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

This is the acknowledgement for my thesis, which can span multiple paragraphs.

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

List of Tables

List of Figures

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 present 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 is 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 by 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 released from file system, so the performance test has space to test on. After environment initalization (`fstrim`, `sync`), the performance test can begin. Testing of performance 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 then 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 snapshoting 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 fragmenetation 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-drift [7], 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 controled by workload table. In addition, this tool offers different probability distributions of file access, making it easier to mimic behavior of natural user. Furthermore, whole process is highly configurable, making it possible to simulate various types of file system usage. Such qualities

predispose fs-drift to be capable of conduction of more realistic results than mentioned attempts.

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 retreive 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-contigous 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], XFS [9] and Ext4 [10], which are also main Red Hat supported file systems. These file systems belong to the group of file systems called journaling file systems.

Journaling file system keeps a structure called journal, which is a buffer of changes not yet commited 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 again.

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 euqally 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, substracts 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 [11].

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 retreived 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 performence 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 cirquit to store and retreive data. SSD has no moving parts, therefore the data access is purely electronic, which results in lower access time and latency than HDD.

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 to one page (4-16 kB), but erasure have to be done to 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 submited back on available part of the drive. The original page is then marked as discarded. This effect is known as write amplification, and is computed as amount of bytes written to the device divided by amount of bytes requested to be writ-

ten 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 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 submited 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. 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 Apendix 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 machine. In a single-device mode, storage generator simply creates new partition on a given device and creates and mounts file system. In a recipe mode, storage generator follows set of bash instruction to create more advanced configurations such as merging multiple devices using LVM, creating LVM cache or encrypted volume.

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 it's flexibility as well as large group of users and contributors. The flexibility is integral for conductuing less artifical 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 succeede. File system will eventually reach a state of equilibrium, when requests are equaly 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 beggining of testing, so other operations execute with higher likeliness.

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 controled by parameter `gaussian-stddev`. This is usefull 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 access 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 apparance of stop file.

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

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

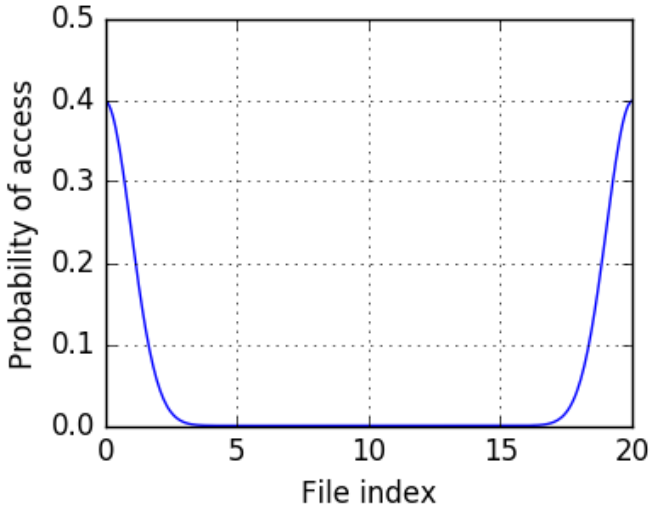

Figure 4.1: Normal distribution of file access

4.4.1 Changes to 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-

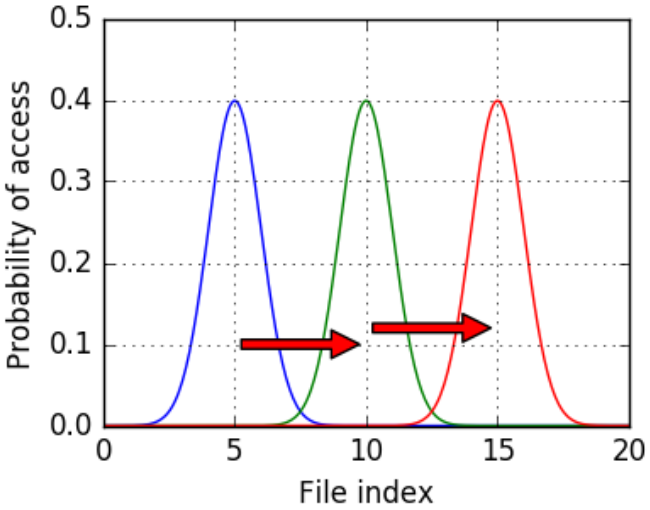

Figure 4.2: Moving random distribution

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 [12]. 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.

