

MASARYK UNIVERSITY
FACULTY OF INFORMATICS



The effects of age on file system performance

BACHELOR'S THESIS

Samuel Petrovič

Brno, Spring 2017

Replace this page with a copy of the official signed thesis assignment and a copy of the Statement of an Author.

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

Acknowledgement

I would like to thank to my technical advisory Matúš Kocka for tremendous help and guidance. I would also like to thank Red Hat for providing necessary testing environment.

Abstract

This is the abstract of my thesis, which can span multiple paragraphs.

Keywords

file system, XFS, EXT4, IO, performance, aging, fragmentation, SSD, HDD, TRIM, fs-drift, FIO, benchmark

Contents

1	Introduction	1
2	Related work	5
2.1	<i>Aging file system using real-life data</i>	5
2.2	<i>Synthetic aging simulation</i>	6
3	File system and storage devices	7
3.1	<i>File systems</i>	7
3.1.1	XFS	8
3.1.2	Ext4	8
3.2	<i>Storage</i>	9
3.2.1	Hard disk drive	9
3.2.2	Solid state drive	9
4	Environment setup and benchmark tools	11
4.1	<i>Beaker</i>	11
4.2	<i>Storage generator</i>	11
4.3	<i>FIO</i>	12
4.4	<i>Fs-drift</i>	12
4.4.1	Changes in original code	14
5	Aging the file system	17
5.1	<i>Fs-drift configuration</i>	17
5.2	<i>File system images</i>	18
5.3	<i>Implementation details</i>	19
6	Performance testing of aged file system	23
6.1	<i>FIO configuration</i>	23
6.2	<i>Test structure</i>	24
7	Results	27
7.1	<i>Testing environment</i>	27
7.2	<i>Data processing</i>	29
7.2.1	Fragmentation	29
7.2.2	Evaluation of data generated by fs-drift	30
7.3	<i>Testing on images of aged file system</i>	30

7.4	<i>Performance of aged file system</i>	30
7.5	<i>Differences between XFS and EXT4</i>	31
7.6	<i>Performance accross different storage devices</i>	35
7.7	<i>Testing with <code>fstrim</code></i>	36
8	Conclusion	39
	Bibliography	41
A	Reports	43
A.1	<i>Test of medium utilisation using XFS on HDD</i>	43
A.2	<i>Test of high utilisation using XFS on HDD</i>	47
A.3	<i>Test of high utilisation using EXT4 on HDD</i>	51
A.4	<i>Test of XFS on SSD with regular trimming</i>	55
A.5	<i>Test of EXT4 on SSD with regular trimming</i>	59
A.6	<i>Test of XFS on SSD without regular trimming</i>	63
B	Examples	67

List of Tables

- 7.1 Comparison of latencies of medium and high utilisation 32
- 7.2 Comparison of latencies gathered during testing with XFS and EXT4 on SSD and HDD 34
- 7.3 Comparison of latencies of gathered during aging test on HDD and SSD 35
- 7.4 Differences in latency on trimmed and non trimmed file system (XFS) 38

List of Figures

- 4.1 Normal distribution of file access 14
- 4.2 Moving random distribution 15
- 4.3 File size distribution as measured by Agrawal [17] 16
- 4.4 Logarithmic normal distribution of file size 16
- 5.1 Activity diagram of file system aging test 21
- 6.1 Activity diagram of testing of aged file system 25
- 7.1 Latency growth induced by aging test 33
- 7.2 Evolution of latency of create operation, in untrimmed file system (XFS) 37
- A.1 Free space fragmentation of XFS during testing of medium utilisation of HDD 43
- A.2 Size distribution of file extents of XFS during testing of medium utilisation of HDD 44
- A.3 Usage of available space of XFS during testing of medium utilisation of HDD 45
- A.4 Evolution of latencies while testing XFS during testing of medium utilisation of HDD 46
- A.5 Evolution of free space fragmentation of XFS during testing of high utilisation of HDD 47
- A.6 Size distribution of file extents of XFS during testing high utilisation of HDD 48
- A.7 Usage of available space of XFS during testing of high utilisation of HDD 49
- A.8 Evolution of latencies while testing XFS during testing of high utilisation of HDD 50
- A.9 Evolution of free space fragmentation of EXT4 during testing of high utilisation of HDD 51
- A.10 Size distribution of file extents of EXT4 during testing of high utilisation of HDD 52
- A.11 Usage of available space of EXT4 during testing of high utilisation of HDD 53
- A.12 Evolution of latencies while testing EXT4 during testing of high utilisation of HDD 54

- A.13 Evolution of free space fragmentation of XFS during testing on SSD with regular trimming 55
- A.14 Size distribution of file extents of XFS during testing on SSD with regular trimming 56
- A.15 Usage of available space of XFS during testing on SSD with regular trimming 57
- A.16 Evolution of latencies of XFS during testing on SSD with regular trimming 58
- A.17 Evolution of free space fragmentation of EXT4 during testing on SSD with regular trimming 59
- A.18 Size distribution of file extents of EXT4 during testing on SSD with regular trimming 60
- A.19 Usage of available space of EXT4 during testing on SSD with regular trimming 61
- A.20 Evolution of latencies of EXT4 during testing on SSD with regular trimming 62
- A.21 Evolution of free space fragmentation of XFS during testing on SSD without regular trimming 63
- A.22 Size distribution of file extents of XFS during testing on SSD without regular trimming 64
- A.23 Usage of available space of XFS during testing on SSD without regular trimming 65
- A.24 Evolution of latencies of XFS during testing on SSD without regular trimming 66

1 Introduction

File systems remain an important part of modern storage solutions. Large, growing databases, multi-media and other storage based applications need to be supported by high-performing infrastructure layer of storing and retrieving information. Such infrastructure have to be provided by operating systems (OS) in form of file system.

Originally, file system was a simple tool developed to handle communication between OS and physical device, but today, it is a very complex piece of software with large set of tools and features to go with.

Performance testing is an integral part of development cycle of most of produced software. Because of growing complexity of file systems, performance testing took of as an important part of file system evaluation.

The standard workflow of performance testing is called out-of-box testing. Its principle is to run benchmark (i.e. testing tool) on a clean instance of OS and on a clean instance of tested file system [1]. Generally, this workflow presents stable and meaningful results, yet it only gives overall idea of file system behavior in early stage of its life cycle.

File systems, as well as other complex software, are subjected to progressive degradation, referred to as software aging [2]. Causes of file system aging are many, but mostly fragmentation of free space, unclustered blocks of data and unreleased memory. This degradation cause problems in performance and functionality over time. Understanding of performance changes of aged file system can help developers to implement various preventions of aging related problems. Furthermore, aging testing can help developers with implementation of long-term performance affecting features.

Researching of file system aging can be done in two stages, aging the file system and testing of the aged instance. In the first stage, observation about *evolution* of fragmentation and performance can be made. The second stage brings insight into *state* of performance of aged file system.

In the first part, this thesis describe implementation of two flexible tests, which correspond with afformentioned stages.

In the first-stage testing, file system is aged using open-source benchmark fs-drift. While aging the file system, fragmentation of free space and latency of IO operations is periodically recorded. After the aging process, additional information about file fragmentation and file size distribution is recorded as well. Once this information is gathered, image of aged file system instance is created. To save space, only metadata of created file system is used, since content of created files is random and therefore irrelevant.

Created images are then used in second-stage testing. By using created images, higher stability and consistency of results is achieved. By reloading the file system image on the device, it is possible to bring the file system to the original, aged state, therefore for every performance test the same file system layout is available.

In the second-stage testing, file system is brought to the aged state by reloading image corresponding with desired testing configuration. Before every performance test, some volume can be released from file system, so the performance test has space to test on. After environment initialization (`fstrim`, `sync`), the performance test can begin. Testing of fresh instance of file system is done as well to conclude if any performance drop occur.

In the later parts of this thesis, usage of developed tests on different configurations of file systems and storage is demonstrated. The subject of research are differences between popular Linux file systems (XFS, Ext4) and differences between used storage technologies (solid state and hard disk drives) in context of aging. Because of nature of collected data, a processing tool was implemented to parse large amount of information into human readable reports with interactive charts.

In the Chapter 2, the text present already conducted research of effects of age and fragmentation on file system performance. Chapter 3 describes used storage and file systems and theirs main features. Chapter 4 introduce tools used in implementation of tests while describing their relevant features. Chapters 5 and 6 describe actual implementation of mentioned tests. Chapter 7 describes testing environment and and used data procesing. Then, it presents results obtained by using created tests. The research is focused describing the effect of aging file systems, differences between file systems in terms of aging and relations of underlying storage to performance of aged file system. In Chapter 8, I discuss the effects of age on file system performance as

well as recommendations for this type of testing. Furthermore, future of testing of file system performance and aging is discussed.

Appendix A contains overall information about tests and generated charts. Appendix B contains examples of usage of used tools.

2 Related work

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

2.1 Aging file system using real-life data

This approach is based on modeling the aged file system using data collected from file systems used in real-life environment.

Such data can be in form of snapshots (i.e. images) of file systems, as was thoroughly described by Smith and Seltzer [3].

The snapshots were collected nightly over a long period of time (one to three years) on more than fifty file systems. By comparing pairs of snapshots in the sequence, performed operations were estimated, resulting in a very realistic aging workload. However, as some studies suggest, most of files have life span shorter than 24 hours [4]. Therefore, as Smith and Seltzer admit, by snapshotting every night, this process does not account for most of the created files, resulting in loss of important part of data.

Furthermore, to age a file system sized 1024 MB, 87.3 GB of data had to be written, taking 39 hours to complete, rendering the workflow impractical for in-production testing needs.

Smith and Seltzer also defined a layout score as a method to evaluate fragmentation of a file system. Layout score is defined as a fraction of blocks of file, which are contiguously allocated. Files of one block size are ignored, since they can't be fragmented, and for every file, first block is ignored too. Evaluation of the whole file system is then computed as an aggregated layout score of all files.

The problem resulting from not tracking shortly lived files can be solved by another approach called collecting traces. Traces are sequences of IO operations performed by OS, captured at various levels (system call, drivers, network, etc.). The sequence of operations can be replayed back to the file system, aging it in a realistic manner.

Overall, using real-life data to age file systems brings realistic results, but at a cost of higher expenses, such as storing the collected data. Additionally, to cover cases of different types of file system usage, data from several such file systems have to be collected, expanding the amount of needed data even further. Such materials are not always available, rendering this type of approach useful only in cases the researcher is already in their possession.

2.2 Synthetic aging simulation

Synthetic aging is a type of aging that does not require real-life data for its running. It relies on purely artificial workload performed on a file system, invoking aging factors, such as fragmentation.

Fast file system aging was described as a part of a trace replay benchmark called TBBT [5]. This type of aging consists of sequence of interleaving append operations on a set of files. By controlling the amount of files involved in the process, researchers had great control over fragmentation. Such workflow, while creating desired fragmentation, is however quite unrealistic, making the results of testing on such file system questionable [1].

Another attempt of inducing fragmentation was made in an empirical study of file system fragmentation in mobile storage systems [6]. The aging process consisted of filling the device by alternative creation of files larger or equal to 100 MB and smaller or equal 100 kB. After 100% file system utilization was reached, 5% of files was randomly deleted.

However, for truly realistic insight, a workflow generator which tries to mimic real-life usage can more suitable.

A workload generator such as fs-drift¹, can be used (while carefully configured) to simulate desired long term real-life usage. While running, fs-drift is creating requests of variety of IO operations. The probability with which would be operation chosen can be controlled by workload table. In addition, this tool offers different probability distributions of file access, making it easier to mimic behavior of natural user.

1. Mixed workload file system aging test [7]

3 File system and storage devices

In this chapter, I present basic information about used file systems and storage devices and its features relevant to performance and aging.

3.1 File systems

File system is a set of tools, methods, logic and structure to control how to store and retrieve data on and from device.

The system stores files either continuously or scattered across device. The basic accessed data unit is called a block, which capacity can be set to various sizes. Blocks are labeled as either free or used.

Files which are non-contiguous are stored in form of extents, which is one or more blocks associated with the file, but stored elsewhere.

Information about how many blocks does a file occupy, as well as other information like date of creation, date of last access or access permissions is known as metadata, e.g. data about stored data. This information is stored separately from the content of files. On modern file systems, metadata are stored in objects called index nodes (e.g. inodes). Each file a file system manages is associated with an inode and every inode has its number in an inode table. On top of that the file system stores metadata about itself (unrelated to any specific file), such as information about bad sectors, free space or block availability in a structure called superblock.

In this thesis, targeted file systems are two most popular Linux file systems [8], XFS¹ and Ext4², which are also main Red Hat supported file systems. These file systems belong to a category of journaling file systems.

Journaling file system keeps a structure called journal, which is a buffer of changes not yet committed to the file system. After system failure, these planned changes can be easily read from the journal, thus making the file system easily fully operational, and in correct and consistent state [9].

1. XFS file system [10]

2. Fourth extended file system [11]

3.1.1 XFS

XFS is a 64-bit journaling file system known for its high scalability (up to 9 EB) and great performance. Such performance is reached by architecture based on allocation groups.

Allocation groups are equally sized linear regions within file system. Each allocation group manages its own inodes and free space, therefore increasing parallelism. Architecture of this design enables for significant scalability of bandwidth, threading, and size of file system, as well as files, simply because multiple processes and threads can access the file system simultaneously.

XFS allocates space as extents stored in pairs of B^+ trees, each pair for each allocation group (improving performance especially when handling large files). B^+ trees is indexed by the length of the free extents, while the other is indexed by the starting block of the free extents. This dual indexing scheme allows efficient location of free extents for IO operations.

Prevention of file system fragmentation consist mainly of a features called delayed allocation and online defragmentation.

Delayed allocation, also called allocate-on-flush is a feature that, when a file is written to the buffer cache, subtracts space from the free space counter, but won't allocate the free-space bitmap. The data is held in memory until it have to be stored because of system call. This approach improves the chance the file will be written in a contiguous group of blocks, avoiding fragmentation and reducing CPU usage as well.

3.1.2 Ext4

Ext4, also called fourth extended filesystem is a 48-bit journaling file system developed as successor of Ext3 for Linux kernel, improving reliability and performance features. Ext4 is scalable up to 1 EiB (approx. 1.15 EB). Traditional Ext2 and Ext3 block mapping scheme was replaced by extent based approach similar to XFS, which positively affects performance.

Similarly to XFS, Ext4 use delayed allocation to increase performance and reduce fragmentation. For cases of fragmentation that still occur, Ext4 provide `e4defrag` tool to defragment either single file, or

whole file system. Performance penalties were, however, recognized and online defragmentation workflow was proposed by Sato [12].

3.2 Storage

3.2.1 Hard disk drive

A hard-disk drive (i.e. HDD) is a type of storage device which use one or more magnetic plates to store data. Data can be retrieved by rotating the plates and positioning magnetic read-write heads. The plates rotate at stable speed at around 7500 rpm (and more on enterprise-level hardware).

Since the parts of HDD have to physically move to reach desired location, there is a latency to the data access. The time for magnetic head to find next relevant block of data is called a *seek time*. Because the length of seek time has significant impact on overall IO performance, OS have to do a lot of optimising, such as pre-fetching.

Obviously, block layout would have a large impact on performance of this kind of device. The amount of fragmentation (thus aging) affect the number of performed seeks. This cause the pressure on file system to store data more contiguously and also cluster related data.

3.2.2 Solid state drive

Solid state drive (i.e. SSD) is a type of storage device which use integrated circuit to store and retrieve data. SSD has no moving parts, therefore the data access is purely electronic, which results in lower access time and latency than HDD [13].

However, on SSDs, data cannot be directly overwritten (as in HDDs). The cell of an SSD can only be directly written to, therefore have to be erased before writing. Moreover, due to physical construction limits, write operation can be conducted on one page (4-16 kB), but erasure have to be done on a whole block (128 to 512 pages). Therefore, if OS have to rewrite some part of a page (e.g. update metadata), the page have to be read, modified and submitted back on available part of the drive. The original page is then marked as discarded. This effect is known as write amplification [14], and is computed as amount of bytes written to the device divided by amount of bytes requested to be

written by user. For example, if user updates 512 B of data on device that uses 8 kB pages, write amplification is then 16.

In addition, the memory cell can be rewritten finite amount of times, therefore, a form of wear leveling has to be employed. Wear leveling prevents frequently accessed blocks from exhaustion of cell life-cycle by moving files around the device.

