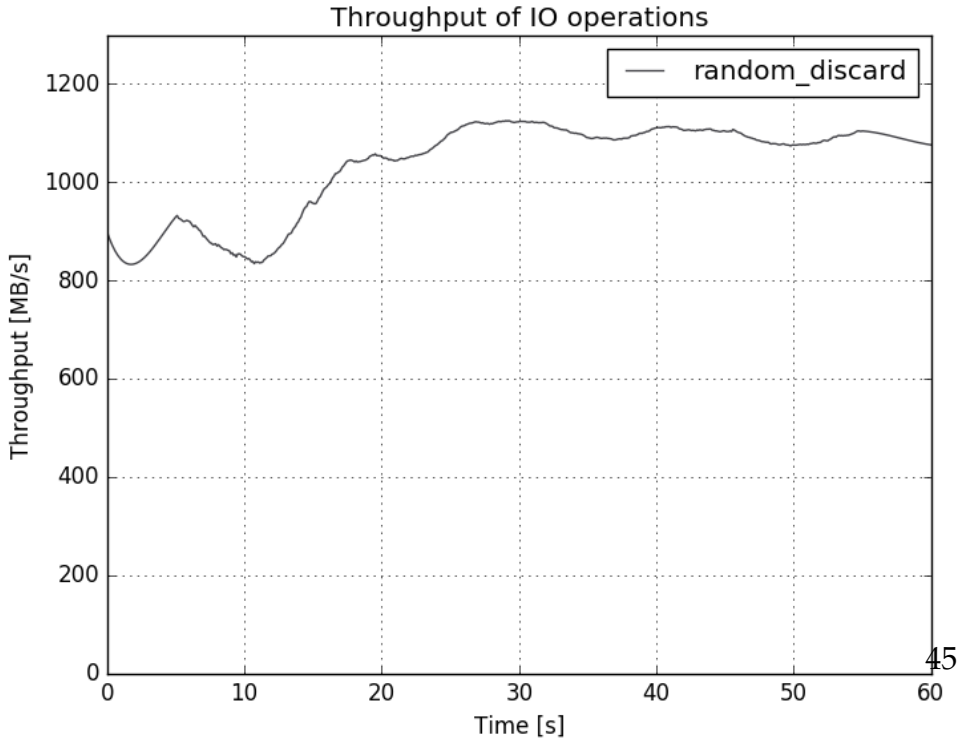


Figure 5.10: Evolution of throughput during random discard workload

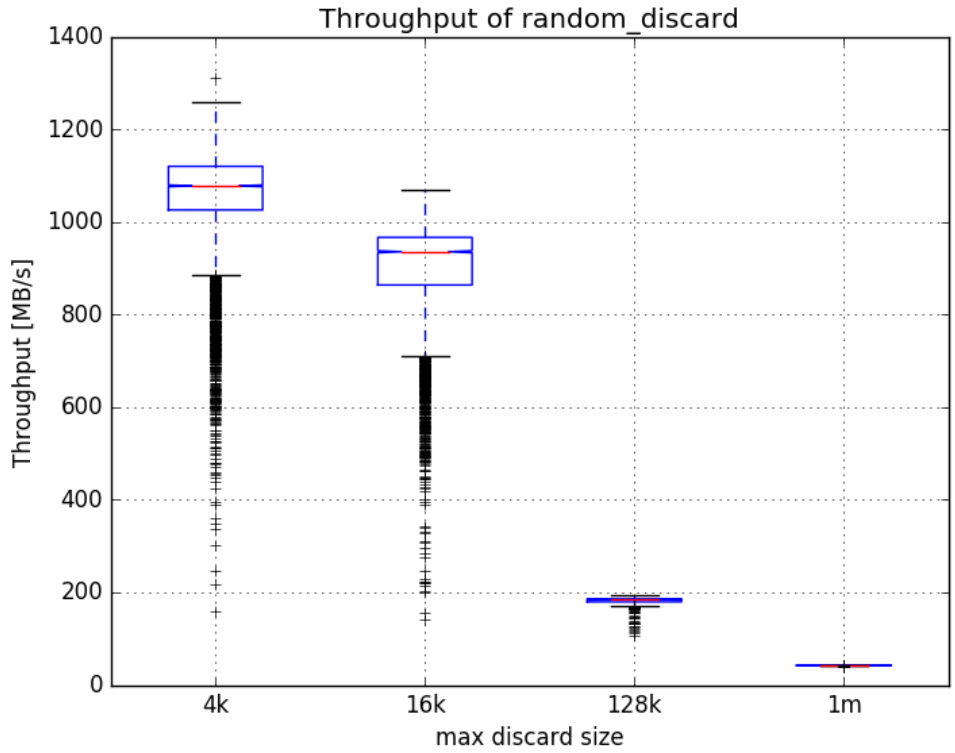


Figure 5.11: Testing of VDO volume