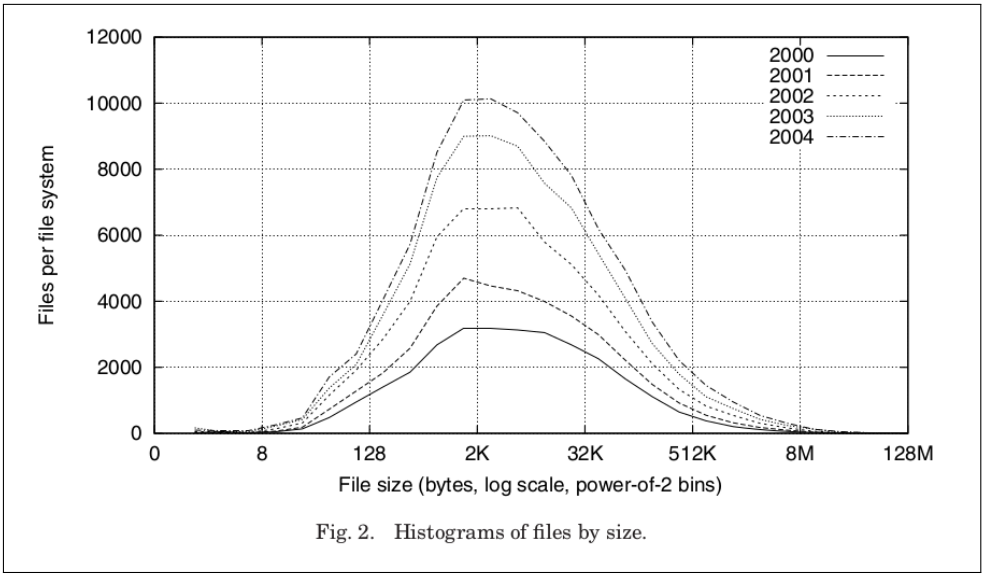

Fig. 2. Histograms of files by size.

Figure 4.3: File size distribution as measured by Agrawal [12]

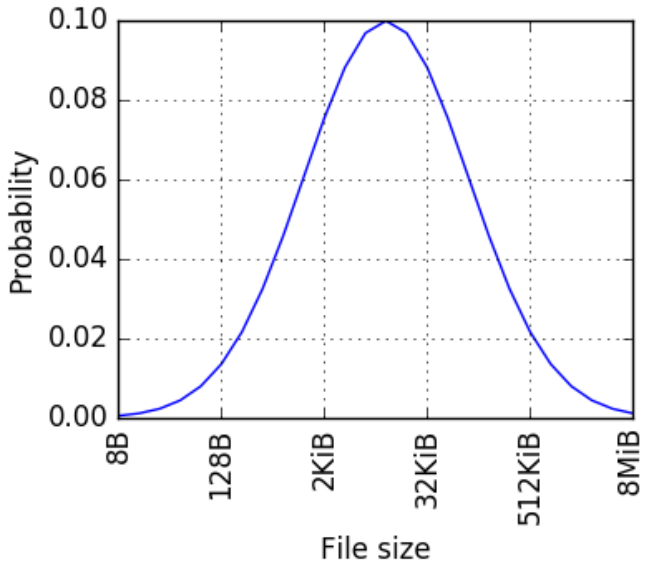

Figure 4.4: Logarhitmic normal distribution of file size

5 Aging the file system

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

5.1 Aging process

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 [12]. 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 meaningfull reporoducibility 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 code[1]. 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 substracted 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 Subsection 4.4.1

File size is another important factor of succesfull simulation. To consider a simulation succesfull, resulting layout should have similar file size distribution as real-life file systems. For that reasons, 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 alterning operations (create, delete, truncate, append, random write). The non-alterning operations are there to verify, if any performance drop occur 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 compresion of image as well. Created images can be further compressed by tools such `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 obsfucated (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 -|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 fragmententation 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. Furhtermore, when all mentioned operations are done, the thread calls `sync`, unpauses fs-drift and goes to sleep.