Static wear leveling [15] rotates even unused files around the drive to ensure equal wear. However, deleting file, in file system doesn't always ensure its deletion on the device. Typically, the file is only marked as deleted, but this information is not submitted to underlying device itself. Problem is, files that are not valid for file system anymore can still circulate on the device, increasing its wear and write amplification, since there are less free block that could have been.

To decrease this effect, trim commands were introduced [16]. Trim command communicate to SSD all the deletions that were realised in the file system, so the drive can erase blocks accordingly. Nevertheless this operation is reducing performance of IO operations while being conducted, it can increase overall performance and life time of an SSD.

4 Environment setup and benchmark tools

In this chapter, I present tools which were used to implement automated tests for creating and storing aged file systems and measuring their performance. Furthermore, I describe the main features and means of their usage. All the presented tools are open source projects.

4.1 Beaker

Beaker is an open source project aimed at automating testing workflow. The software can provision system from a pool of labs, install OS and packages, configure environment and perform tasks. The whole process is guided by sequence of instructions in an XML format. Examples can be found in Appendix A.

The workflows created as a part of this thesis are implemented in form of Beaker task packages, which can be directly used by Beaker. A task package has to include Makefile which can then run other scripts included in the package. Scripts can also communicate with Beaker via API installed on every machine. By this means, tests can send logs or report their status on server while running.

4.2 Storage generator

Storage generator is a Beaker task developed by Jozef Mikovič. It is capable of automated configuration of storage on a provisioned system. In a single device mode, storage generator simply creates new partition on the given device and creates and mounts file system. In a recipe mode, storage generator follows set of instruction to create more advanced configurations such as merging multiple devices using LVM, creating LVM cache or encrypted volume. Example of usage can be found in Example B.3

The creation of XFS file system is standard, but Ext4 file system is created with additional option, disabling *lazy init*. Lazy init is a feature which allows for fast creation of file system by not allocating all the available space at once. The space is allocated later, as the file system

grows. Such additional allocation, however would skew data collected in the first hours of the test, therefore it is disabled in these tests.

4.3 FIO

Flexible Input/Output tool is a workload generator written by Jens Axboe. It is a tool well known for its flexibility as well as large group of users and contributors. The flexibility is integral for conducting less artificial and more natural performance tests. However, approaching more natural test behavior, stability of results drop, so ideal equilibrium between these two requirement has to be found.

FIO accepts the workload specification as either a configuration file or a single line. Multiple different jobs can be specified as well as global options valid for every job.

There is a possibility to choose from 4 IO operations to be performed (or their mix). These operations are sequential write, sequential read, random write and random read. Verification of the issued data is offered as well. Size of generated file and block size can be controlled too and it can be either stable or chosen from given range. For cache-tiering workloads, different random distributions (e.g. Zipf, Gauss) can be specified. FIO also supports process forking and threading.

After the test, FIO generates overall report of measured performance. However, logging of multiple properties can be enabled, giving researchers even more oversight about the nature of file system performance.

4.4 Fs-drift

Fs-drift is a very flexible aging test, which can be used to simulate lots of different workloads. The test is based on random file access and randomly generated mix of requests. These requests can be writes, reads, creates, appends, truncates or deletes.

At the beginning of run time, the top directory is empty, therefore create requests success the most, other requests, such as read or delete, will fail because not many files has yet been created. Over time, as the file system grows, create requests began to fail and other

requests will more likely succeed. File system will eventually reach a state of equilibrium, when requests are equally likely to execute. From this point, the file system would not grow anymore, and the test runs unless one of the stop conditions are met.

The mix of operation probabilities can be specified in separate CSV file. Fs-drift will try to issue more create operations at the beginning of testing, so other operations execute with higher likelihood.

The file to perform a request on is randomly chosen from the list of indexes. If the type of random distribution is set to *uniform*, all indexes have the same probability to be chosen. However, if the type of random distribution is set to *gaussian*, the probability will behave according to normal distribution with the center at index 0 and width controlled by parameter *gaussian-stddev*. This is useful for performing cache-tiering tests. Please note, that file index is computed as modulo maximal number of files, therefore instead of accessing negative index values, the test accesses indexes from the other side of spectrum, see Figure 4.1

Furthermore, fs-drift offers one more option to influence random distribution. After setting parameter *mean-velocity*, fs-drift will choose files by means of moving random distribution. The principle relies on a simulated time, which runs inside the test. For every tick of the simulated time, the center of bell curve will move on the file index array by the value specified by *mean-velocity* parameter. By enabling this feature, the process of testing moves closer to reality by simulating more natural patterns of file system access (the user won't access file system randomly, but rather works with some set of data at a time). On Figure 4.2, you can see bell curve moving by 5 units two times.

Fs-drift offers even more parameters to control the test run such as number of directories, levels of directories, or enabling *fsync*/*fdatasync* to be called. To stop the fs-drift, one of the stop conditions have to be met. The stop condition can be either reached maximal number of performed operations, running out of time, or appearance of stop file.

For evaluation of the aging process, fs-drift can log latency of the performed operations. However, the log doesn't differentiate between the operations, which could be useful for further research.

Used configuration of fs-drift for purposes of aging testing is further described in Chapter 5.

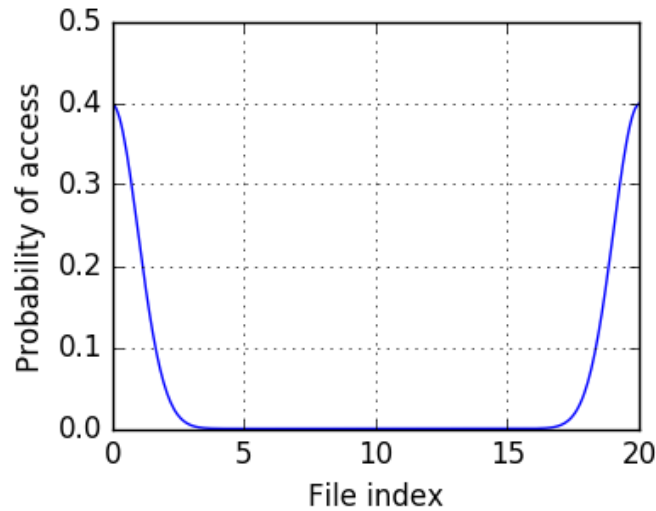


Figure 4.1: Normal distribution of file access

4.4.1 Changes in original code

Several changes had to be made to the original code prior to testing.

The most obvious problem with the tools was, that it did not conduct delete operations. As stated, deleting files is quite crucial to file system aging.

Another problem emerged when gathering statistics of response time evolution through the aging process. Since the tool is generating the IO requests at random, sometimes error occurs. Most operations need the file to exist to success, but sometimes, the file is non-existent. The problem with response time logging was, it logged the response time even if the operation didn't carry, causing noise in the data.

Further problem with response time logging was, it didn't differentiate between operations. Such distinction could greatly affect the way researchers can find problems with file system evolution.

As mentioned, possibility of specifying *pause file* was added, making the fs-drift easy to pause for file system contents inspection.

Another feature added to fs-drift was non-uniform distribution of file sizes. Originally, fs-drift only uniform distribution to choose file size from zero to specified maximum size. Such implementation offers very little control over file size distribution and is quite unre-

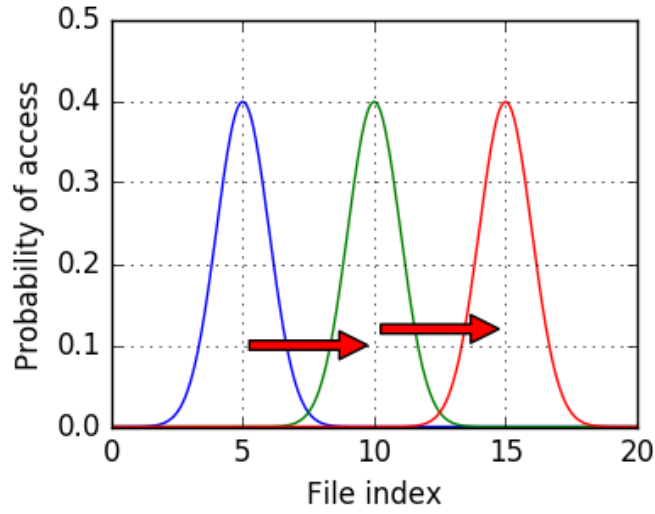


Figure 4.2: Moving random distribution

alistic. Therefore, possibility to request natural file size distribution was added. As a reference for file size distribution, five year old study of file system metadata was used [17]. Figure 4.3 shows measured file size distribution of used file systems. To mimic this layout, log-normal distribution was modeled. The shape of resulting distribution is shown in Figure 4.4.

The version used for testing in this thesis have all afformentioned features implemented and problems repaired. The code is also included in eletronic appendix.

4. ENVIRONMENT SETUP AND BENCHMARK TOOLS

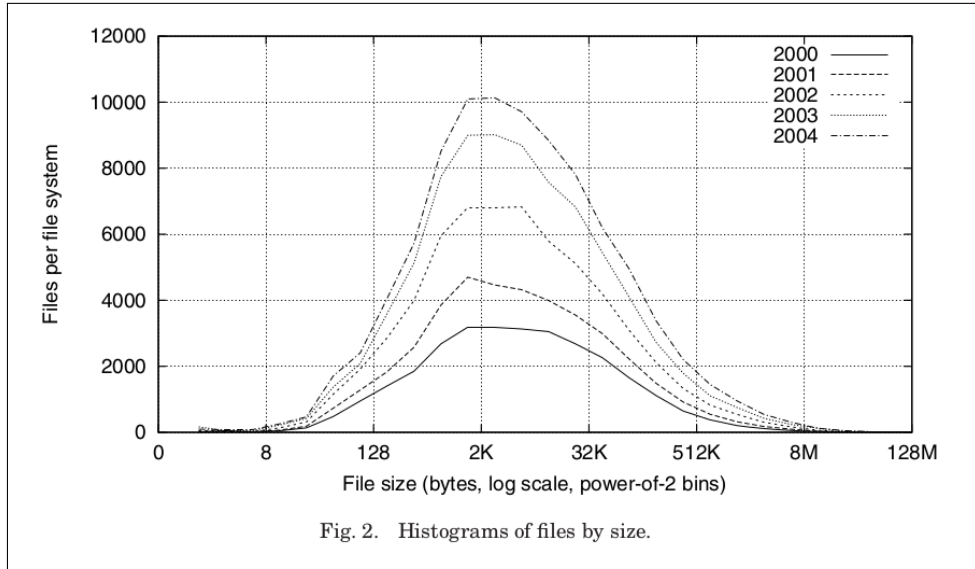


Figure 4.3: File size distribution as measured by Agrawal [17]

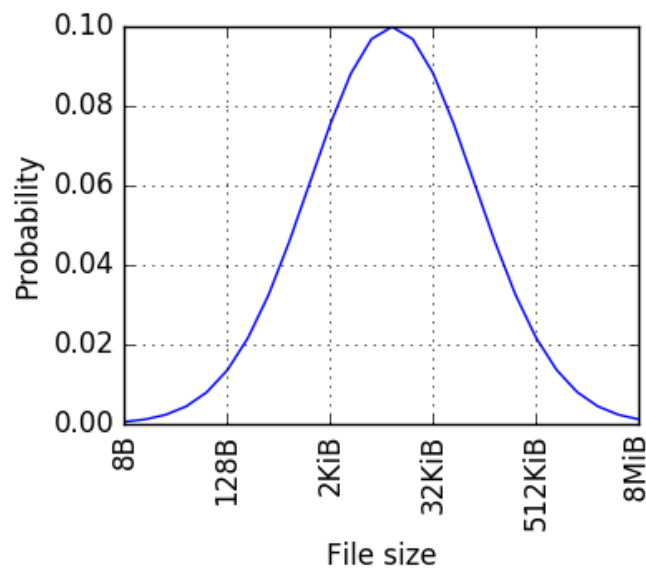


Figure 4.4: Logarithmic normal distribution of file size

5 Aging the file system

In this chapter, I describe process of development of file system aging workflow and its implementation as a form of automated test.

5.1 Fs-drift configuration

As mentioned, fs-drift was used as a mean of bringing file systems to an aged, fragmented state. Fs-drift is quite flexible, therefore a lot of parameters and their impact on the final layout had to be considered.

First, the amount of fullness had to be taken into account. Heavily used file systems tend to be full at amounts ranging from 65 [3] to 100 percent [17]. However, fs-drift does not offer an option to directly control the fullness of the file system. As the creator states in README, to fill a file system, maximum number of files and mean size of file should be defined such that the product is greater than the available space. Parameters to overload the volume are not difficult to come up with. The problem is, the random nature of the test doesn't allow for meaningful reproducibility of the reached equilibrium. In most cases, fs-drift plainly saturates given volume, so utilization remains at 100% through the rest of the testing time. This drawback was overridden by a change in fs-drift code¹. By adding the possibility to pause fs-drift, other processes can stop fs-drift to generate IO requests if the file system usage reaches a specified amount and release some space. This way, desired amount of maximal utilization can be reached.

Duration of the test is an important factor as well. In general, the testing time should be long enough for the environment to stabilise. After developing a test, it is necessary to conduct few pilot runs to confirm if any stabilisation of performance occur and set the duration accordingly for best effectivity

Fs-drift offers to control testing length by elapsed time. The elapsed time is computed as current time subtracted from starting time. However, pausing the fs-drift to release space, examine file system, etc. resulted in non-uniform testing time across runs. Therefore, testing length by operation count is used in conducted tests.

1. For all the changes made to original code of fs-drift, see Chapter 4

File size is another important factor of successful simulation. To consider a simulation successful, resulting layout should have similar file size distribution as real-life file systems. For that reason, when testing, natural file size distribution is turned on. The specifics of natural file distribution are described in Subsection 4.4.1.

During the test, all available file operations are used, with more weight placed on layout alternating operations (create, delete, truncate, append, random write). The non-alternating operations are there to verify, if any performance drop occurs during the aging process.

5.2 File system images

File system images can be created by using tools developed to inspect file systems in case of emergency. For Ext file systems, there is a tool called `e2image` and for XFS, `xfs_metadump`. Both tools create images as sparse files, so compression is needed.

`E2image` tool can save whole contents of a file system or just its metadata and offers compression of image as well. Created images can be further compressed by tools such as `bzip2` or `tar`. Such images can be later reloaded back on a device. From that point, file system can be mounted. Example 5.1 shows creating file system image using `e2image` tool. Example 5.2 shows reloading image on device.

Example 5.1: Creating compressed image using `e2image`

```
$ e2image -Q $DEVICE $NAME.qcow2
```

Example 5.2: Reloading compressed image

```
$ e2image -r $NAME.qcow2 $DEVICE
```

`Xfs_metadump` saves XFS file system metadata to a file. Due to privacy reasons file names are obfuscated (this can be disabled by `-o` parameter). As well as `e2image` tool, the image file is sparse, but `xfs_metadump` doesn't offer a way to compress the output. However, output can be redirected to `stdout` and compressed further on. Generated images, when uncompressed, can be reloaded back on device by tool `xfs_mdrestore`. File system can be then mounted and inspected as needed. Example 5.3 shows creating file system image using `xfs_metadump`. Example 5.4 shows reloading image on device using `xfs_mdrestore`.

Example 5.3: Creating compressed image using xfs_metadump

```
$ xfs_metadump -o $DEVICE -l bzip2 > $NAME
```

Example 5.4: Reloading image using xfs_mdrestore

```
$ xfs_mdrestore $NAME $DEVICE
```

5.3 Implementation details

Workflow of image creating is contained in the Beaker task `drift_job`. After extracting `fs-drift`, the main script starts `python` script, which handles the process of running `fs-drift`. Settings of `fs-drift` are passed as a parameter and then parsed inside the script.

Before running the `fs-drift`, asynchronous thread `async_worker` is triggered. `Async_worker` offers ways to additionally control testing and to work with file system while the test is running. The thread wakes up at specified intervals. Upon awakening, `fs-drift` is paused, then free space fragmentation and file system usage is logged. If the usage reaches defined limit, specified amount of space is randomly released from file system. If the test runs on an SSD, `fstrim` can be optionally called. Furthermore, when all mentioned operations are done, the thread calls `sync`, unpauses `fs-drift` and goes to sleep.

After `fs-drift` ends, fragmentation of used space is logged and image of file system is created using presented tools and archived using `bzip2`. All the generated data and information about environment is archived as well and text file describing the test is generated. These three files are then sent (via `rsync`) to the specified destination.