created with variable maximum discard size. Prior to the test, the device was filled with random write workload

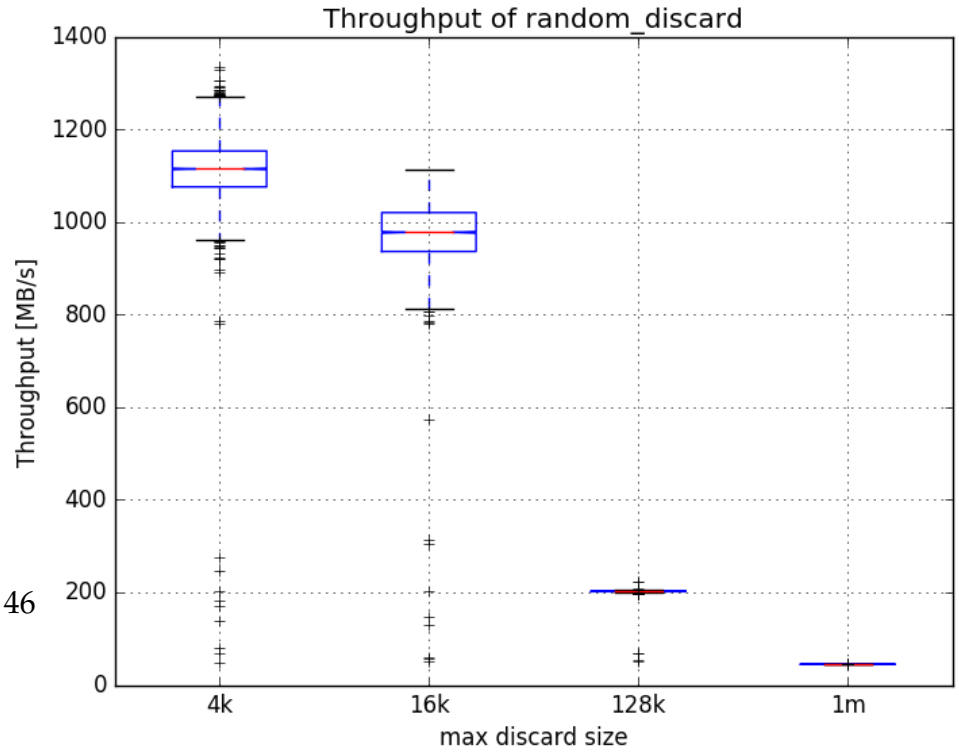
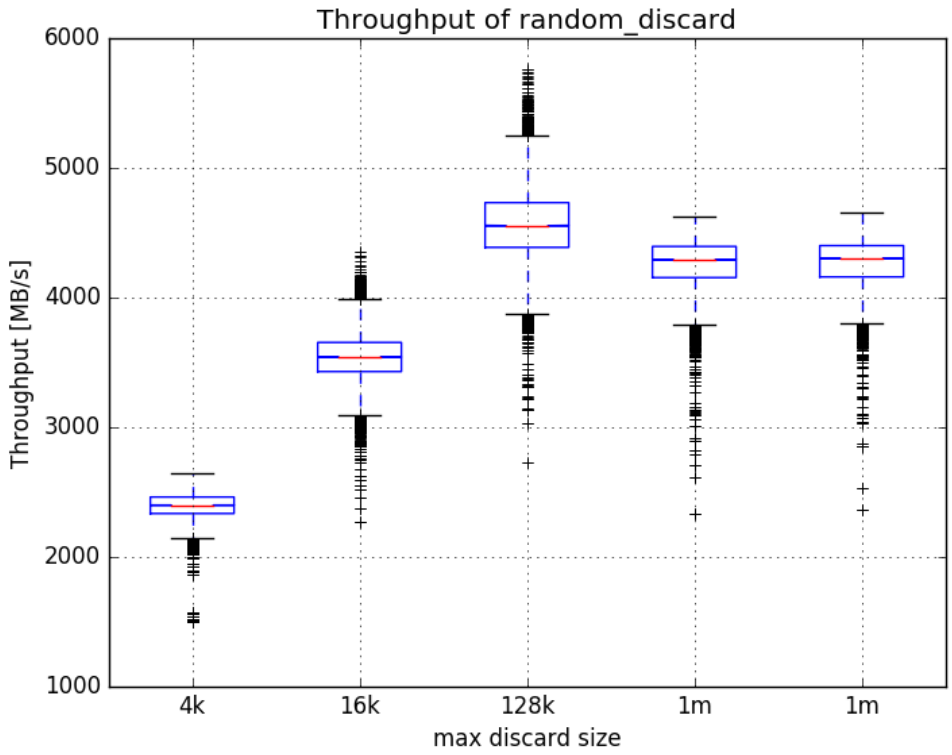