After fs-drift ends, fragmetation 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 weather or not to send data to server (usefull for developing purposes)
2. M, mountpoint
3. d, disk, device usded 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

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

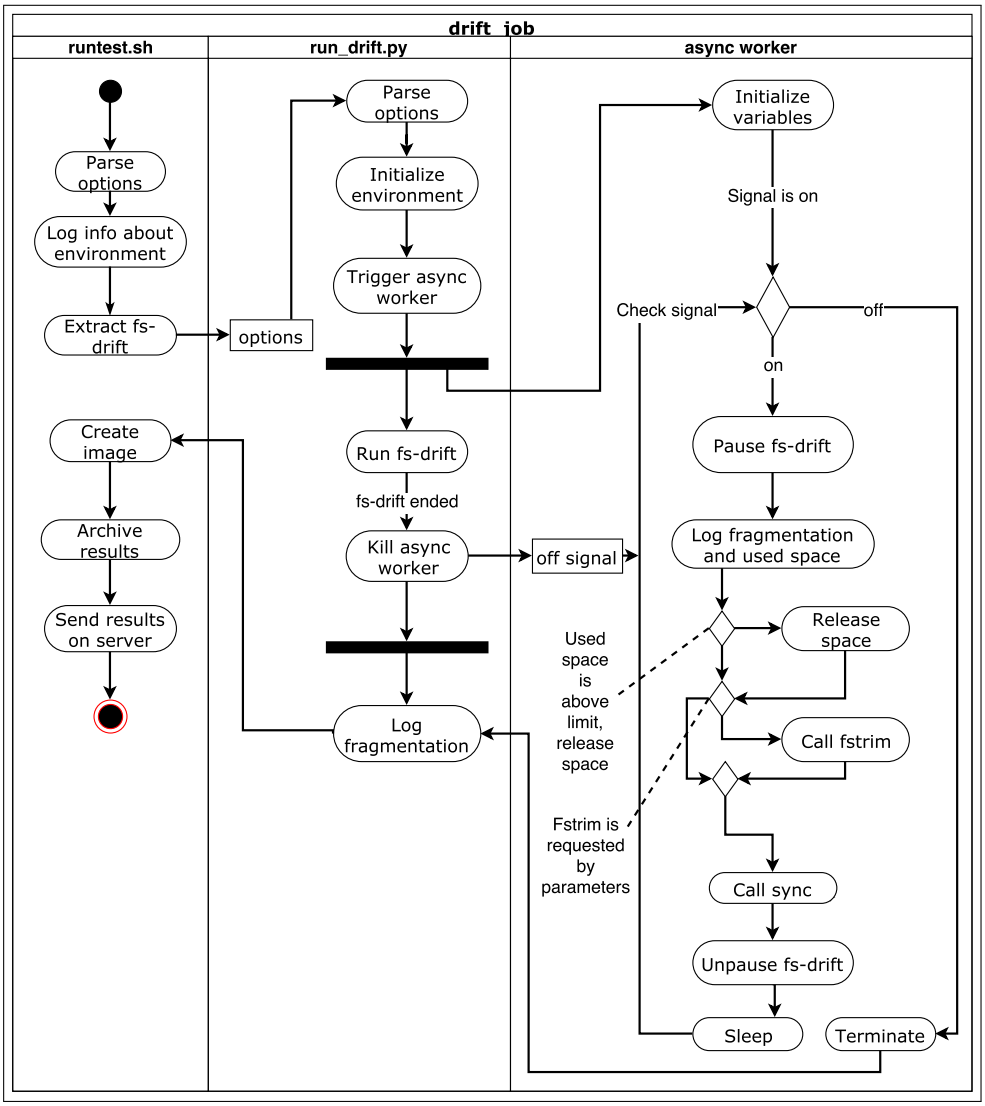

Figure 5.1: Activity diagram of aging test

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 settings

By default, FIO will create one file for every triggered thread. Furtermore, 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 iterest 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.

Similarily 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 instalation of necessary tools (libs, fio), the package finds and downloads coresponding file system image according to obtained parameters. As shown, images are stored compressed, therefore decompression is needed after download. Once these steps are succesfully 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 mouted. If the mounted file system utilize too much space, some files can be randomly deleted. Afterwards, pre-test fragmentation is logged and environment is initalized. When the test is over, post-test fragmentation is gathered.