For better oversight of the tests functionality, its activity diagram is presented as Figure 5.1

Parameters available for `drift_job`:

1. `s`, `sync`, flag to signalise whether or not to send data to server (useful for developing purposes)
2. `M`, mountpoint
3. `d`, disk, device used during test
4. `r`, recipe, parameters to pass to `fs-drift`
5. `t`, tag, string to distinguish different storage configurations
6. `q`, drifttype, string to distinguish different aging configurations

5. AGING THE FILE SYSTEM

- 7. `m`, `maintain`, parameter to specify maximum volume usage and amount to be freed
- 8. `f`, `fstrim`, parameter to specify if `fstrim` should be periodically called

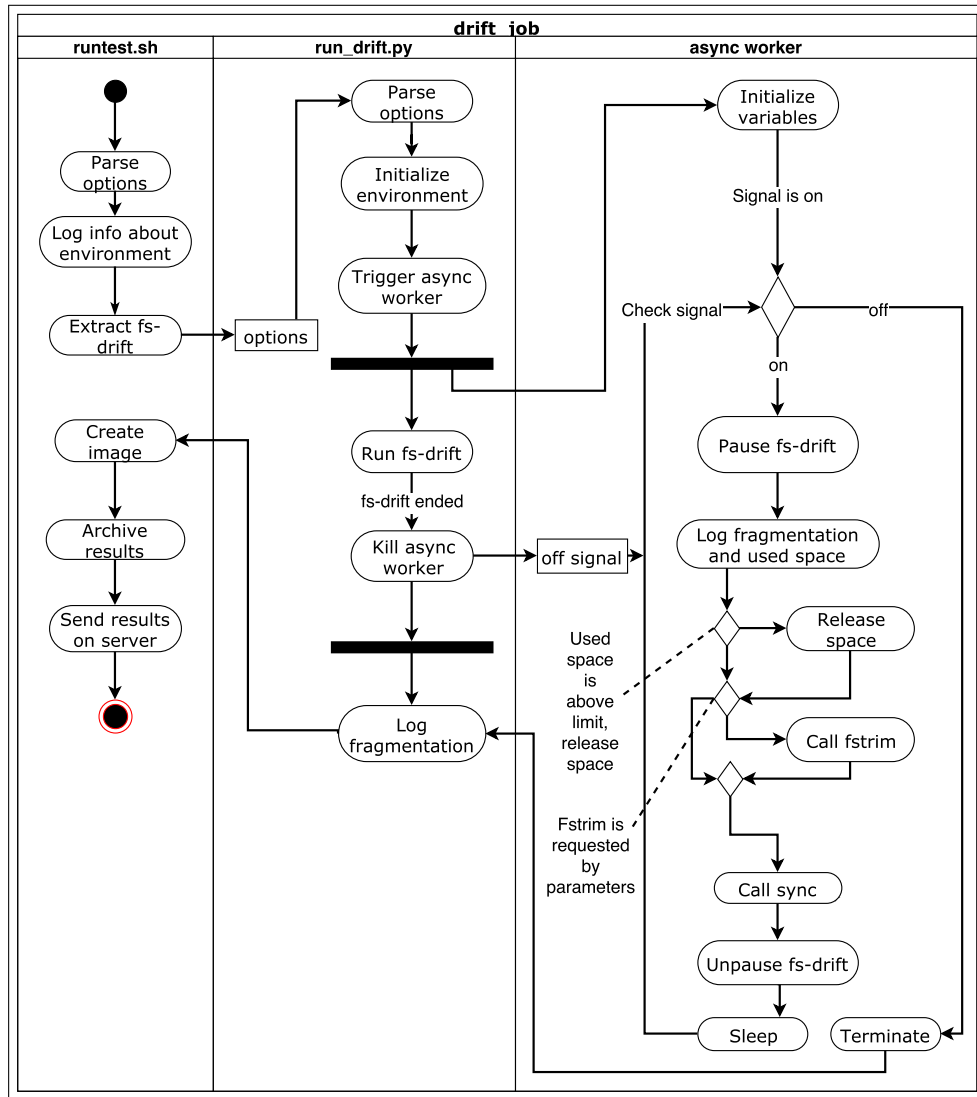


Figure 5.1: Activity diagram of file system aging test

6 Performance testing of aged file system

In this chapter, I describe structure of performance test which use images created by previous workflow. In the first section, I present settings of FIO benchmark for the optimal results and in section two, I describe implementation of this test as beaker task.

6.1 FIO configuration

By default, FIO will create one file for every triggered thread. Furthermore, this file is created and opened before FIO begins issuing IO requests. This default settings is unfit for file system aging testing, because the subject of interest is among others file allocation.

The first step to make FIO workload more relevant to file system aging is to ask for creation of many files for every thread. The number of files was set in such a way, their individual size was approximately 4 kB. FIO will try to open all the files at once, which results in error for more than 102 files, therefore, maximal amount of open files was specified to 100. Furthermore, to include creation of files and allocation into the performance measurement, `create_on_open` was set, so FIO creates file at the moment of opening, if the file does not exist. Another important factor is distribution of which FIO access the files. Default is set to round-robin, but was reset to gauss, so the distribution of file access is more similar to file system aging test.

Similarly to aging test, FIO should perform `fsync` from time to time. This can be set as amount of write requests after which `fsync` should be performed.

Overall size is set in such a way, it consumes most of space left after loading the image. Same size is used while testing on fresh file system. Block size is set to 4 kB as that is default block size of used file systems.

To ensure stable runtime of FIO test, time-based testing is used with runtime of 10 minutes. It is not expected of FIO to issue total specified size, but to measure the performance effectively.

6.2 Test structure

Performance testing of created images is contained in Beaker task `recipe_fio_aging`. Upon installation of necessary tools (libs, fio), the package finds and downloads corresponding file system image according to obtained parameters. As shown, images are stored compressed, therefore decompression is needed after download. Once these steps are successfully completed, testing phase can begin.

Before every test, initialization is performed by running `sync`, `fstrim` and dropping caches.

At first, performance measurements of fresh file system is done. Test configuration of fresh and aged test is similar, with an exception, that after testing fresh file system, FIO will delete all the created files. This is disabled for aged tests, because post-test fragmentation is gathered afterwards.

After testing fresh file system, image is loaded and mounted. If the mounted file system utilize too much space, some files can be randomly deleted. Afterwards, pre-test fragmentation is logged and environment is initialized. When the test is over, post-test fragmentation is gathered.

The configuration of FIO can be passed as parameter to `recipe_fio_aging` task. Such approach makes this testing workflow as flexible as original benchmark.

For statistical correctness, these tests can run in loops n times. Fresh and aged testing is divided into separate loops.

After last iteration, the results are archived and sent to data-collecting server.

For better oversight of the tests functionality, its activity diagram is presented as Figure 6.1

Parameters available for `recipe_fio_aging`:

1. `sync`, flag to signalise wheather or not to send data to server (usefull for developing purposes)
2. `numjobs`, number of test repetitions. For statistical stability
3. `mountpoint`
4. `device`
5. `recipe`, parameters to pass to FIO test
6. `tag`, string to distinguish different tests

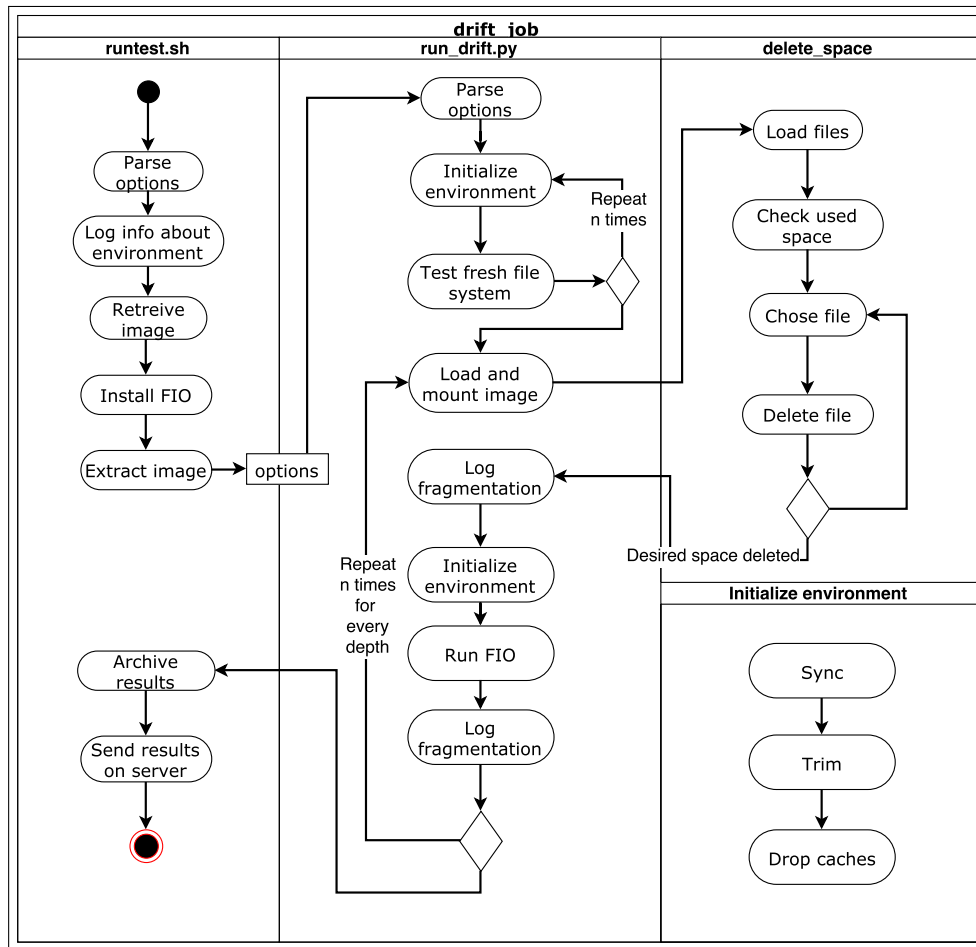


Figure 6.1: Activity diagram of testing of aged file system

7 Results

This chapter presents results of implemented tests. In the first section I describe environment used for testing. In the second section, I describe means of data evaluation. In the third section, I show elementary differences between fresh and aged file systems. In the fourth section, I describe differences between two tested file systems (XFS, Ext4) in terms of aging. In fifth section I demonstrate the effect file system aging have on underlying storage.

7.1 Testing environment

The testing machines were available to me from a pool of systems of Red Hat Beaker environment. For every machine, its model, processor, memory and used storage devices are listed in table 7.1.

The OS installed on all Machines were RHEL-7.3 with kernel 3.10.0-514.el7.x86_64. Tuned profile was set to performance. More information about installed packages can be found in Example B.2.

7. RESULTS

Machine 1	
Model	Lenovo System x3250 M6
Processor	Intel Xeon E3-1230 v5
Clock speed	3.40 GHz (4 cores)
Memory	16 384 MB
Storage	
Device	HP Proliant HardDrive
Interface	SAS
Capacity	600 GB
Machine 2	
Model	Lenovo System x3250 M6
Processor	Intel Xeon E3-1230 v5
Clock speed	3.40 GHz (4 cores)
Memory	16 384 MB
Storage	
Device	SSD
Interface	SATA
Capacity	120 GB
Machine 3	
Model	IBM x3650 System M3
Processor	Intel Xeon X5650 v2
Clock speed	2.67 GHz (6 cores)
Memory	12 288 MB
Storage	
Device	ServeRAID M5015
Interface	SAS
Capacity	300 GB

7.2 Data processing

Accounting for great amount of data generated by the tests, some kind of automatic processing needs to be implemented to make evaluation easier for humans. Therefore, I developed set of scripts which can compare results of two different runs of implemented test. The output of those script is human-readable HTML report. The report displays information about testing environment and different charts described bellow. All the reports generated from the tests can be found in Appendix A. The scripts used to generate reports are part of this thesis in form of electronic appendix

7.2.1 Fragmentation

During the tests, two types of fragmentation were recorded, namely fragmentation of used space and fragmentation of free space.

To find out fragmentation of used space, `xfs_io fiemap` was used. This tool can display physical mapping of any file, using 512 B blocks. Therefore, if run on every file in file system, histogram of fragments of used space can be computed. Files created by fs-drift which are smaller then file system blocksize are nevertheless mapped on one block of file system.

Generally, fragmentation of free space is easier to obtain, since this information is stored in the metadata of file system. However, the way of reading the information differs through implementations of file systems. In Ext4, the tool `e2freefrag` displays histogram of free space of Ext* file system. When looking for this information in XFS, at first user has to find number of allocation groups in the file system instance. Then, for every allocation group, histogram of free space can be obtained using `xfs_db`.

As stated, in the aging test, fragmentation of free space is logged periodically, allowing for an analysis of how fragmentation evolves in time. For the purpose of visualising this evolution, 3D charts are introduced in the final reports.

7.2.2 Evaluation of data generated by fs-drift

The main researched output of fs-drift is an *evolution* of latency of different operations. This property is displayed as a chart with elapsed time on X axis and measured latency on Y axis. Since the data are quite noisy, interpolation and filtering using Savitzky-Golay filter¹ was used to display smooth curve. Further analysis of obtained data should be done in the future to understand behavior of aged file systems even better. Such deep analysis, however, is beyond scope of this thesis.

7.3 Testing on images of aged file system

By testing using second implemented test, no meaningful results were obtained. The observed throughput seemed to be the same across runs on all generated images. The conclusion of this testing is, that it simply does not work as intended. As for testing workflow, images are successfully downloaded, loaded on device and mounted. Afterwards, FIO test is successfully deployed. This can be confirmed from logs generated during the test.

One cause of this problem can be that FIO is simply not suitable for revealing differences between fresh and aged file systems. Another cause of the problem can be in implementation of FIO itself. If there is some bug that causes FIO not to operate as specified by parameters, e.g. FIO will pre-allocate files before performance measuring, the test can output results similar as observed.

Modifying or debugging the benchmark can be a subject of future related work, however such extensive research is beyond scope of this thesis.

7.4 Performance of aged file system

The file system aging tool successfully induced fragmentation in all performed tests. From the data generated by aging tool, it is apparent, that the aging process may negatively affect performance of file sys-

1. Savitzky-Golay filter is a digital filter used for smoothing noisy data. It is implemented as part of open source scientific tools for Python, SciPy [18].

tems. Performance degradation varies with used storage, file system implementation and maintainance techniques (e.g. `fstrim`).

On Figure A.1, we can see evolution of fragmentation of free space gathered while testing at medium file system utilisation (80%). From the Figure A.5, we can see that higher file system utilisation (99%) induces tremendous fragmentation of free space.

Despite this occurrence, it is remarkable, that fragmentation of used space is very similar accross these two test runs. On Figure A.2 and A.6, we can see, that both file systems have about 92% of files optimally allocated. Considering amount of fragments of free space differs in orders of tens of thousands, it is remarkable, that amount of non-contiguous extents of free files differs only by 33%

On Table 7.1, we can see overall raise of latency in regard of highly utilised file system.

Furthermore, sequential read and append display signs of slight progressive performance degradation through test. Furthermore, operation truncate and create shows sign of performance degradation only on high utilised file systems. Figure 7.1 displays latency growth of operations truncate and read on high utilised file system.

7.5 Differences between XFS and EXT4

From the test conducted on highly utilised HDD, we can conclude that XFS fights free space fragmentation much better. On a Figure A.5, which shows last 6 hours of testing high utilisation of XFS on HDD, we can see relatively low amount of free space extents. The peaks in number of extents are caused by random space releasing conducted by aging test, which can be confirmed by examining correspoding untillisation evolution chart (Figure A.6). After every peak occurrence, it is clear that number of small extents is progressively lower until the next peak. It is clear that XFS succesfully defragments free space between the volume releasing iterations. On the other hand, while examining evolution of free space fragmentation of EXT4 (Figure A.9, last 6 hours of the test), we can see that after volume releasing, the amount of small extents of free space remain the same. Furthermore, through most of the testing time EXT4 contained about 20 thousands of extents of free space, which is significantly more than XFS.