Figure 5.12: Testing of VDO volume created with variable maximum discard size on an empty VDO. The volume has been preallocated, but no additional data was written.



caption[Testing of VDO volume created with variable maximum discard size on an empty VDO, unallocated]Testing of VDO volume created with various maximum discard size parameters. This test was conducted on a fresh, unallocated instance of VDO

5.5 Write policies

VDO provides for different options for writing policies that are chosen with regard to the underlying device.

In synchronized mode, VDO user assumes the data is written to the persistent storage and no further commands are needed to make it persistent. User should set this option only when device under the VDO guarantees the data is written persistently when the write request is completed. If a volatile device is used with synchronous mode, user could potentially lose data. This option should be used only when the used device has persistent write cache or a write-through cache.

In asynchronous mode, VDO does not guarantee the data is written to the persistent storage after the completion of write request. Only when the user or a structure using VDO issues flush command, it can be sure the data is written persistently.

Up until lately, VDO async mode was not compliant with ACID policy, which could result in unexpected data loss. ACID policy of asynchronous mode was introduced in RHEL-8.2, however, the old ACID non-compliant version was kept in VDO under the name async-unsafe for users that don't mind minor data loss and would see a performance problem using safe async.

It is important for developers to know the performance impact of writing policies, therefore performance testing should be conducted.

In this section, performance testing of the three available policies was conducted and the results can be observed on Figure 5.13 and Table 5.4.

It is apparent that the synchronous policy will be the slowest, since for every write request VDO have to flush data to a storage. However, more interesting observation can be made between ACID-compliant and unsafe asynchronous mode. Since VDO is an enterprises product, a lot of care needed to be taken so that the safe asynchronous mode will not slow down VDO considerably. We can see in the presented results that this effort was successful.