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

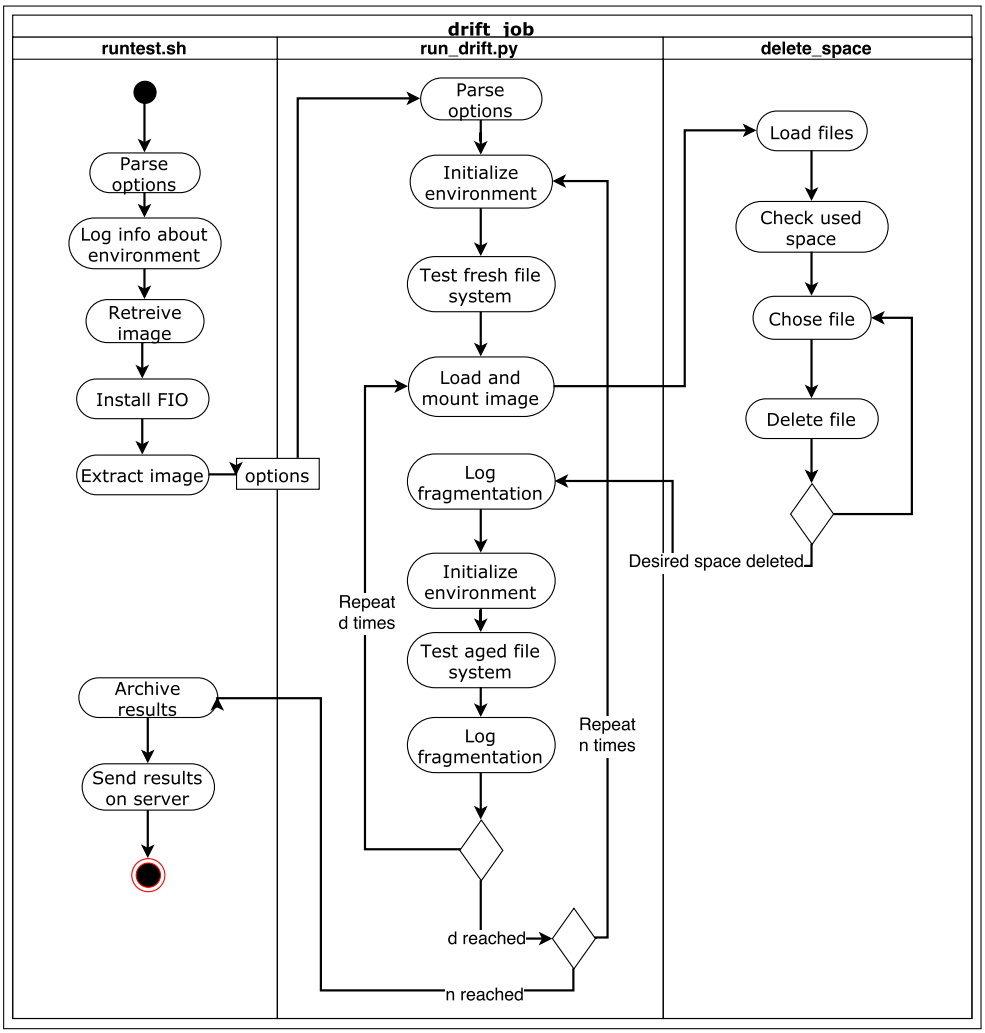

Figure 6.1: Activity diagram of testing of aged file system

7 Results

This chapter present 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

In this section, I describe testing environment and storage used for testing with created tests. The testing machines were available to me from a pool of systems of Red Hat Beaker environment.

Machine 1	
Model	Lenovo™System x3250 M6
Processor	Intel®Xeon®E3-1230 v5
Clock speed	3.40 GHz (4 cores)
Memory	1628 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	1628 MB
Storage	
Device 1	HP Proliant HardDrive
Interface	SAS
Capacity	600 GB
Device 2	SSD
Interface	SATA
Capacity	120 GB

Machine 3	
Model	IBM x3650 System M4
Processor	Intel®Xeon®E5-2620 v2
Clock speed	2.10 GHz (4 cores)
Memory	65 536 MB
Storage	
Device 1&2	IBM Solid State Drive
Interface	SATA
Capacity	400 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. This may skew the resulting histogram and it may be apparent, that fs-drift haven't created files smaller then default block size.