7. RESULTS

	80%	99%	
random write	11.40 ms	11.84 ms	median
	16.23 ms	17.86 ms	mean
truncate	8.62 ms	8.93 ms	median
	10.59 ms	11.66 ms	mean
sequential read	8.03 ms	8.54 ms	median
	11.20 ms	14.30 ms	mean
create	11.73 ms	11.98 ms	median
	12.57 ms	16.49 ms	mean
random read	6.42 ms	6.56 ms	median
	7.28 ms	8.05 ms	mean
append	17.66 ms	17.93 ms	median
	22.00 ms	27.30 ms	mean
delete	0.10 ms	0.13 ms	median
	3.63 ms	4.26 ms	mean

Table 7.1: Comparison of latencies of medium and high utilisation

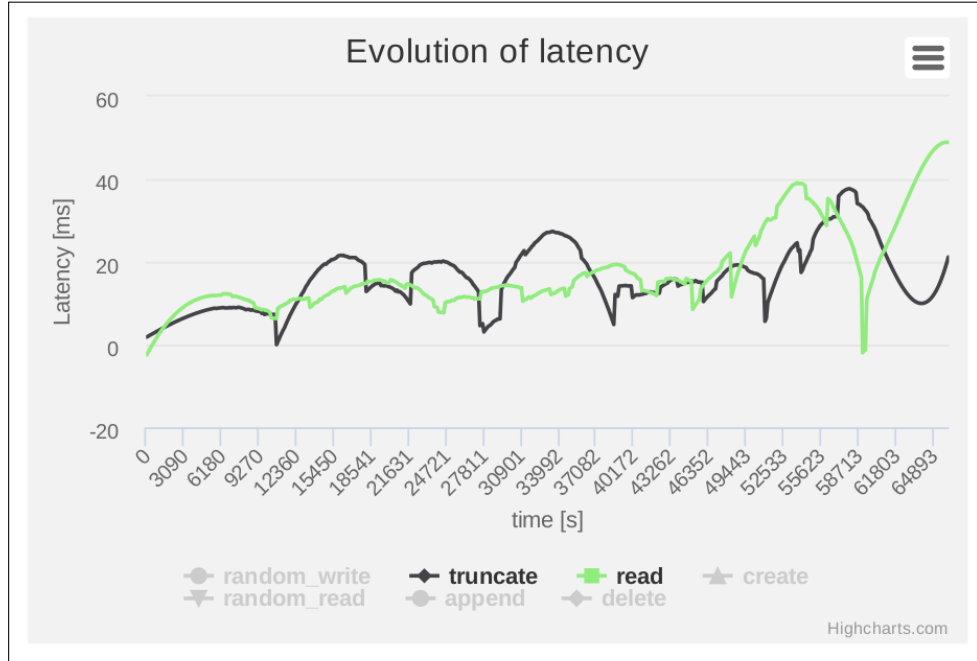


Figure 7.1: Latency growth induced by aging test

Such free space fragmentation can have significant impact on overall file system performance. This can be observed in spite of block allocation operations such as create and append, as we can see in Table 7.2.

When testing with SSD, XFS and EXT4 have very similar disk layouts in spite of free and used space fragmentation. After maximal disk utilisation is reached, both file systems have up to 50 000 fragments of free space (Figure A.13 and Figure A.17). At the end of the test, both file systems had 98% of files optimally allocated. The only difference when considering block layout seems to be in size distribution of extents. As can be seen on Figure A.14 and Figure A.18, EXT4 has more smaller fragments than XFS. Since continuity of relevant data may play role even while using SSD, this could mean potential performance penalty.

In table 7.2, we can see median and mean of all operations through the whole test. It is clear that all the operations except delete and truncate are slightly faster if performed by XFS. This small difference can be result of lesser used space fragmentation in XFS.

7. RESULTS

	SSD		HDD		
	XFS	EXT4	XFS	EXT4	
random write	0.434 ms	0.594 ms	11.84 ms	11.61 ms	median
	1.40 ms	3.05 ms	17.86 ms	19.55 ms	mean
truncate	0.70ms	0.32 ms	8.93 ms	5.40 ms	median
	1.48 ms	1.35 ms	11.66 ms	5.82 ms	mean
sequential read	1.37 ms	1.56 ms	8.54 ms	7.89 ms	median
	3.24 ms	3.47 ms	14.30 ms	10.99 ms	mean
create	0.38 ms	0.40 ms	11.98 ms	18.20 ms	median
	4.02 ms	3.84 ms	16.49 ms	19.59 ms	mean
random read	0.3 ms	0.46 ms	6.56 ms	5.77 ms	median
	1.32 ms	1.37 ms	8.048 ms	6.22 ms	mean
append	1.49 ms	1.76 ms	17.93 ms	23.90 ms	median
	6.46 ms	6.65 ms	27.29 ms	28.69 ms	mean
delete	0.097 ms	0.049 ms	0.128 ms	0.093 ms	median
	0.701 ms	0.346 ms	4.26 ms	1.14 ms	mean

Table 7.2: Comparison of latencies gathered during testing with XFS and EXT4 on SSD and HDD

	HDD	SSD	
random write	11.84 ms	0.43 ms	median
	17.86 ms	1.41 ms	mean
truncate	8.93 ms	0.70 ms	median
	11.67 ms	1.47 ms	mean
sequential read	8.54 ms	1.37 ms	median
	14.30 ms	3.24 ms	mean
create	11.98 ms	0.33 ms	median
	16.49 ms	4.02 ms	mean
random read	6.56 ms	0.29 ms	median
	8.04 ms	1.32 ms	mean
append	17.93 ms	1.49 ms	median
	27.29 ms	6.47 ms	mean
delete	0.128 ms	0.097 ms	median
	4.26 ms	0.70 ms	mean

Table 7.3: Comparison of latencies of gathered during aging test on HDD and SSD

7.6 Performance accross different storage devices

Underlying storage has significant impact on file system performance. As mentioned, SSDs lack of moving parts allows the file system to perform IOs faster. Even when connected through a slower (SATA) interface than HDD (SAS), SSD issues operations much faster, as we can see in Table 7.3. Despite the difference in overall speed of the devices, underlying storage technology have apparently no effect on fragmentation, because file systems deploy the same defragment strategies in both cases. However, latency grow observed on HDD, as mentioned in Section 7.3 was not observed when testing on SSD.

7.7 Testing with `fstrim`

All the mentioned results generated by tests conducted on SSD devices had regular trimming scheduled as part of the test. However, experimental test runs without trimming were also conducted.

From the tests, it is apparent, that regular running of `fstrim` have indeed beneficial effects on file system performance. The most affected operation was create when testing the XFS. Apparent latency growth of create can be seen in Figure 7.2. Overall difference in latency can be found in Table 7.4.

Trimming should not have any effects on fragmentation of file systems, but a difference can be seen in the overall used space fragmentation. While both file systems (trimmed and not trimmed) have similar percentage of optimally allocated files, file system that was not trimmed contains significantly smaller amount of fragments and larger average fragment than trimmed file system (Figure A.14 and A.22). This could be explained by great latency of untrimmed device, which may have resulted for file system to have more time to schedule written blocks better.

This result is great example of testing long-term performance affecting features by using aging test, as well as importance of regular trimming of SSD devices.

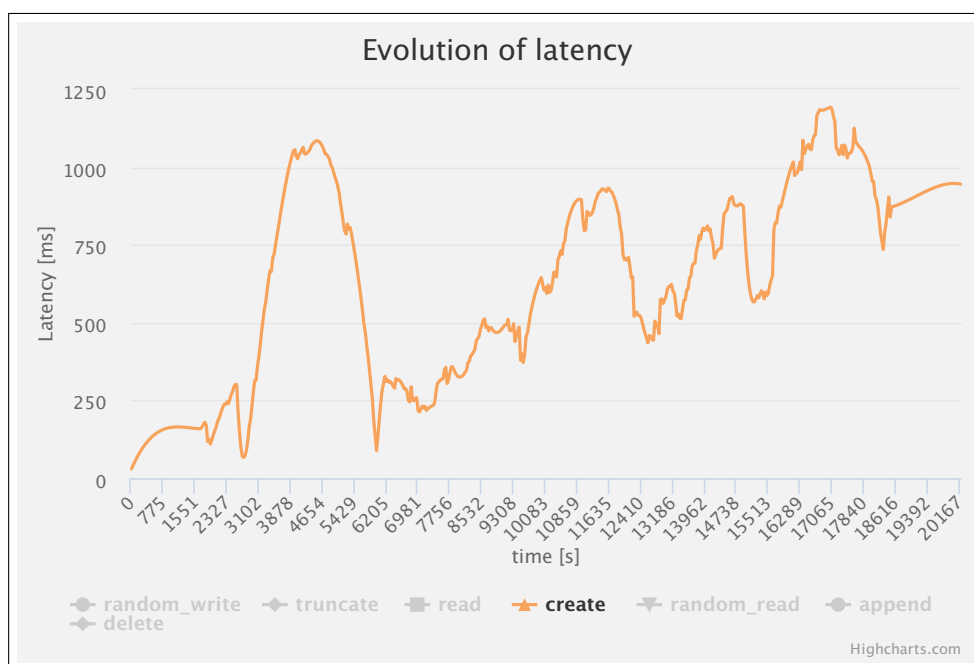


Figure 7.2: Evolution of latency of create operation, in untrimmed file system (XFS)

7. RESULTS

	Trimmed	Not trimmed	
random write	0.43 ms	0.52 ms	median
	1.40 ms	4.99 ms	mean
truncate	0.70 ms	1.05 ms	median
	1.48 ms	3.17 ms	mean
sequential read	1.37 ms	1.85 ms	median
	3.24 ms	4.74 ms	mean
create	0.33 ms	0.34 ms	median
	4.02 ms	6.02 ms	mean
random read	0.30 ms	0.38 ms	median
	1.32 ms	1.99 ms	mean
append	1.49 ms	1.99 ms	median
	6.47 ms	8.33 ms	mean
delete	0.097 ms	0.102 ms	median
	0.70 ms	0.97 ms	mean

Table 7.4: Differences in latency on trimmed and non trimmed file system (XFS)

8 Conclusion

The aim of this thesis was to introduce means of testing of file system aging performance. This was attempted by creating two automated tests, which use existing benchmarks for testing.

The file system aging test uses workload generator to simulate aging. Because test is automated, environment configuration has to be done as well as tool preparation. While aging, the test gathers lots of data and statistics about file system usage, fragmentation levels and performance of conducted IO operations. After the test, metadata images of file systems are automatically created and compressed. All the generated data is compressed too and sent to remote server together with file system image.

This file system aging test successfully induces fragmentation and aging of used file systems. From the generated data, it can be observed, that aging indeed causes performance penalty on file systems. These effects were observed in both tested file systems.

Further aim of this thesis was to compare performance of different aged file systems and effect of used storage technologies on performance.

From the obtained data, it is apparent that with higher utilisation of file systems, performance degrading effect becomes observable. Even while using flash based devices (i.e. SSD), the aging can have degrading effect on performance of some IO operations.

Furthermore, this test was used to demonstrate effect of long term performance affecting tool `fstrim`, displaying another use case of this test besides evaluation of file system implementation itself.

This thesis was aimed on testing so called local file systems with simple configuration, which means, whole drive is presented to file system as a single block device. However, storage usage is evolving into new approaches. Cloud solutions are becoming more and more popular, therefore file systems, which operate over storage clusters are already in production (e.g. CephFS¹). Such file systems have to deal with many new problems like working with metadata, which are stored on remote server, or distribute data evenly across the cluster.

1. Ceph storage cluster file system [19]

8. CONCLUSION

Virtualisation also requires new solutions and additional performance control. With developement of tools like LVM, it si possible to merge multiple devices into huge, flexible arrays. Approaches like thin-provisioning then offer further possibilities of providing storage space to users.

With such growing complexity, file system aging (and software aging generally) can have significant impact on performance of these systems. It is therfore very important to research and understand complex behavior of file system aging.

The automated file system aging test and means of evaluation of generated data are the main results of this thesis. Since the aging test was implemented as a Beaker task, it is compatible with Red Hat Kernel Performance team workflow and was therefore included in testing matrix of Red Hat Enterprise Linux 7.4 and Pegas and upstream kernels. Including this test in regular testing matrix is an important step in expanding ways of examining file systems in new kernel versions. Testing file system aging may reveal and help repair complex problems that can emerge during file systems developement.

The results of this thesis were presented to developers of file systems and open source community as well. Future plans for file system aging are discussed with developers of XFS file system.

Bibliography

- [1] 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.
- [2] Domenico Cotroneo et al. "A Survey of Software Aging and Rejuvenation Studies". In: *J. Emerg. Technol. Comput. Syst.* 10.1 (Jan. 2014), 8:1–8:34. ISSN: 1550-4832. DOI: 10.1145/2539117.
- [3] Keith A. Smith and Margo I. Seltzer. "File System Aging; Increasing the Relevance of File System Benchmarks". In: *SIGMETRICS Perform. Eval. Rev.* 25.1 (June 1997), pp. 203–213. ISSN: 0163-5999. DOI: 10.1145/258623.258689.
- [4] John K. Ousterhout et al. "A Trace-driven Analysis of the UNIX 4.2 BSD File System". In: *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*. SOSP '85. Orcas Island, Washington, USA: ACM, 1985, pp. 15–24. ISBN: 0-89791-174-1. DOI: 10.1145/323647.323631.
- [5] Ningning Zhu et al. "TBBT: Scalable and Accurate Trace Replay for File Server Evaluation". In: *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*. FAST'05. San Francisco, CA: USENIX Association, 2005, pp. 24–24. URL: <http://dl.acm.org/citation.cfm?id=1251028>. 1251052.
- [6] Cheng Ji et al. "An empirical study of file-system fragmentation in mobile storage systems". In: *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. USENIX Association. 2016.
- [7] Ben England. *fs-drift*. <https://github.com/parallel-fs-utils/fs-drift>. 2017.
- [8] Lanyue Lu et al. "A Study of Linux File System Evolution". In: *Proceedings of the 11th USENIX Conference on File and Storage Technologies*. FAST'13. San Jose, CA: USENIX Association, 2013, pp. 31–44. URL: <http://dl.acm.org/citation.cfm?id=2591272>. 2591276.
- [9] Vijayan Prabhakaran et al. "Analysis and Evolution of Journaling File Systems." In: *USENIX Annual Technical Conference, General Track*. 2005, pp. 105–120.

BIBLIOGRAPHY

- [10] Adam Sweeney et al. "Scalability in the XFS File System". In: *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*. ATEC '96. San Diego, CA: USENIX Association, 1996, pp. 1–1. URL: <http://dl.acm.org/citation.cfm?id=1268299.1268300>.
- [11] Avantika Mathur et al. "The new ext4 filesystem: current status and future plans". In: *Proceedings of the Linux Symposium*. Vol. 2. 2007, pp. 21–33.
- [12] Takashi Sato. "ext4 online defragmentation". In: *Proceedings of the Linux Symposium*. Vol. 2. Citeseer. 2007, pp. 179–86.
- [13] Vamsee Kasavajhala. "Solid state drive vs. hard disk drive price and performance study". In: *Proc. Dell Tech. White Paper* (2011), pp. 8–9.
- [14] Xiao-Yu Hu et al. "Write Amplification Analysis in Flash-based Solid State Drives". In: *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. SYSTOR '09. Haifa, Israel: ACM, 2009, 10:1–10:9. ISBN: 978-1-60558-623-6. DOI: 10.1145/1534530.1534544.
- [15] Yuan-Hao Chang et al. "Endurance Enhancement of Flash-memory Storage Systems: An Efficient Static Wear Leveling Design". In: *Proceedings of the 44th Annual Design Automation Conference*. DAC '07. San Diego, California: ACM, 2007, pp. 212–217. ISBN: 978-1-59593-627-1. DOI: 10.1145/1278480.1278533.
- [16] Tasha others Frankie. "A Mathematical Model of the Trim Command in NAND-flash SSDs". In: *Proceedings of the 50th Annual Southeast Regional Conference*. ACM-SE '12. Tuscaloosa, Alabama: ACM, 2012, pp. 59–64. ISBN: 978-1-4503-1203-5. DOI: 10.1145/2184512.2184527.
- [17] Nitin Agrawal et al. "A five-year study of file-system metadata". In: *ACM Transactions on Storage (TOS)* 3.3 (2007), p. 9.
- [18] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. 2001–. URL: <http://www.scipy.org/>.
- [19] Sage A Weil et al. "Ceph: A scalable, high-performance distributed file system". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 307–320.