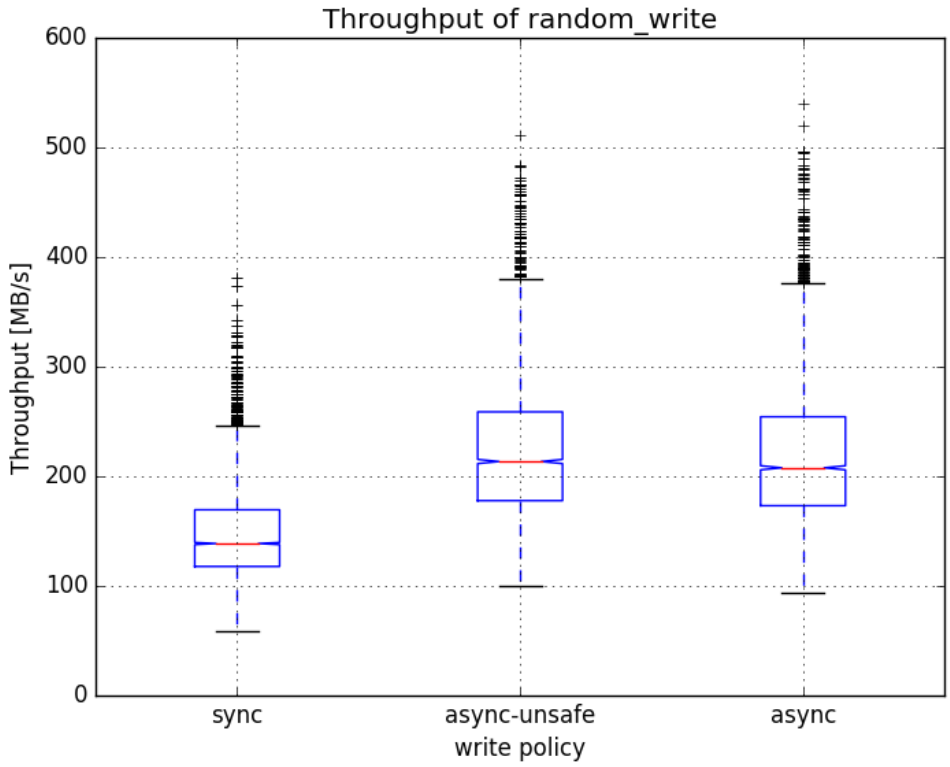


Figure 5.13: Testing of VDO volume created with various write policies. Synchronous, asynchronous and ACID non-compliant asnychronous-unsafe policy.

5.6 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 experiment 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

Throughput of random write (MB/s)						
	median	1st q.	3rd q.	min	max	stdev
sync	138.06	117.13	168.84	57.61	381.13	44
async-unsafe	212.87	177.17	258.17	99.24	510.32	66
async	207.14	172.57	253.68	92.76	539.94	69

Table 5.4: Testing of VDO volume created with various write policies. Synchronous, asynchronous and ACID non-compliant asynchronous-unsafe policy.

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 the recovery journal in the final 130 MB of the physical device. Because every data write has to do several journal writes, making the journal region of the physical device be on a completely different, faster device can greatly increase 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 extended 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.

Because the process of creating this configuration is not very intuitive, sequence of commands to do so is presented in 5.1

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.

One additional test was conducted by backing the journal with even more powerful device. The ending region of VDO was spread over a part of NVMe SSD in hopes it would speed up the process even further. However, power of NVMe is in its number of queues. The process that handles VDO journaling is not multi threaded. That is

the reason why backing VDO journal with NVMe instead of slower SSD brings few additional benefits.