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 grups 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 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 filter had to be used for meaningfull display of latency change in time. Additionally linear regression is computed and displayed for every operation providing even more insight into the effects of aging through test.

7.2.3 FIO

7.3 Performance of aged file system

The file sysem aging tool succesfully induced fragmentation in all performed tests. From the data generated by aging tool, it is apparent, that the aging process may negatively affects performance of file systems. 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 (up to 80%). From the Figure A.2, we can see that higher file system utilisation induces tremendous fragmentation of free space.

Despite this occurence, it is remarkable, that fragmentation of used space is very similar accross these two test runs. On Figure X and Y, 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, seqential 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.

Table 7.1: Comparison of latencies of high and medium utilisation

	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
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

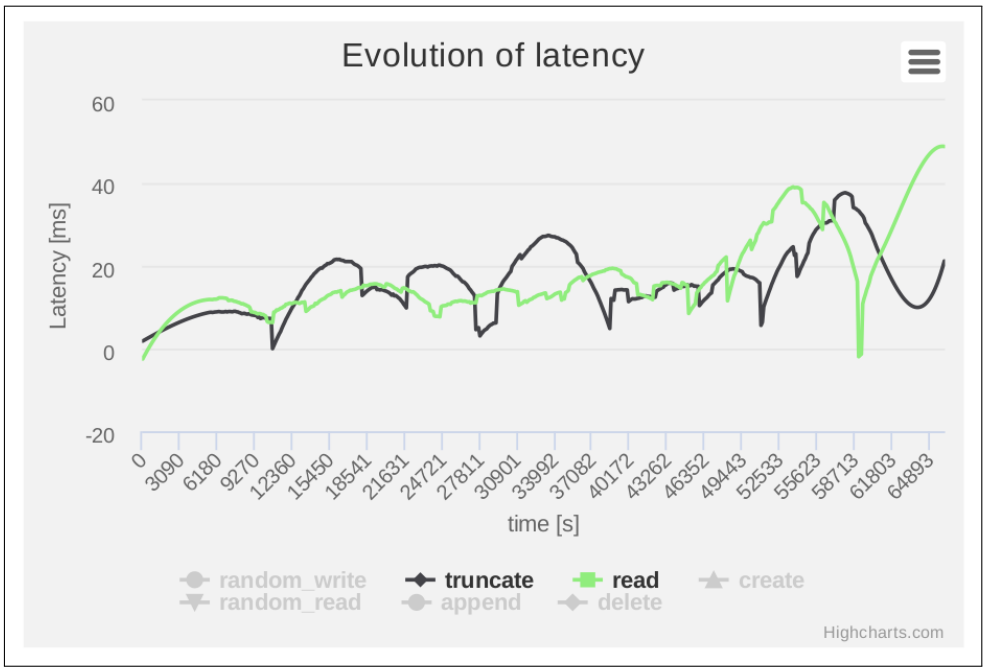

Figure 7.1: Latency growth induced by aging test

7.4 Differences betweem XFS and EXT4

When testing with SSD, XFS and EXT4 have very similar disk layouts in spite of free and used space fragmentation. After maximal disk utilisation both file systems have up to 50 000 fragments. At the end of the test, both file systems had 98% of files optimally allocated. The only difference when considering file block layout seems to be in size distribution of extents. As can be seen on Figure X, 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.

Table 7.2: Table of overall latencies

	XFS	EXT4	
random write	0.434 ms	0.594 ms	median
	1.40 ms	3.05 ms	mean
truncate	0.70ms	0.32 ms	median
	1.48 ms	1.35 ms	mean
read	1.37 ms	1.56 ms	median
	3.24 ms	3.47 ms	mean
create	0.38 ms	0.40 ms	median
	4.02 ms	3.84 ms	mean
random read	0.3 ms	0.46 ms	median
	1.32 ms	1.37 ms	mean
append	1.49 ms	1.76 ms	median
	6.46 ms	6.65 ms	mean
delete	0.097 ms	0.049 ms	median
	0.701 ms	0.346 ms	mean

7.5 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.

8 Conclusion

The aim of this thesis

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] 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.

[10] 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.

[11] Takashi Sato. "ext4 online defragmentation". In: *Proceedings of the Linux Symposium*. Vol. 2. Citeseer. 2007, pp. 179–86.