A Reports

A.1 Test of medium utilisation using XFS on HDD

- System: Machine 1
- OS: RHEL-7.3
- Kernel: 3.10.0-514.el7.x86_64

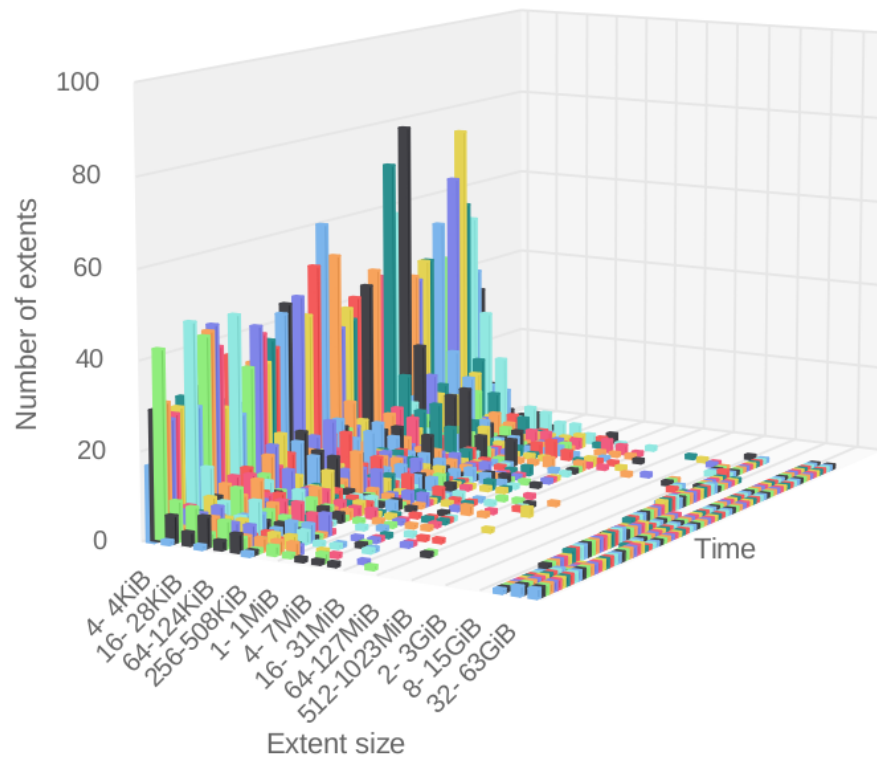


Figure A.1: On this figure we can see tendency of XFS to merge free space blocks into very large extents (up to 63 GiB). Furthermore, low fragmentation can be observed during testing time, with quite large blocks of free space even at the end of the test. This chart displays last 7 hours of the test.

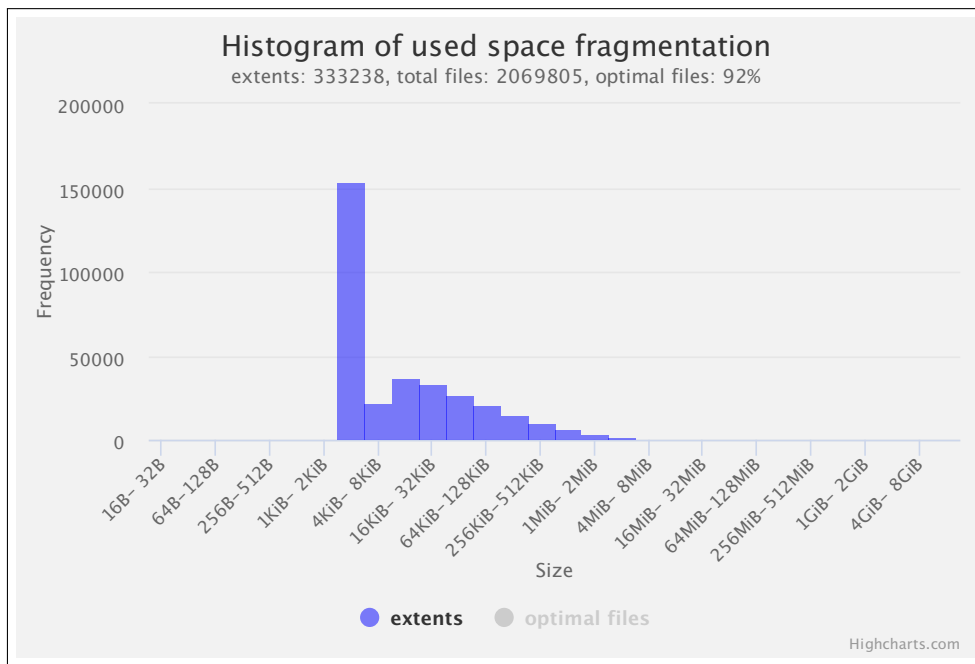


Figure A.2: On this chart, we can see that despite low fragmentation of free space, about 333 thousands of file extents were present in the final block layout. However, 92% of files were still optimally allocated. Furthermore, we can see that most of the created extents are the smallest possible size (4 kB, i.e. file system block size).

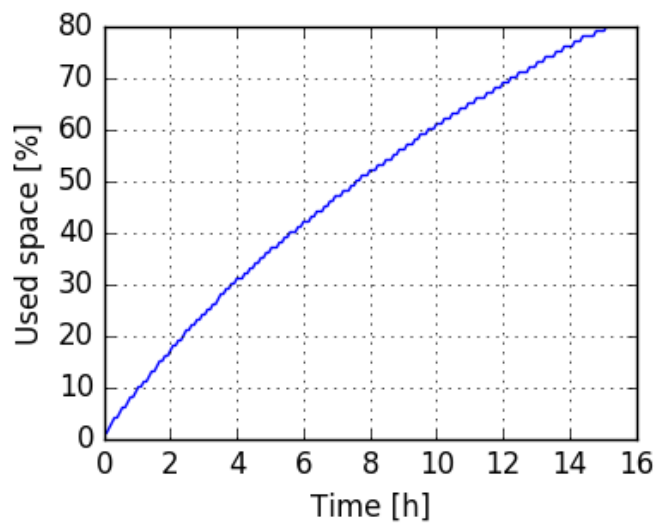


Figure A.3: On this chart, system utilisation over time is displayed. We can see slight curvature of the line, meaning, growth of file system is slowing down. This can be caused by progressively more succesfull delete operation (with more files, there is greatear probability of deleting) and by progressively slower block allocating operations such as create, append and random write.

A. REPORTS

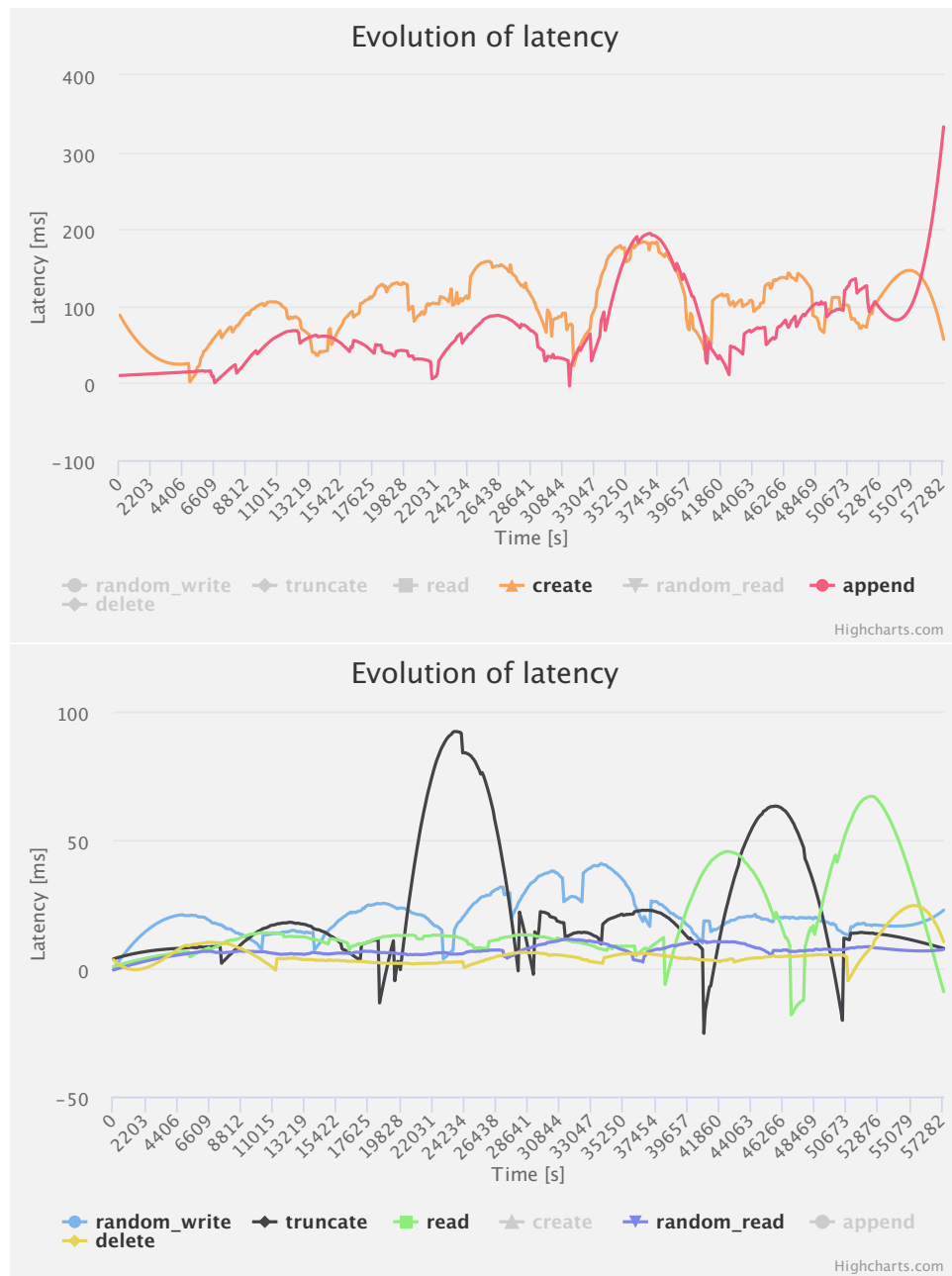


Figure A.4: On these charts, we can see evolution of latencies of conducted IO operations during testing XFS with medium utilisation. Progressively larger fluctuations can be seen in sequential read operation. Latency of operation append seems to be slowly raising in the last hour of test.

A.2 Test of high utilisation using XFS on HDD

- System: Machine 3
- OS: RHEL-7.3
- Kernel: 3.10.0-514.el7.x86_64

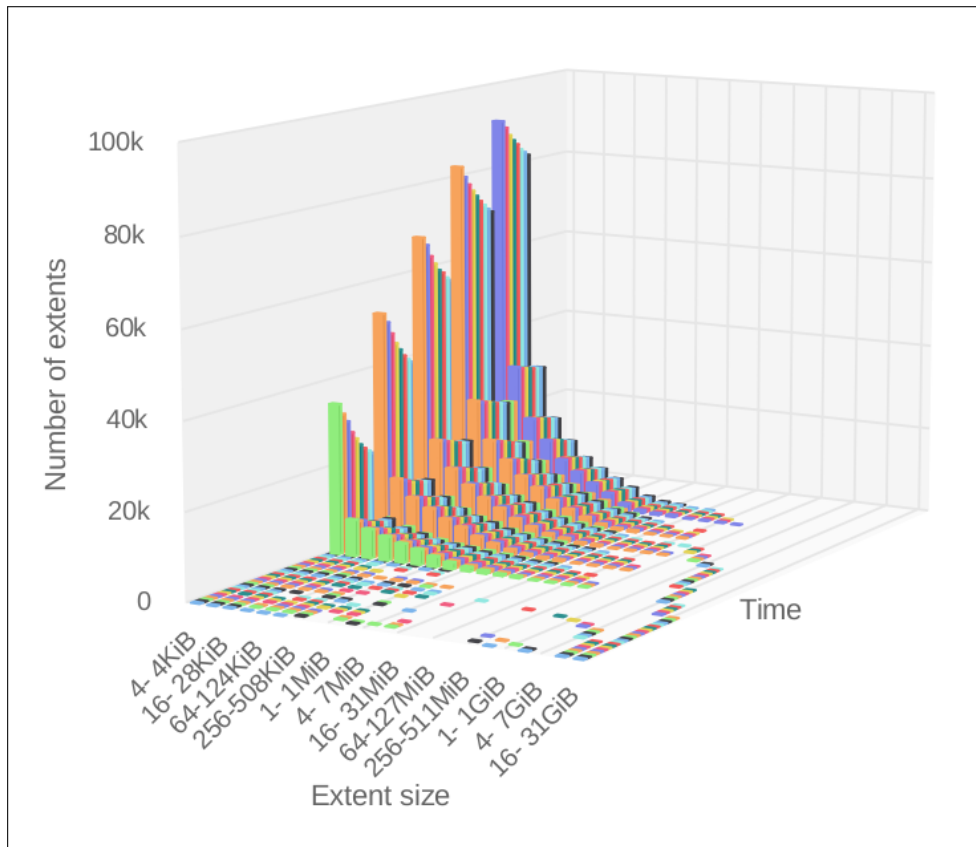


Figure A.5: On this chart, we can see evolution of free space fragmentation in the last 5 hours of the test. When comparing this chart with Figure A.7, we can see that peak in number of extents always appear when the test is randomly deleting space. After that, XFS tries to merge small free space extents into larger ones and successfully defragmenting free space.

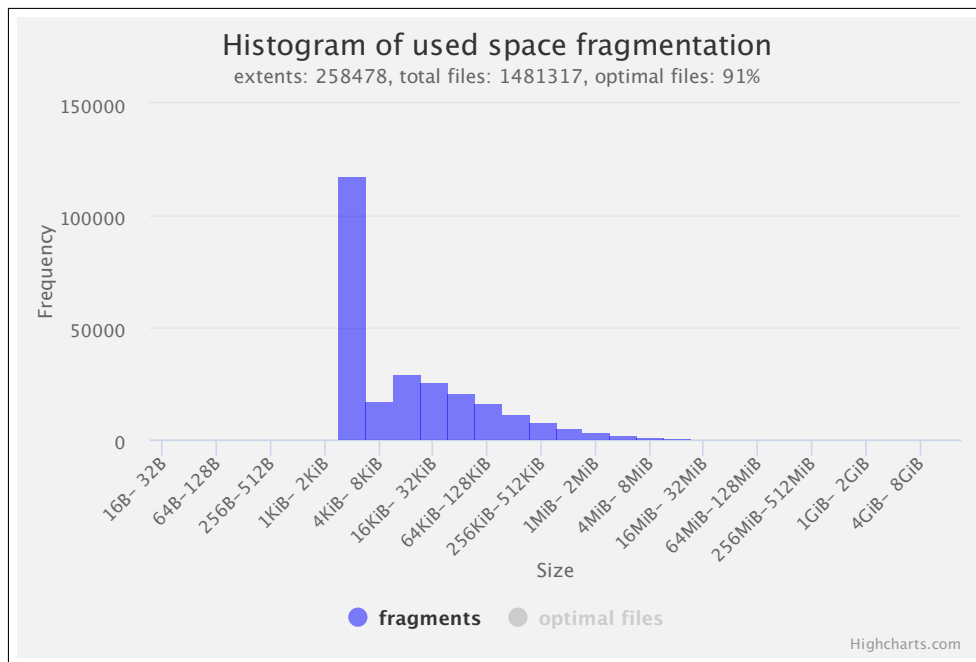


Figure A.6: On this chart, we can see that despite high fragmentation of free space, 91% of files were optimally allocated. Distribution of extents of used space is similar to distribution observed while testing with medium utilisation.

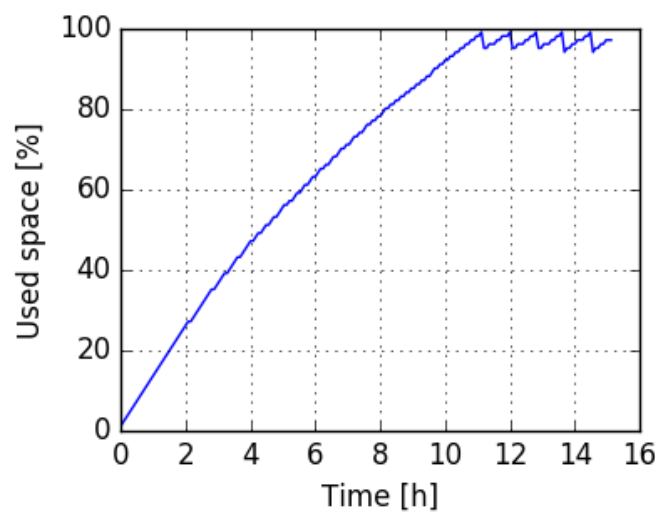


Figure A.7: File system utilisation over time. We can see that deletion of used space was triggered five times. It took the test approximately 10.5 hours to fill available space (300 GB).