We can observe the change in throughput using various hardware in a Figure 5.16

Example 5.1: Creating a volume with last reagion on NVMe device

```
$ # /dev/sda1 is an 8GB partition on HDD
$ # /dev/nvme0n1 is a 5GB partiiton on NVMe, however, size doesn't
    matter since we're only using first 2GB
$ vgcreate vg /dev/sda1 /dev/nvme0n1
$ lvcreate -n testLV1 -L 8g vg /dev/sda1
$ lvextend vg/testLV1 -L 10g
$ vdo create --name=testVDO --device=/dev/mapper/vg-testLV1  --
    vdoLogicalSize=80g
```

Throughput of random write (MB/s)						
test name	median	1st q.	3rd q.	min	max	stdev
vdo_hdd_baseline	1.28	0.97	1.87	0.85	4.73	0
tail_ssd	16.93	12.27	25.38	7.29	152.26	28
tail_nvme	19.08	13.32	29.67	8.27	149.68	21

Table 5.5: Performance of VDO with ending regions placed on different devices. In the first test, the recovery journal is together with data blocks on an HDD. In the second and third test, the journal is forced on SSD and NVMe device respectively.

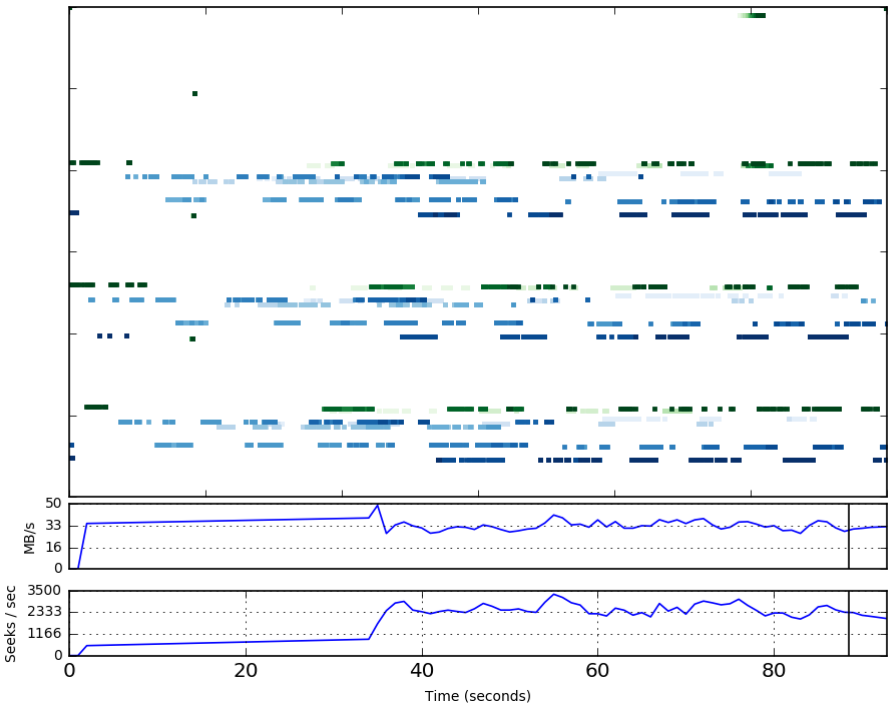


Figure 5.14: Access pattern of VDO to data blocks on an underlying device.