[12] Nitin Agrawal et al. "A five-year study of file-system metadata". In: *ACM Transactions on Storage (TOS)* 3.3 (2007), p. 9.

A Reports

A.1 XFS, HDD, medium utilisation

1. Testing environment: Machine 3
2. OS: RHEL-7.4

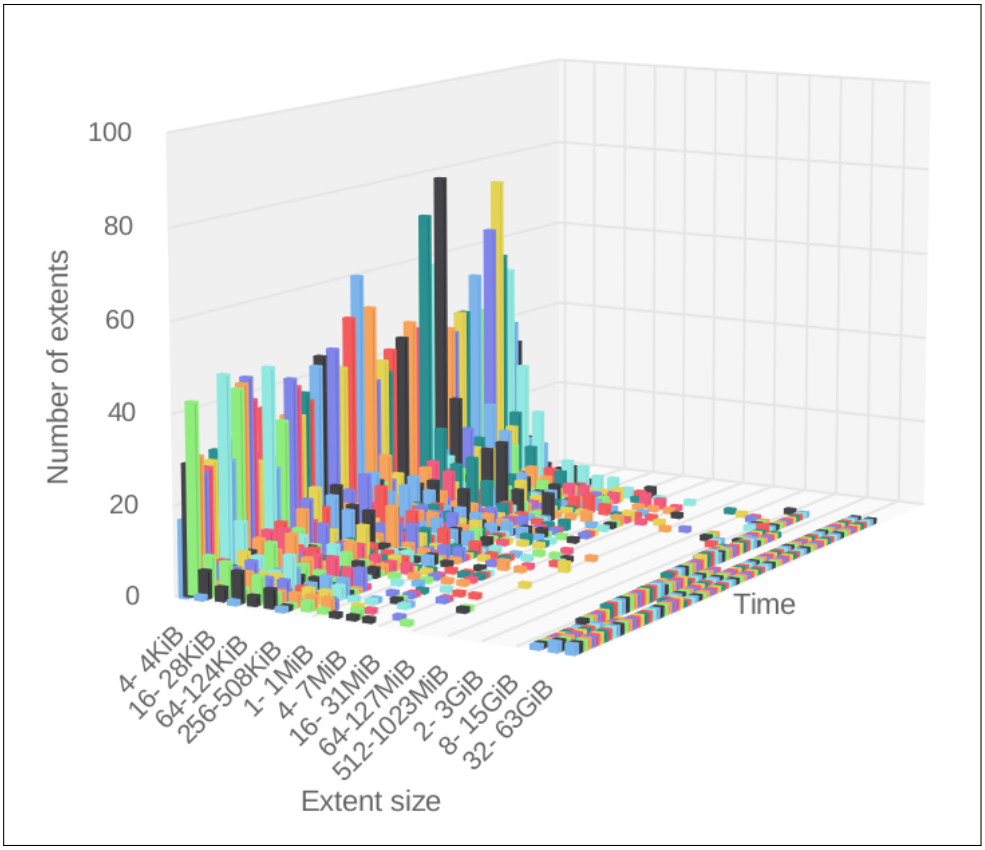

Figure A.1: Evolution of free space fragmentation with medium utilisation (XFS)