A. REPORTS

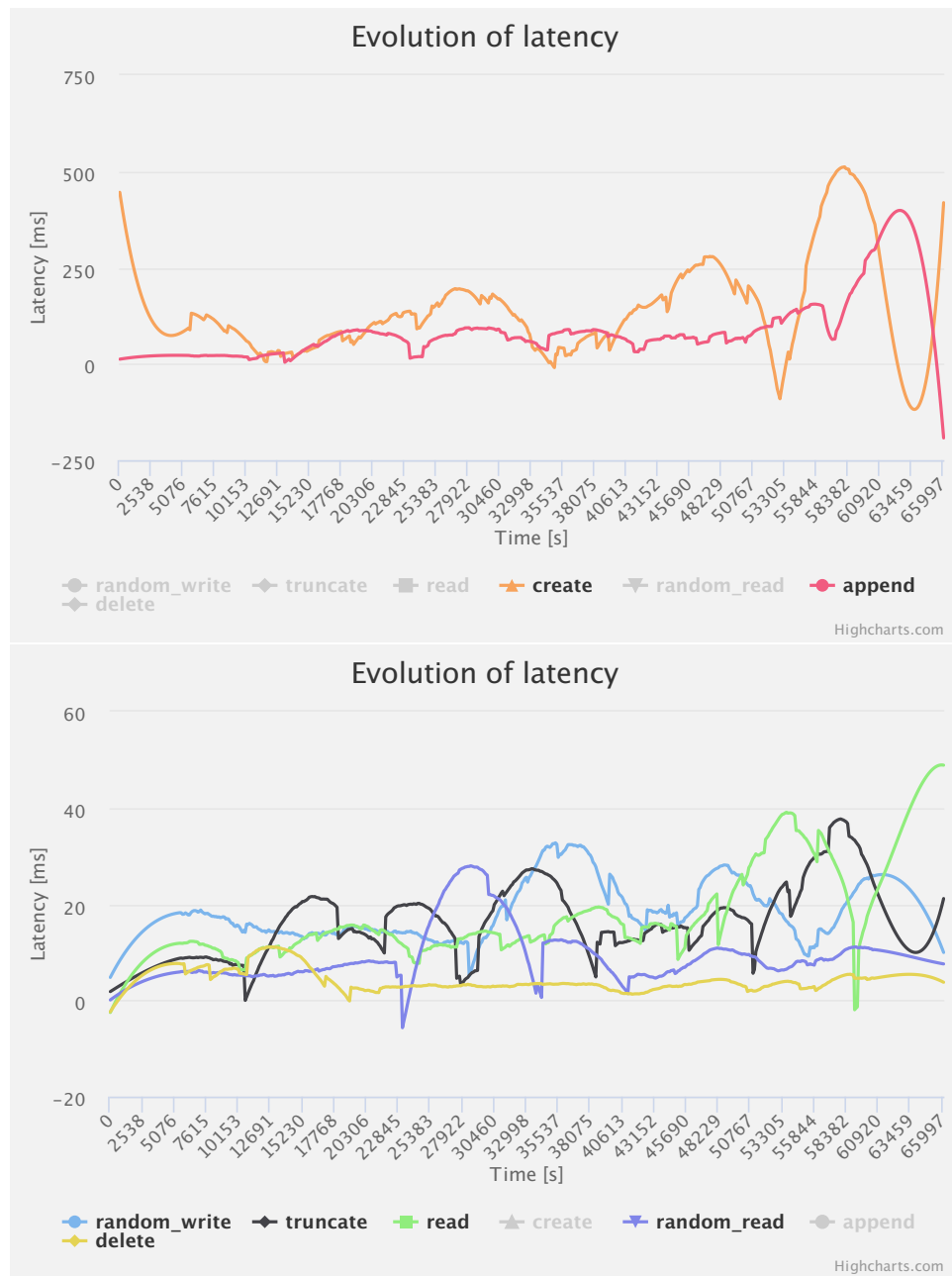


Figure A.8: On these charts, we can see evolution of latencies of conducted IO operations during testing XFS with high utilisation. Progressively larger fluctuations can be seen in create and append operations. Some other operations such as sequential read of truncate are showing progressive growth mainly after file system reached highest utilisation.

A.3 Test of high utilisation using EXT4 on HDD

- System: Machine 3
- OS: RHEL-7.3
- Kernel: 3.10.0-514.el7.x86_64

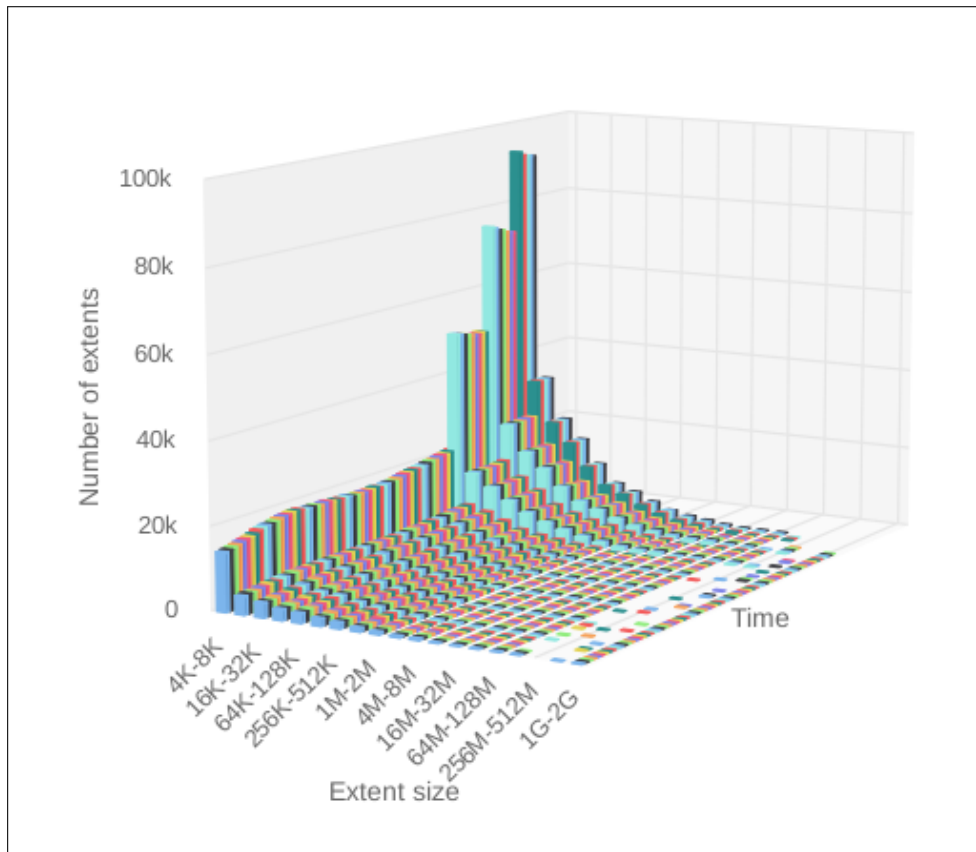


Figure A.9: This chart shows evolution of free space fragmentation in the last 5 hours of the test. When comparing this chart with Figure A.11, we can see that peak in number of extents always appear when the test is randomly deleting space. After that, EXT4 fails to merge small free space extents into larger ones as it could be observed in XFS.

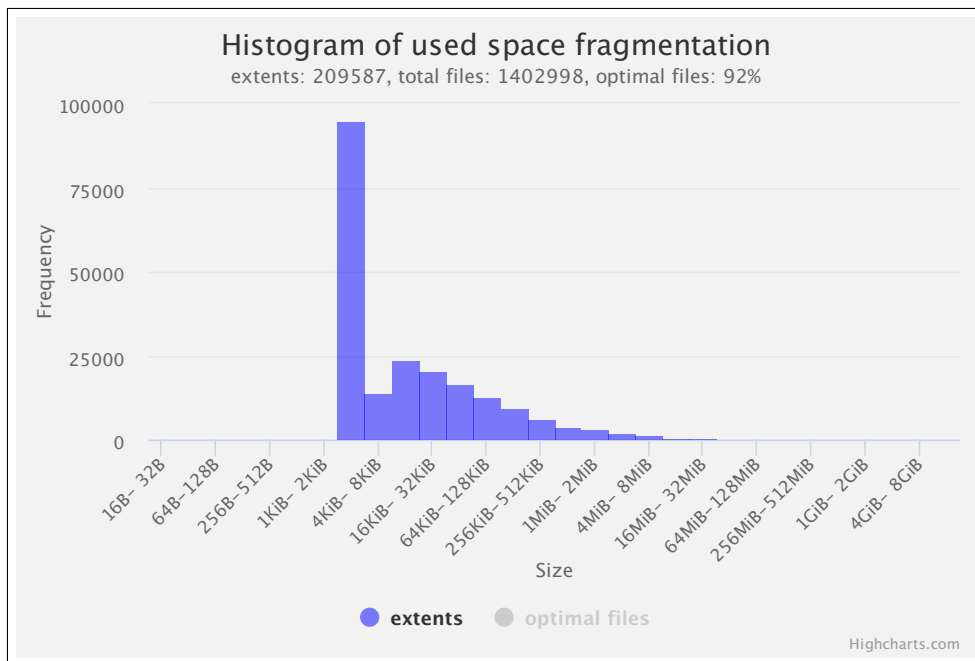


Figure A.10: On this chart, we can see distribution of size of file extents. Despite high fragmentation of free space during the test, the final block layout of file system have around 209 thousands of extents. 92% of files were optimally allocated. Furthermore, we can see that most of the created extents are the smallest possible size (4 kB, i.e. file system block size).

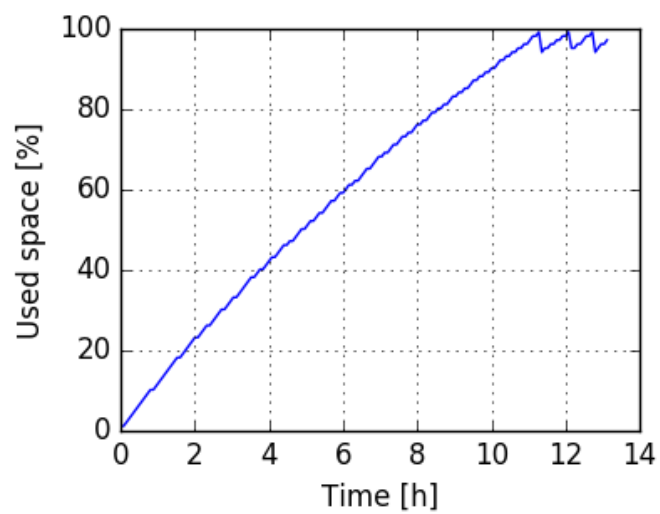


Figure A.11: Chart displays file system utilisation over time. We can see that deletion of used space was triggered 3 times. It took the test approximately 10.5 hours to fill available space (300 GB).

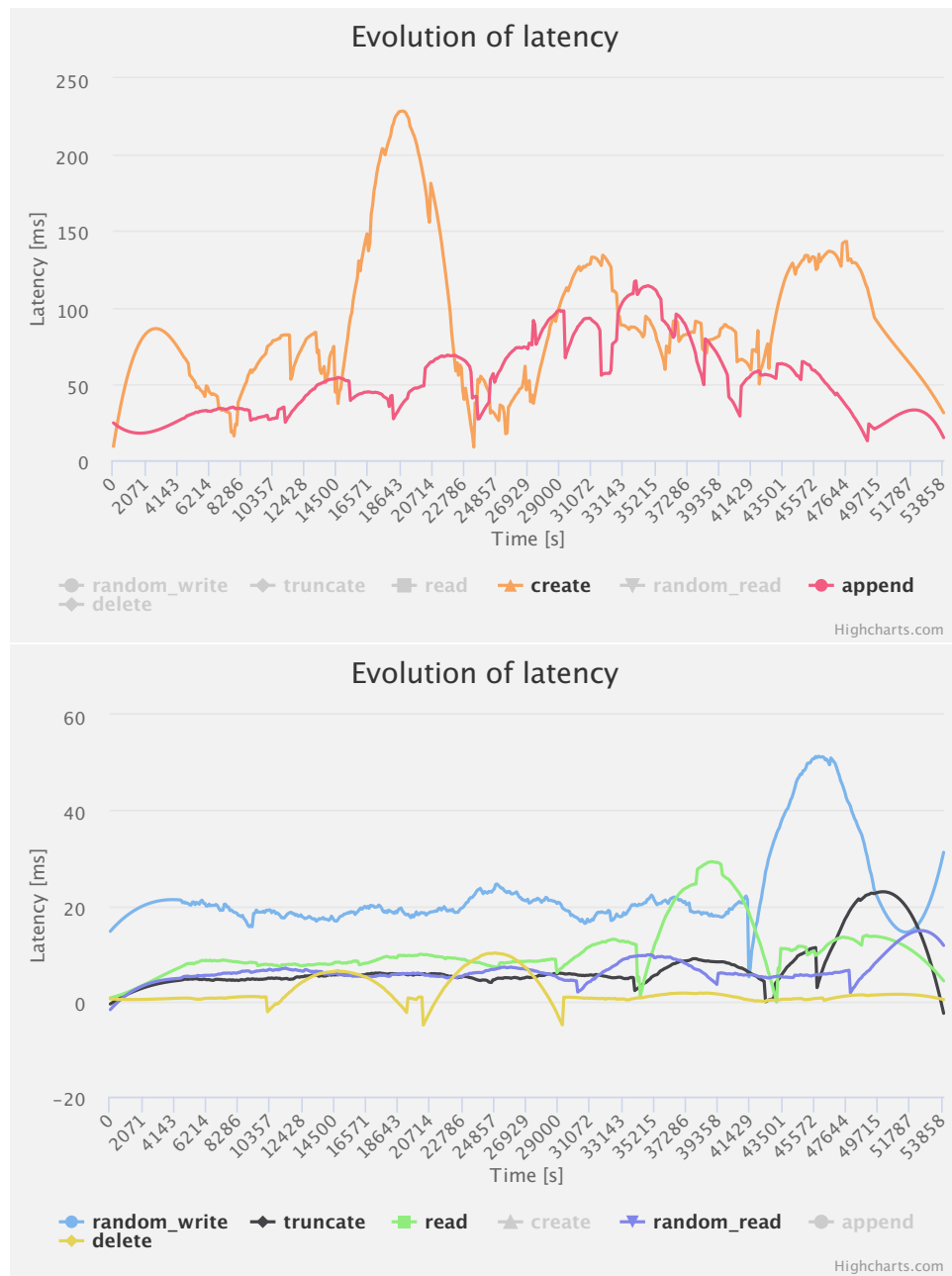


Figure A.12: On these charts, we can see evolution of latencies of conducted IO operations during testing EXT4 with high utilisation. Progressively larger fluctuations can be seen in operations random write, truncate and read.

A.4 Test of XFS on SSD with regular trimming

- System: Machine 2
- OS: RHEL-7.3
- Kernel: 3.10.0-514.el7.x86_64

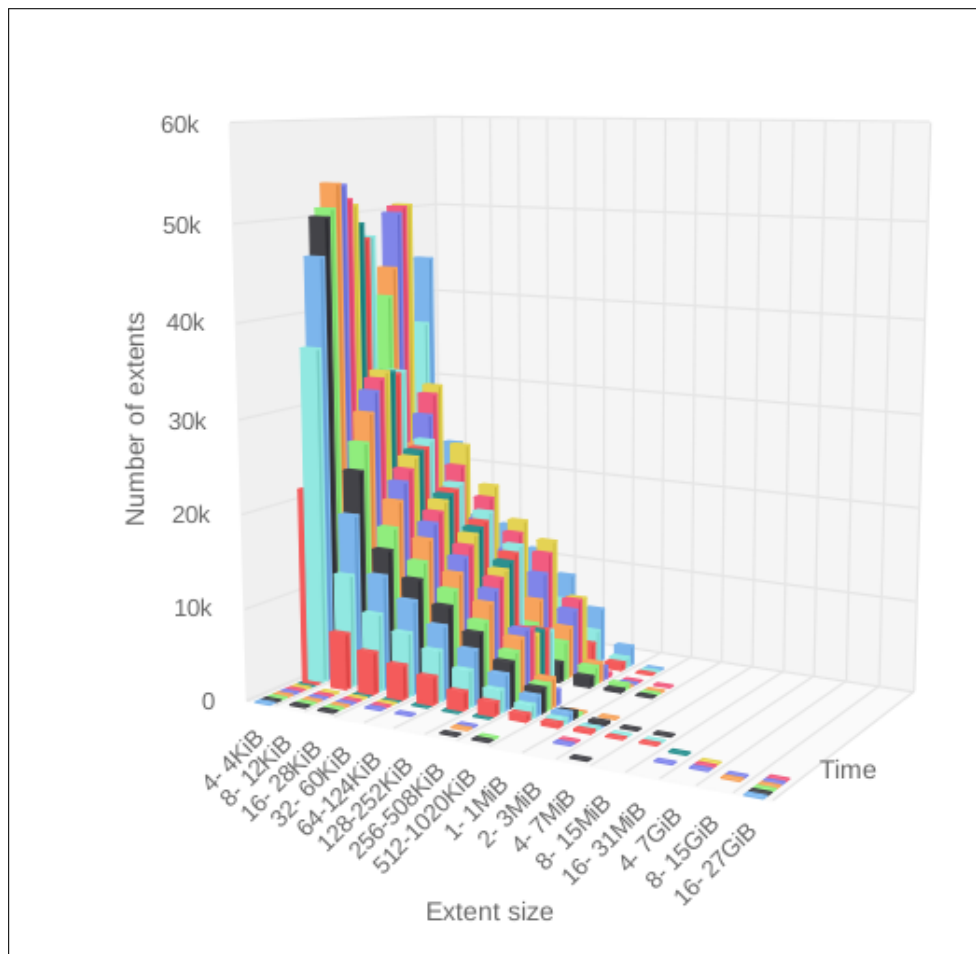


Figure A.13: This chart shows evolution of free space fragmentation through the whole test. When comparing this chart with Figure A.15, we can see large raise of number of extents occurred after the test consumed all the available volume.