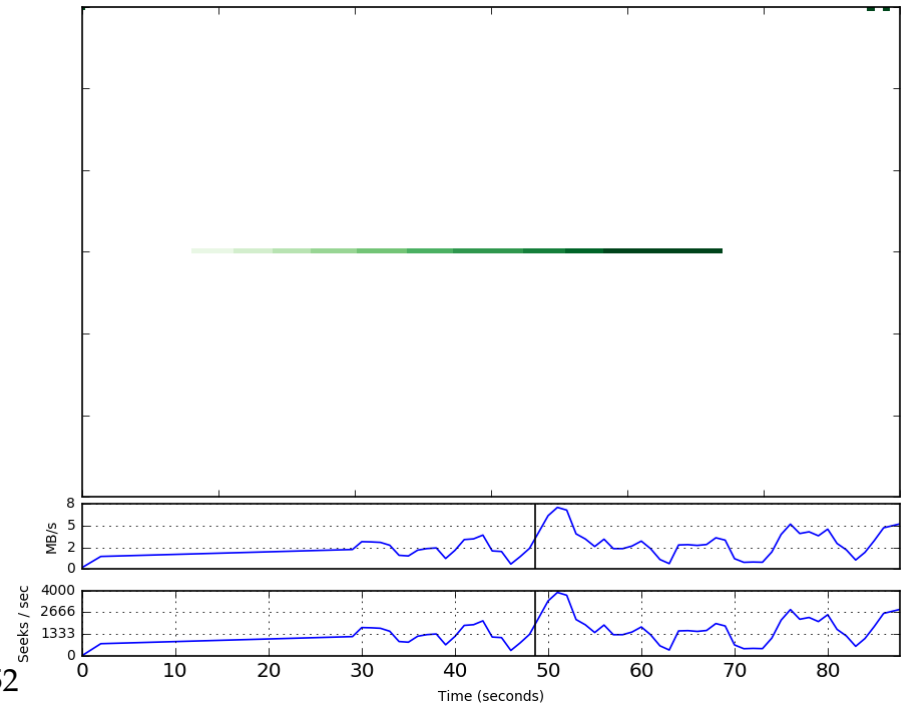


Figure 5.15: Access pattern of VDO to the recovery journal placed on additional fast storage.

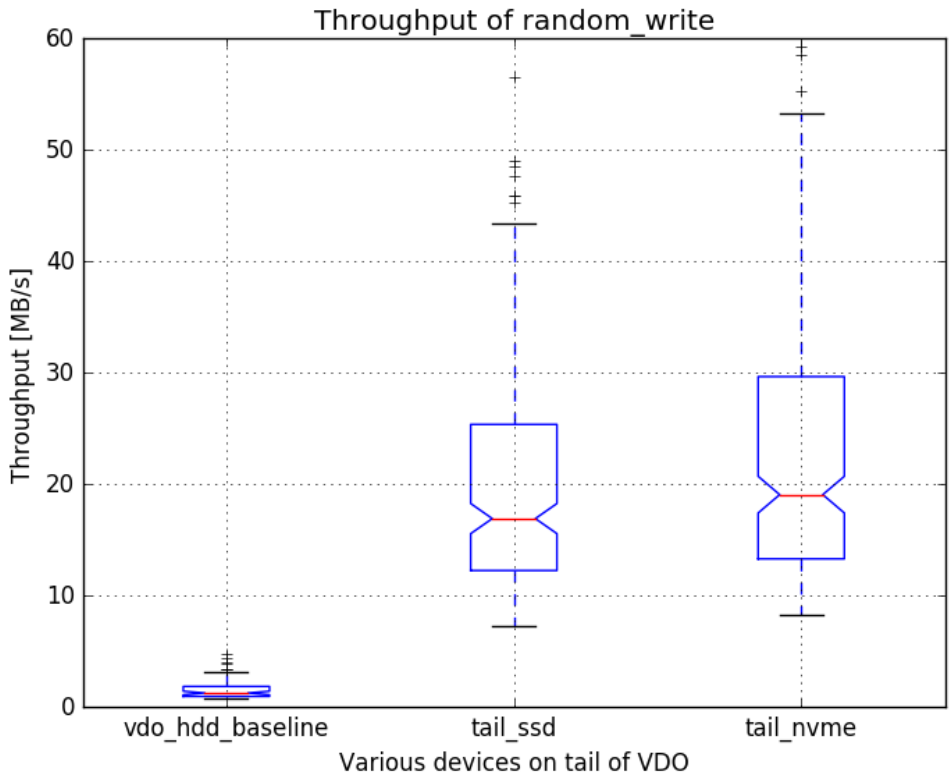


Figure 5.16: Performance of VDO with ending regions placed on different devices. In the first test, the recovery journal is together with data blocks on an HDD. In the second and third test, the journal is forced on SSD and NVMe device respectively.

