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

filesystem, xfs, IO operation, aging, fragmentation ...

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 (e.g. 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.

Testing of file system aging fundamentally consists of two steps. First is to bring fresh file system to an aged state and second is the actual performance test of the aged instance.

To achieve consistency of results and to shorten testing time, file system images are used in this thesis. Once the image of an aged file system is created, it can be stored for later use. By reloading the file system image on the device, it is possible to bring the file system to the original aged state, increasing stability of results. To save space, only metadata of created file system is used, since content of created files is random and therefore irrelevant. Replayed metadata point at various blocks on device, recreating fragmentation while seldom taking significantly less space.

Foremost, this thesis describe implementation of flexible tests which represent the two aforementioned steps. The first test is able to age fresh file system and store the result as an image for later use. The performance statistics collected in the process, as well as resulting layout can be used to evaluate ability of file system to respond to aging. Second test can evaluate resulting image even further by releasing some space and conducting performance test on resulting layout.

Furthermore, using developed tests to test different configurations of file systems and storage is demonstrated. The subject of research are differences between popular Linux file systems (XFS, Ext4) and storage technology (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. All the generated reports are part of this thesis in form of Appendix A.

The aging tests implemented in this thesis are established as part of testing cycle in Red Hat Kernel Performance team.

In the second chapter, the text present already conducted research of effects of age and fragmentation on file system performance. Third chapter describes used 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 and remaining chapters present the results obtained by using created tests. In the conclusion chapter, I discuss the effects of age on file system performance as well as further options for file system aging performance testing.

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 I/O 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 (metadata snapshot of 500 GB Ext4 file system can have up to 500 MB). 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 data is not always available, rendering this type of approach useful only in cases the researcher is already in possession of said data.

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].

The way to conduct synthetic aging, while keeping some amount of realism is to try to mimic the real-life usage by creating requests of random nature.

Aging workload generator such as fs-drift [6], used in this thesis, can be used (while carefully configured) to mimic long term real-life usage. Fs-drift simply creates a sequence of randomly chosen requests to be handled by the file system. The probability of the request type to be chosen is controlled by the workload table. In addition, whole process is highly configurable, making it possible to simulate various types of file system usage. Furthermore, a random distribution of file access can be controlled to mimic real-life user. All mentioned qualities, if used correctly, could result in fast, real-life mimicking file system aging.

3 File systems

In this chapter, I present basic information about file systems and describe main features of chosen file system in regard of fragmentation and scalability, which are important topics when discussing file system aging.

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 [7], XFS [8] and Ext4 [9], 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 XFS

XFS is a 64-bit journaling file system known for its high scalability (up to 9 exabytes) 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). One of the 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 I/O 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, that the file will be written in a contiguous group of blocks, avoiding fragmentation and reducing CPU usage as well.

3.2 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 exbibyte (approx. 1.15 exabyte). 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 support for online defragmentation [10] and e4defrag tool to defragment either single file, or whole file system.

4 Used tools

In this chapter, I presnet 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.

4.2 FIO

Flexible Input/Output tool is a IO 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 accept the workload specification as either a configuration file or a single line. Multiple different jobs can be specified as well as global options for every job.

The benchmark creates requests on system level, allowing for great power over the generated workflow.

There is a possibility to choose from 4 I/O 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 cachetiering workloads, different random distributions (f.e. Zipf) can be specified. FIO also supports process forking and threading.

After the test, FIO will compute 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.

4.3 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, see 4.1. 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 4.2

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 using *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 4.3, you can see bell curve moving by 5 units two times.