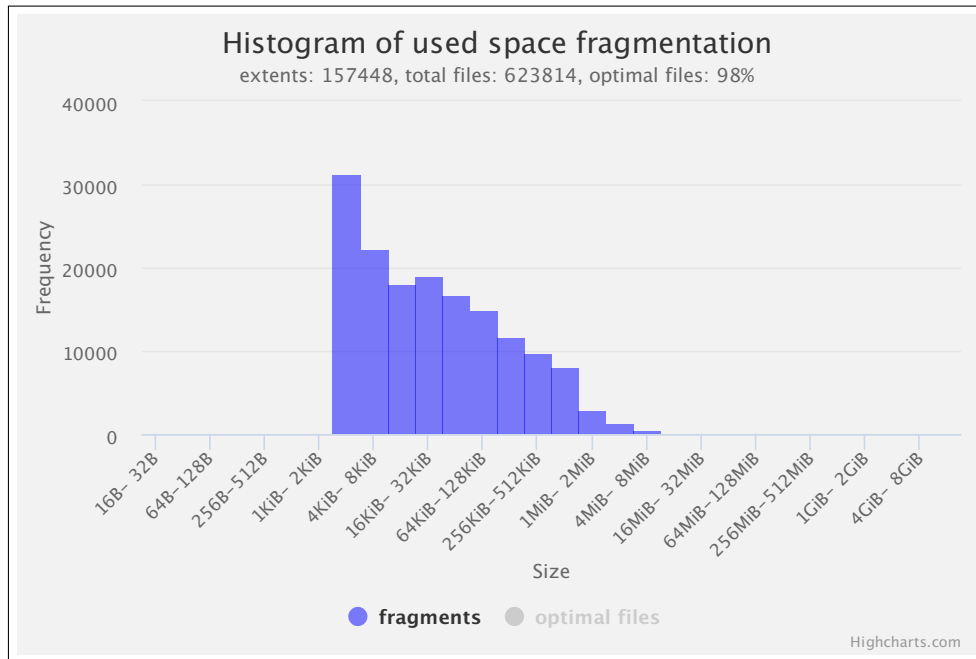


Figure A.14: On this chart, we can see distribution of size of file extents. Despite high fragmentation of free space during the test, the final block layout of file system have around 157 thousands of extents. 98% of files were optimally allocated. Furthermore, we can observe that extent size distribution is similar to file size distribution in the test. Please note that smallest possible extent size in file system is the size of its block size (4 kB by default).

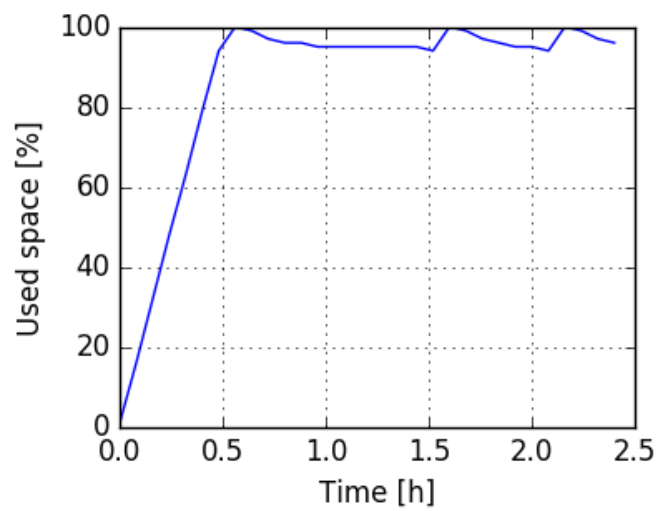


Figure A.15: Chart displays file system utilisation over time. Test filled the available volume (120 GB) in half an hour. It is difficult to say how many volume deleting iterations were triggered, because the test managed to fill released space between usage logging intervals.

A. REPORTS

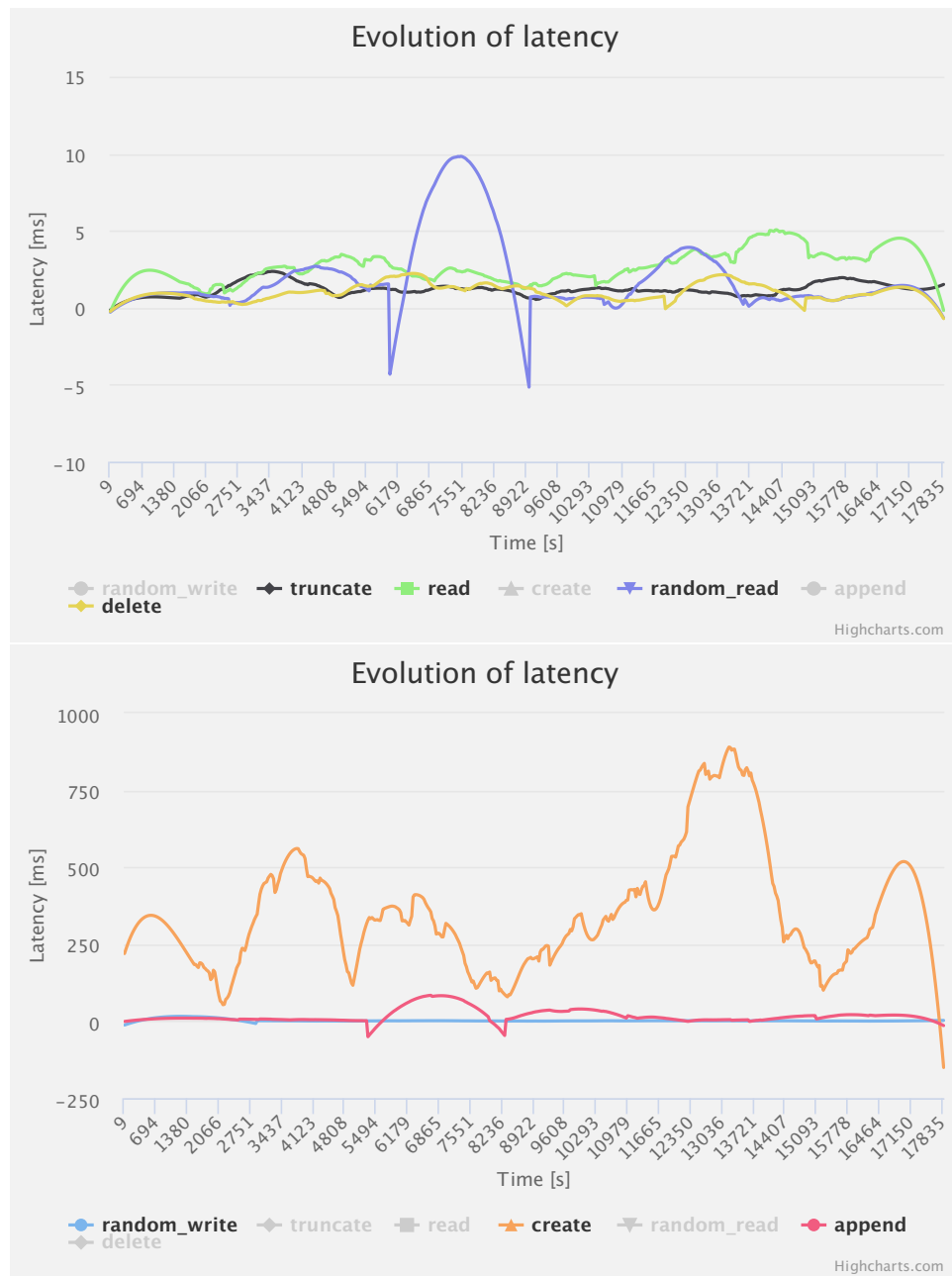


Figure A.16: On these charts, we can see evolution of latencies of conducted IO operations during testing XFS on SSD device with regular trimming performed. Large fluctuations can be seen in create operations. Other operations are not showing significal performance change trend during testing time.

A.5 Test of EXT4 on SSD with regular trimming

- System: Machine 2
- OS: RHEL-7.3
- Kernel: 3.10.0-514.el7.x86_64

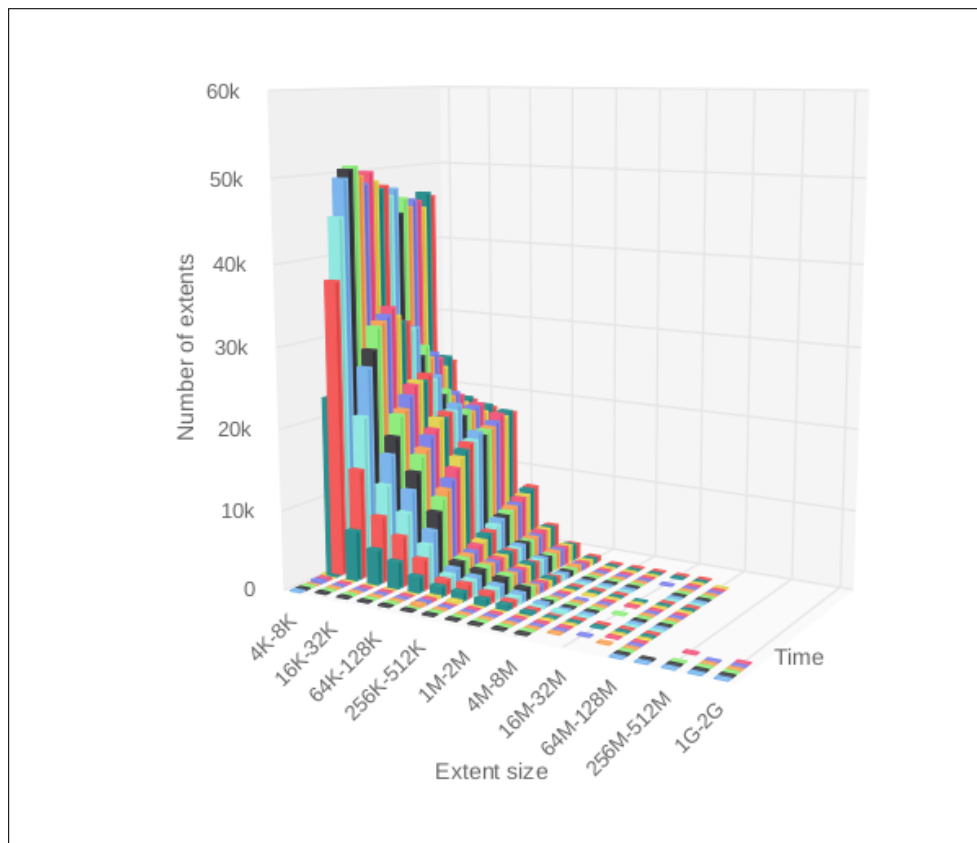


Figure A.17: This chart shows evolution of free space fragmentation through the whole test. When comparing this chart with Figure A.19, we can see large raise of number of extents occurred after the test consumed all the available volume.

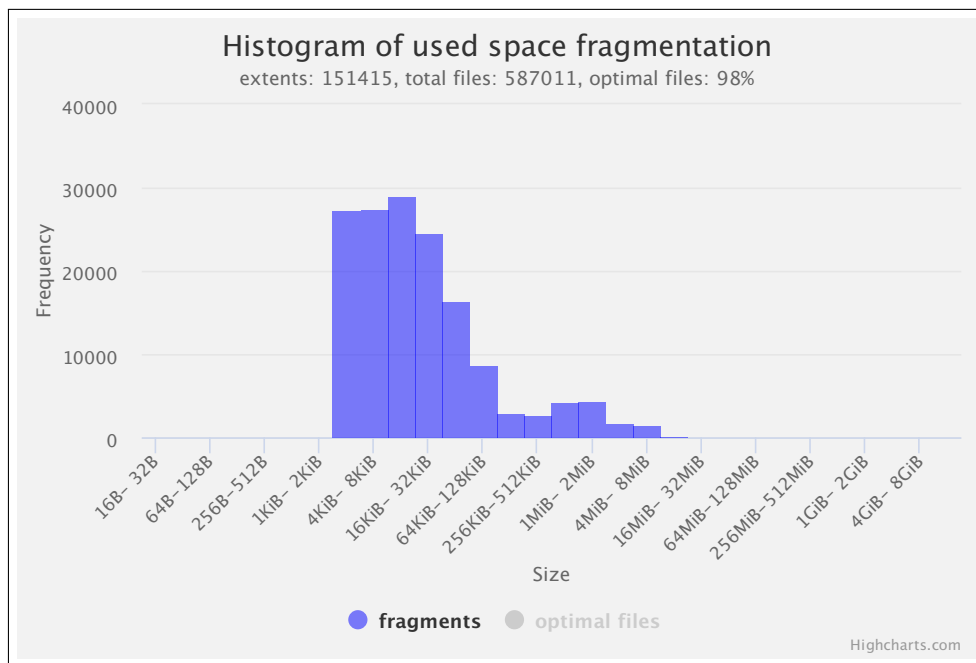


Figure A.18: On this chart, we can see distribution of size of file extents. Despite high fragmentation of free space during the test, the final block layout of file system have around 151 thousands of extents. 98% of files were optimally allocated. Furthermore, we can observe that extent size distribution has lower amount of larger extents (>128 kB) while having similar amounts of smaller extents (4-32 kB).

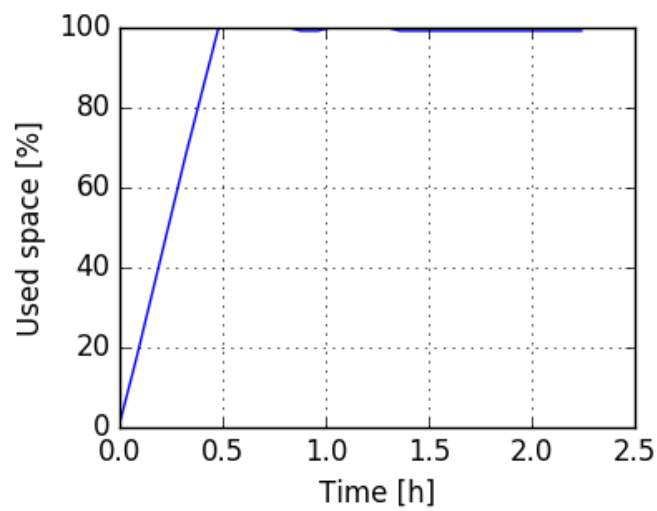


Figure A.19: Chart displays file system utilisation over time. Test filled the available volume (120 GB) in half an hour. It is difficult to say how many volume deleting iterations were triggered, because the test managed to fill released space between usage logging intervals.