6 Conclusion

VDO is a carefully designed structure that focuses on minimizing usage of other resources in order to save storage capacity. It is a complex technology that should be understood on at least basic level by anyone looking to use it. This thesis provides overview and explanation of VDO internal structures and their purposes.

Space saving and reducing the cost of storage is quite attractive to many professional, but also non-professional users. However, the cost of running and maintaining such technology should not be high enough to outweigh the advantages. That is the reason why performance testing should be considered by anyone interested in adopting the technology.

Users attempting to test performance of VDO for their deployment purposes should understand it's tuning mechanisms and their relationship with VDO internal structures and finally the outputted performance.

Many features were developed to user-friendly and easy-to-use benchmark fs-drift, so that anyone can test the block layer effectively. However, the terminology used in documentation of fs-drift is parallel to the terminology of other IO issuing benchmarks, so the explained principles can be transferred to multiple other frameworks.

Testing package using fs-drift was developed in a manner compliant with Red Hat Kernel Performance team. Along with the data processing library, it will be used to conduct regular tests of all major VDO releases. This effort should help developers to catch performance issues early on, so that the consistent, high-preforming product can be delivered to the community. Results from the tests were processed into easily readable reports available in the electronic appendix. Subset of all generated graphs and tables was included in the main text in order to illustrate some of the main points.

Using the created testing package, data processing library and explained methodology, various performance tests were conducted. Goal of some of the test was to further explain principles and internal logic of VDO. Following tests were aimed at performance tuning of VDO, by testing the most commonly used tunables as well as more ad-

vanced strategies. Commentaries accompanying the presented results presented useful advice for performance testing and tuning VDO.

Testing of VDO with unallocated mapping tree demonstrated the importance to set the right preconditions prior to the testing. By performing a simple preallocation operation, the test can achieve stable results much sooner, which saves the testing time.

Demonstration of a steady state test design represents a valuable advice to potential VDO users. It is important that anyone looking to deploy VDO know about it's behavior under loaded conditions. The performance of VDO observably decreases after a critical point of utilization and it's mapping space becomes fragmented.

Number of VDO threads is the main parameter users will use in attempts to increase performance. In the text, purpose of all tunable VDO threads is explained as well as their relationship with internal structures and performance. Testing the VDO under heavy load with various thread counts shows how increasing number of VDO threads can have positive effect on it's performance.

Users can also run into problems with overloading VDO's block map cache. Problem of small block map cache was demonstrated along with thorough explanation of it's cause and recommendation for additional tuning.

Testing of various writing policies can help users to decide which option to employ and what could be the expected consequences. Moreover, some users could be worried that re-engineering asynchronous writing policy to be ACID-compliant can decrease the performance of VDO. From the obtained test results, it appears not to be a problem, which was the aim of developers after all.

Discard operation is important for additional reclaiming of unused, but inaccessible parts of the volume. It is expected the users will issue discards on their own since there is no mechanism for VDO to automatically do that. However, discarding large amounts of space can take a lot of time, so it's important to be know how VDO discards work and how to tune them.

Last set of tests demonstrates advanced method of VDO performance tuning. In case a user will identify a journaling mechanism as bottleneck of VDO performance, possible remedy was introduced and explained along with example of exact commands for achiev-

ing it. This testing was an exciting way to show many possibilities of performance tuning and working with LVM.

The results from these tests were processed into easy-to-view reports available in Virtual Appendix B. The data processing script and testing package along with latest instance of fs-drift is also provided. All the new features developed for fs-drift are openly accessible by community.

Bibliography

[1] *VDO memory and storage requierements*. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/deduplicating_and_compressing_storage/deploying-vdo_deduplicating-and-compressing-storage#vdo-requirements_deploying-vdo. Accessed: 2020-10-05.

[2] *UDS index in VDO*. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/deduplicating_and_compressing_storage/maintaining-vdo_deduplicating-and-compressing-storage. Accessed: 2020-10-05.