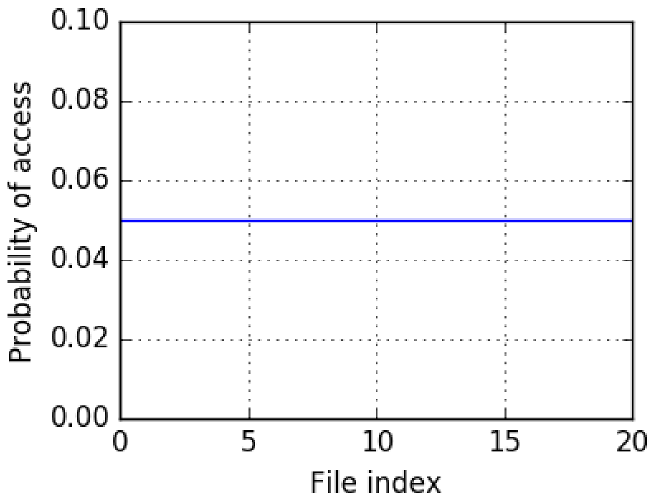

Figure 4.1: Uniform distribution of file access

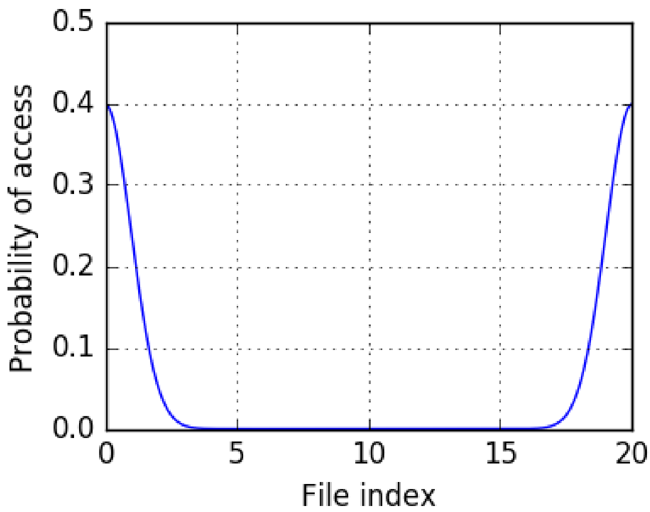

Figure 4.2: Normal distribution of file access

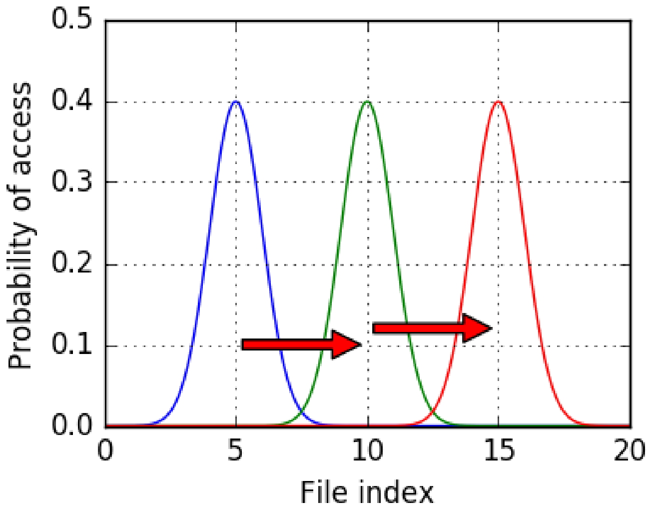

Figure 4.3: Moving random distribution

For purpose of this thesis, several changes had to be made to the original code. Besides small errors like invalid Readme information, a bug in code had to be repaired, otherwise the fs-drift would not delete files while issuing *delete* operation, marginally affecting aging process. Without this fix, the volume would just saturate and remain in more or less consistent state for the rest of the runtime.

Additionally, to be able to inspect and work with the file system while aging process is still running, I added an option to specify *pause file*. If the pause file is present, the fs-drift would not issue any requests and waits until the file is removed[1].

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.

1. For all the changes made to master branch of fs-drift, see ...

Used configuration of fs-drift for purposes of aging testing is further described in chapter 4.

4.3.1 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 I/O requests at random, sometimes, error occurs. Most operations (except *create*) 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 to 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 just used *randint* to uniformly choose file size at interval 1 to *maximum_file_size*. Such implementation offers very little control over file size distribution. Therefore, possibility to specify array of weights was added, so fs-drift can choose file size at different distribution.

The version used for testing in this thesis have all the mentioned problems repaired and features implemented.

4.4 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.

5 Creating aged 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 aging the fresh file system, bringing it to the aged, fragmented state. Fs-drift is quite flexible, therefore a lot of parameters and their impact on the final block 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 75 to 100 percent [11]. 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 used space remains at 100% through the rest of the testing time. This drawback was overridden by a small change in fs-drift code[1]. By adding the possibility to specify a *pause file*, other processes can stop fs-drift to generate IO requests for some time. This allows asynchronous thread triggered before fs-drift to pause it, when the file system usage reaches a specified amount and free some space.

In fs-drift, there is an option to define a workload table, describing probability of used operations. Since the goal of this workload is to create fragmented file system in a short time, operations which do not alter file system block layout are not included. Only create, delete, append, trucate and random write have representation in this workload.

File size is another important factor of succesfull simulation. To consider a simulation succesfull, resulting layout should have similar

1. For all the changes made to master branch of fs-drift, see Chapter 3

file size distribution as real-life file systems. In real-life file systems, file size is distributed non-uniformly, therefore to achieve realistic file system layout, file size distribution has to be non-uniform during the test.

As a reference for file size distribution, five year old study of file system metadata was used [11]. Figure 3 shows measured file size distribution of used file systems. To mimic this layout, I modeled logarhitmic normal random distribution. The shape of resulting distribution is shown in figure 5.1.