A.2 XFS, HDD, high utilisation

1. Testing environment: Machine 3
2. OS: RHEL-7.4

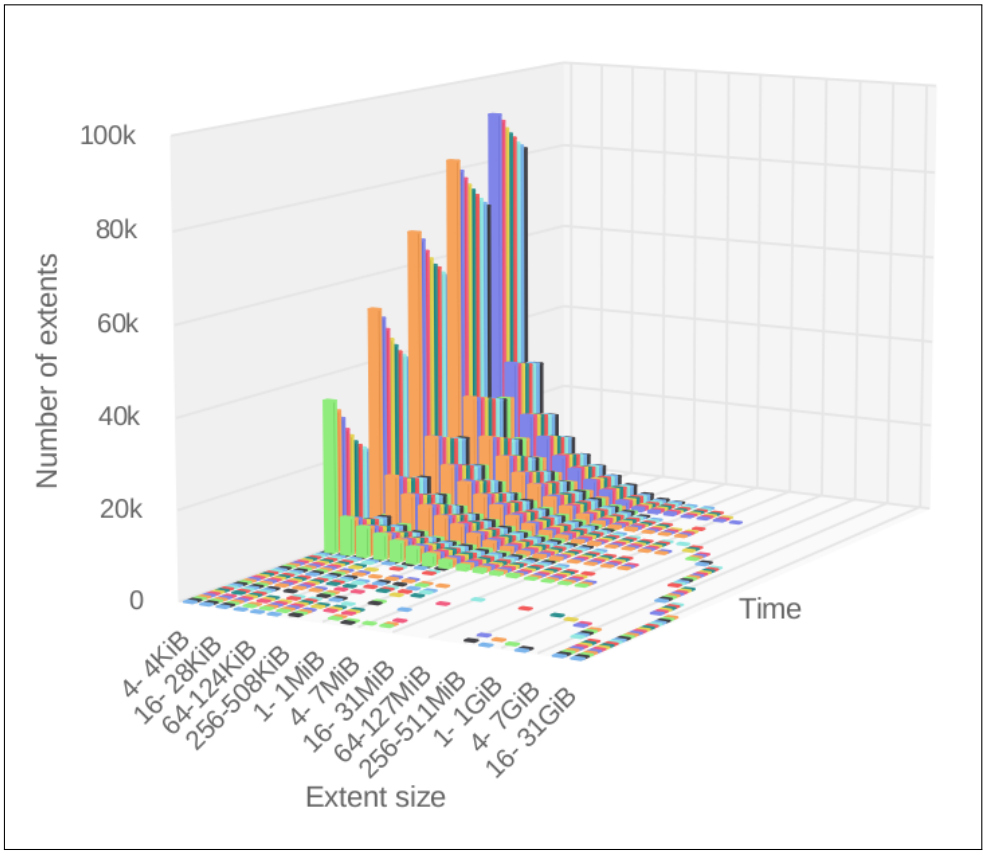

Figure A.2: Evolution of free space fragmentation with high utilisation (XFS)

B Examples

Example B.1: Specifying OS to be installed

```
<distroRequires>
  <and>
    <distro_family op="=" value="RedHatEnterpriseLinux7"/>
    <distro_variant op="=" value="Server"/>
    <distro_name op="=" value="RHEL-7.3"/>
    <distro_arch op="=" value="x86_64"/>
  </and>
</distroRequires>
```

Example B.2: Configuring environment using kickstart

```
<kickstart>
  <![CDATA[
    install
    lang en_US.UTF-8
    skipx
    keyboard us
    rootpw redhat
    firewall --disabled
    authconfig --enableshadow --enablemd5
    selinux --enforcing
    timezone --utc Europe/Prague

    bootloader --location=mbr --driveorder=sda
    zerombr
    clearpart --all --initlabel --drives=sda
    part /boot --fstype=ext2 --size=200 --asprimary --label=BOOT --
        ondisk=sda
    part /mnt/tests --fstype=ext4 --size=40960 --asprimary --label=MNT
         --ondisk=sda
    part / --fstype=ext4 --size=1 --grow --asprimary --label=ROOT  --
        ondisk=sda
    reboot
    %packages --excludedocs --ignoremissing --nobase
    @core
    wget
    python
    dhcpv6-client
    dhclient
    yum
]]>
</kickstart>
```

Example B.3: Executing task and passing arguments

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

```
44          </params>
45        </task>
```

Example B.4: Configuring storage using storage generator in beaker environment

```
46        <task name="/kernel_fsperf/storage_generator" role="STANDALONE">
47          <params>
48            <param name="TEST_PARAM_STORAGE_GENERATOR" value="-s create
                  -f xfs -t lvm -m /RHTSspareLUN1 -r jokerlvm -T 2
                  SATASSDLVM_xfs"/>
49          </params>
50        </task>
```

Table B.1: Table of overall latencies

	XFS	EXT4	
random write			median
		s	mean
truncate		0.32 ms	median
		1.35 ms	mean
read		1.56 ms	median
		3.47 ms	mean
create		0.40 ms	median
		3.84 ms	mean
random read		0.46 ms	median
		1.37 ms	mean
append		1.76 ms	median
	6.46 ms	6.65 ms	mean
delete	0.097 ms	0.049 ms	median
	0.701 ms	0.346 ms	mean