[3] *VDO write policies*. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/deduplicating_and_compressing_storage/maintaining-vdo_deduplicating-and-compressing-storage#selecting-a-vdo-write-mode_maintaining-vdo. Accessed: 2020-10-05.

[4] Theo Haerder and Andreas Reuter. "Principles of Transaction-Oriented Database Recovery". In: *ACM Computing Surveys* 15 (1983), pp. 287–317.

[5] *Async-unsafe write policy*. https://www.redhat.com/archives/vdo-devel/2020-May/msg00001.html. Accessed: 2020-10-05.

[6] Avishay Traeger et al. "A Nine Year Study of File System and Storage Benchmarking". In: *Trans. Storage* 4.2 (May 2008), 5:1–5:56. ISSN: 1553-3077. DOI: 10.1145/1367829.1367831.

[7] *RHEL-8.1 release notes*. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/8.1_release_notes/index. Accessed: 2020-10-05.

[8] *RHEL-8.2 release notes*. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/8.2_release_notes/index. Accessed: 2020-10-05.

[9] Richard Jones. *Library for generating HTML code*. 2017. URL: https://pypi.python.org/pypi/html.

[10] Eric Jones et al. *SciPy: Open source scientific tools for Python*. 2001–. URL: http://www.scipy.org/.

[11] *Seekwatcher*. https://oss.oracle.com/~mason/seekwatcher/. Accessed: 2020-22-02.

A Testing hardware

Testing hardware was borrowed form Red Hat Kernel performance team laboratory. The machines are powerful enough to testing high performing drivers and tools. All storage devices used are regularly checked for health and immediately replaced if faulty.

A.0.1 Machine 1

A.0.2 Machine 2

This machine is used for regular testing of various file systems and storage layers. It's equipped with 3 220 GB SATA SSD, one 220 GB SAS rotational HDD. Additionally, there is one NVMe device.

Machine 1	
Model	Supermicro X11SPL-F
Processor	Intel Xeon Silver 4110
Clock speed	2.10 GHz (8 cores)
Memory	49 152 MB

Table A.1: Testing machine 1

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
Testing SSD	
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

Table A.2: Testing devices instelled on Machine 1

Machine 1	
Model	Supermicro X11SPL-F
Processor	Intel Xeon Silver 4110
Clock speed	2.10 GHz (8 cores)
Memory	49 152 MB

Table A.3: Testing machine 2

Testing Hard Drives	
Model	WD HGST Ultrastar
Capacity	1 TB
Interface	SAS 12 GB
Type	Rotational HDD
Logical sector size	4096 B
Physical sector size	4096 B
Testing SSDs 3x	
Model	Micron 5100 MTFD
Capacity	240 GB
Interface	SATA 6 GB
Type	SSD
Logical sector size	512 B
Physical sector size	4096 B
Tesing NVMe	
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

Table A.4: Testing devices installed on Machine 2

B Virtual appendix

Virtual appendix contains a code or data that could not be displayed in the main text.

B.0.1 fs-drift

Version of fs-drift benchmark that has been used for conducting tests for the purposes of this thesis. This improved version is also available to the community online using git framework. This version will be used for regular testing of VDO, thin provisioning and file systems in Red Hat Kernel Performance team.

B.0.2 drift_job

A testing package containing fs-drift developed to be compliant with Red Hat Kernel Performance team testing workflow. The testing package prepares environment, runs the tests and gather results and system metadata, later sending all this on an result gathering server.

B.0.3 drift_compare

Library for creating easy-to-view reports from data generated by drift_job testing package. The reports will display many useful information about testing environment as well as actual charts and plots displaying measured performance.

B.0.4 results

Subset of all testing results that were chosen to be displayed in this thesis. Every section of Chapter 5 is represented by corresponding folder. Some of the folders contain videos obtained while tracing block requests.