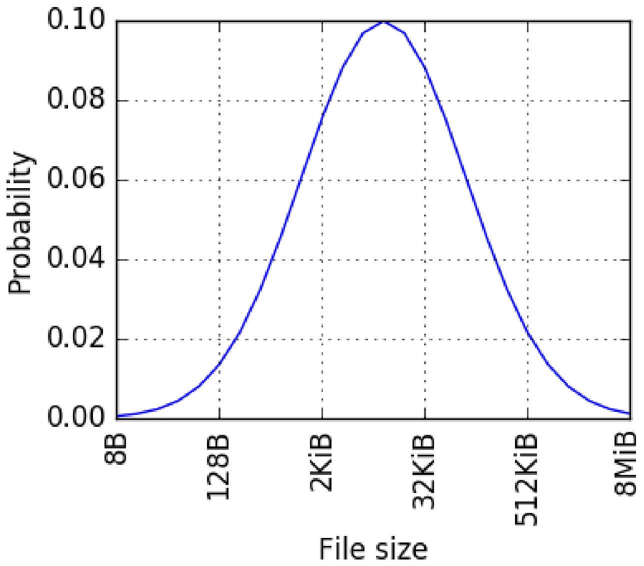

Figure 5.1: Logarhitmic normal distribution of file size

Duration of the test is an important factor as well. In general, the testing time should be long enough for the environment to stabilise. When considering file system aging, the test should spend most of the run time after the saturation phase (e.g. the phase of filling the file system to a desired fullness). After developing a test, it is necessary to conduct few pilot runs to confirm if any stabilisation occur and set the duration accordingly. On the other hand, for the test to be usefull in professional environment, testing time should not be too long. The goal is to maximize testing correctness while minimising the time for the test to complete.

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* and *xfs_mdrestore*. 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 replayed back on a device. From that point, file system can be mounted and revised.

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 replayed back on device by tool *xfs_mdrestore*. File system can be then mouned and inspected as needed.

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 is triggered to periodically log free space fragmentation and file system usage while fs-drift is running. If the usage reaches –hranica– specified amount of space si freed from file system[2].

After the fs-drift ends, 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.

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

2. The mechanism of space releasing from file sytem is described in next chapter

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 Benchmark settings

To ensure stability of test results, simple form of standard performance test was used.

6.2 Test structure

Performance testing of created images is done by a package 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, the image is restored on the device by using presented tools (e2image, xfs_mdrestore). If the image restoring completes succesfully, file system can be mounted and worked with exactly like it would be just after the aging process.

After image restoration, some amount of the files needs to be deleted to create space for the performance test to take place. The files to be removed are chosen randomly until desired amount of volume has been freed. By using this workflow, e.g. freeing some amount of space, we can simulate aged file system in various phases of aging by using just one image of a very fragmented file system.

When free space is reclaimed, FIO test will take place using parameters given to recipe_fio_aging. The overall space occupied by the test should not be larger than available space on the file system, otherwise the test will either fail completely or report incorrect results.

For statistical correctness, the FIO test can run several times in a row. After last iteration, the results are archived and sent to data-collecting server.

Parameters available for recipe_fio_aging:

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

7 Testing environment

In this chapter, I describe testing environment and storage used for testing with created tests.

Machine1	
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 EG0600FBVFP
Interface	SAS
Capacity	600 GB

Machine2	
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 EG0600FBVFP
Interface	SAS
Capacity	600 GB
Device 2	SSD
Interface	SATA
Capacity	120 GB

Machine3	
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 SSDSC2BB480G4i
Interface	SATA
Capacity	400 GB

Installation of system and configuring is done using tools presented in Chapter 3.

The system installed on machines is RHEL-7.3 with kernel 3.10.0-514.el7.x86_64

7.1 Storage

7.1.1 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 I/O 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.

7.1.2 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 solid state drives, 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 blocks (e.g. update metadata), the data have to be read, erased from the drive and written back modified. This effect, known as write amplification, can have significant impact on performance of file systems.

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 cells 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. However, 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.

8 Results

8.1 Performance of aged file system

8.2 Differences betweem XFS and EXT4

8.3 Differences accross different storage

9 Conclusion

Here I will admit, that these results were not really surprising and ABSOLUTELY no breakthrough, however, as noone really research this branch of QE, the results are definitely a step further in this field.

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. URL: http://doi.acm.org/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. URL: http://doi.acm.org/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. URL: http://doi.acm.org/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. URL: http://doi.acm.org/10.1145/323647.323631.

[5] Ningning Zhu, Jiawu Chen, and Tzi-Cker Chiueh. "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] Ben England. *fs-drift*. https://github.com/parallel-fs-utils/fs-drift. 2017.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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

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"/>
```

```
      </params>
    </task>
```

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

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