A. REPORTS

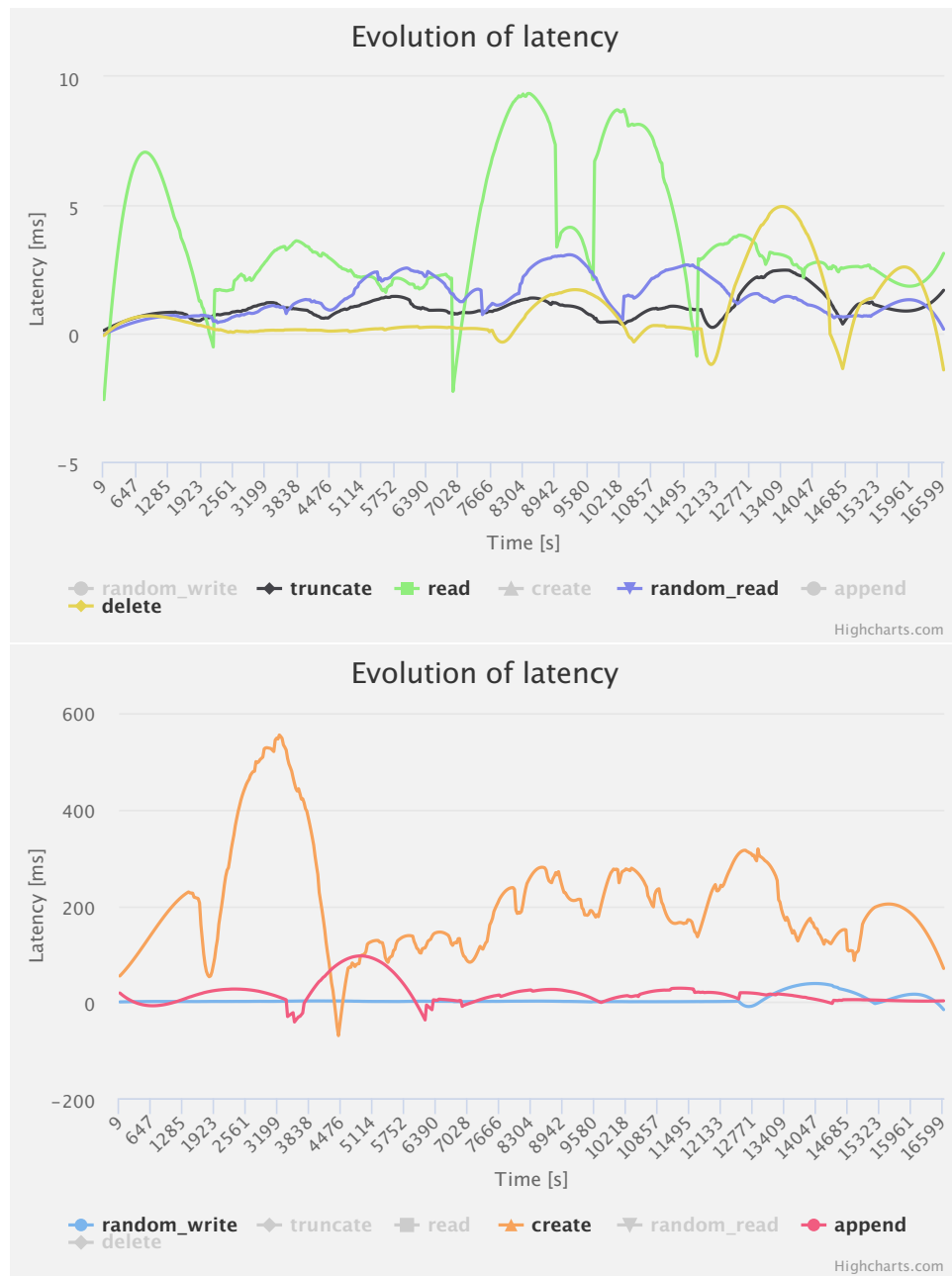


Figure A.20: On these charts, we can see evolution of latencies of conducted IO operations during testing EXT4 on SSD device with regular trimming performed. Some large fluctuations can be seen in read operation, but otherwise not meaningful trend in change of performance can be observed.

A.6 Test of XFS on SSD without regular trimming

- System: Machine 2
- OS: RHEL-7.3
- Kernel: 3.10.0-514.el7.x86_64

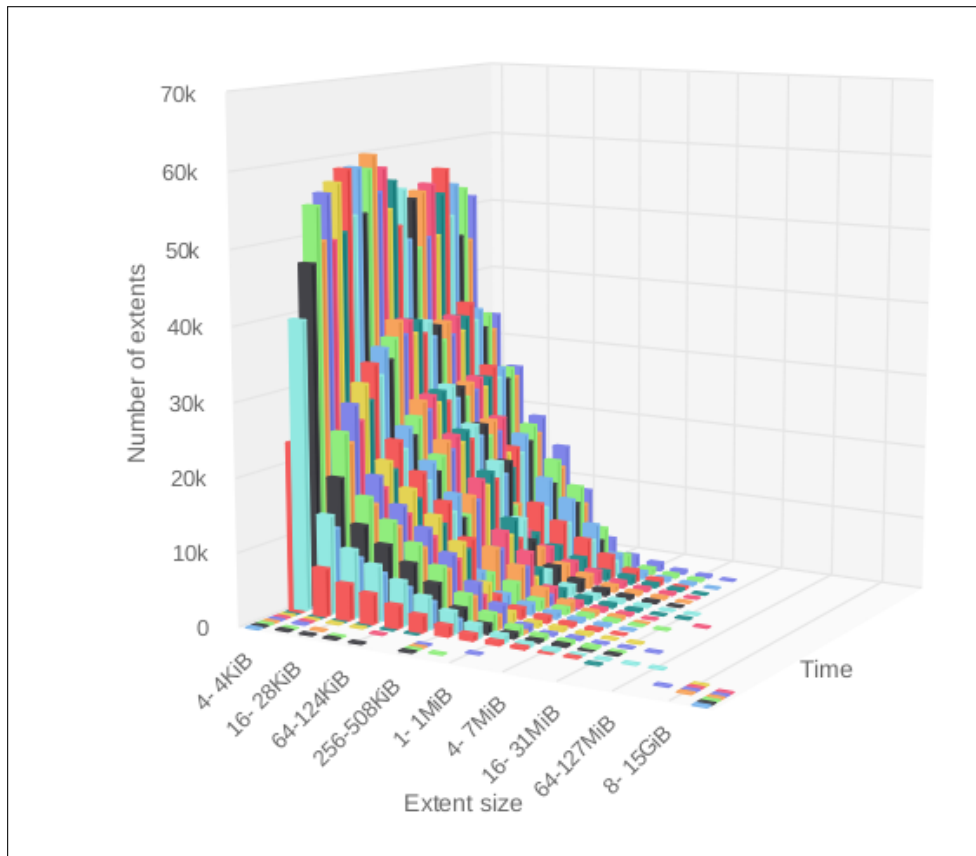


Figure A.21: This chart shows evolution of free space fragmentation through the whole test. When comparing this chart with Figure A.19, we can see large raise of number of extents occurred after the test consumed all the available volume. We can see that untrimmed XFS file system has slightly more free space extents that trimmed XFS file system (FigureA.13)

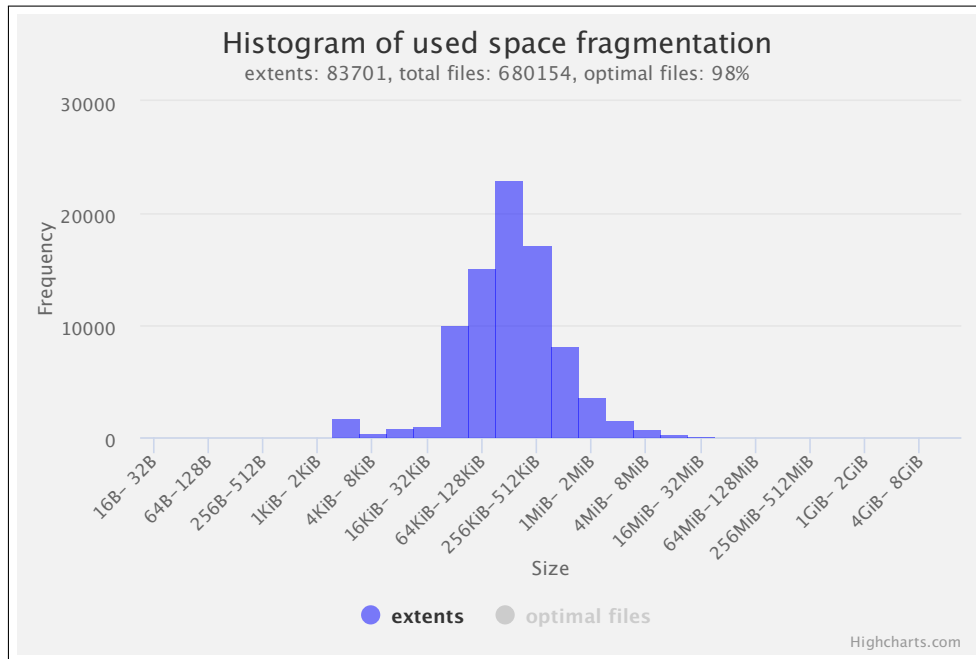


Figure A.22: On this chart, we can see distribution of size of file extents. Despite high fragmentation of free space during the test, the final block layout of file system have around 83 thousands of extents. 98% of files were optimally allocated. Furthermore, we can observe that untrimmed XFS file system has more large file extents than trimmed. This can be caused by underlying device being slow, which in turn gives file system more time to reschedule its requests better.

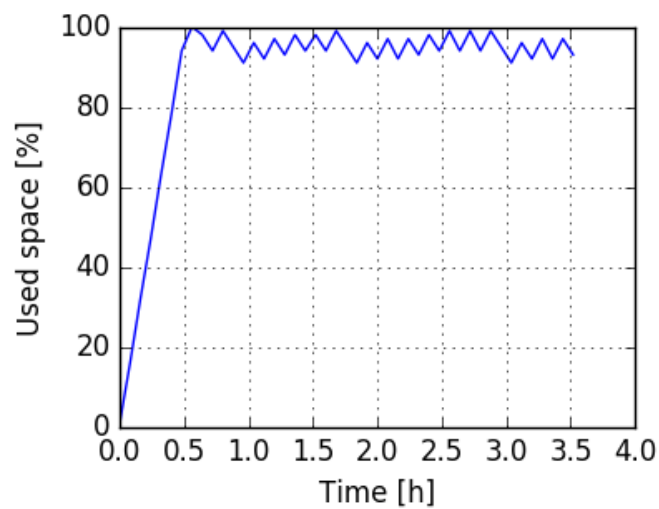


Figure A.23: Chart displays file system utilisation over time. Test filled the available volume (120 GB) in half an hour. Unless on trimmed tests, random file deleting iterations are observable on this graph, because the device was not able to fill volume inbetween data collecting intervals. This is a clear sign that untrimmed file system is significantly slower.

A. REPORTS

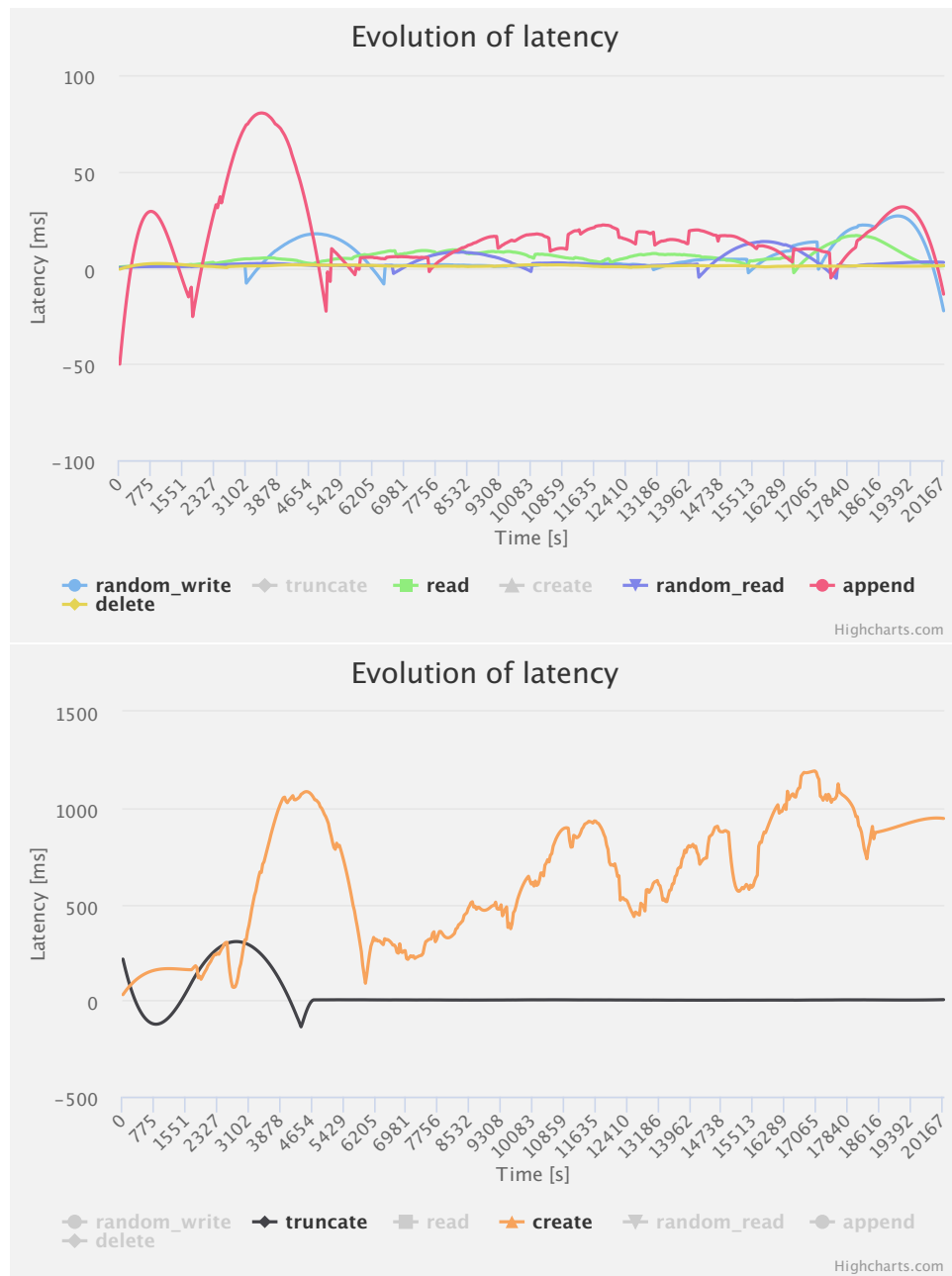


Figure A.24: On these charts, we can see evolution of latencies of conducted IO operations during testing XFS on SSD device without regular trimming performed. Progressive degradation can be observed in create operation. There may be small growth and progressive larger fluctuations in sequential read, random read and random write.

B Examples

Example B.1: Specifying OS to be installed

```
5 <distroRequires>
6   <and>
7     <distro_family op="=" value="RedHatEnterpriseLinux7"/>
8     <distro_variant op="=" value="Server"/>
9     <distro_name op="=" value="RHEL-7.3"/>
10    <distro_arch op="=" value="x86_64"/>
11  </and>
12 </distroRequires>
```

Example B.2: Configuring environment and installing packages using kickstart

```
13 <kickstart>
14 <![CDATA[
15
16 install
17 lang en_US.UTF-8
18 skipx
19 keyboard us
20 rootpw redhat
21 firewall --disabled
22 authconfig --enableshadow --enablemd5
23 selinux --enforcing
24 timezone --utc Europe/Prague
25
26 bootloader --location=mbr --driveorder=sda
27 zerombr
28 clearpart --all --initlabel --drives=sda
29 part /boot --fstype=ext2 --size=200 --asprimary --label=BOOT --ondisk=
   sda
30 part /mnt/tests --fstype=ext4 --size=40960 --asprimary --label=MNT --
   ondisk=sda
31 part / --fstype=ext4 --size=1 --grow --asprimary --label=ROOT --
   ondisk=sda
32 reboot
33 %packages --excludedocs --ignoremissing --nobase
34 @core
35 wget
36 python
37 perl-devel
38 parted
39 cpuspeed
40 perl
41 dhcpv6-client
42 dhclient
43 yum
44 yum-rhn-plugin
45 yum-security
46 yum-updatesd
47 openssh-server
```

B. EXAMPLES

```
48 openssh-clients
49 bc
50 screen
51 nfs-utils
52 seekwatcher
53 sysstat
54 xfsprogs
55 e2fsprogs
56 hdparm
57 sdparm
58 gcc
59 tuned
60 cpufrequtils
61 cryptsetup-luks
62 vim-enhanced
63 rsync
64 lvm2
65 smartmontools
66 git
67 iotop
68 %end
69 ]]>
70 </kickstart>
```

Example B.3: Configuring single device and file system EXT4 using storage generator

```
71 <task name="/kernel_fsperf/storage_generator" role="STANDALONE">
72   <params>
73     <param name="TEST_PARAM_STORAGE_GENERATOR" value="-s create
74               -f ext4 -t single -m /RHTSspareLUN1 -d /dev/sdc -T 1
75               SASHDD_ext4"/>
76   </params>
77 </task>
```