Masaryk University
Faculty of Informatics

The effects of age on file system performance

Bachelor's Thesis

Samuel Petrovic

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 Petrovic

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

We live in an age of information. Hardware and software technology improves every day. There is an increasing need to wrok with larger and larger databases, multimedia applications and to store great amount of data. This cause great pressure on performance of storing and retrieving information, e.g. input and output (I/O) operations.

The part of an operation system that handles communication with a physical device is called a file system. Because of increasing demands on performance of I/O operations, there has been quite wide tehcnological progress in approach to performance issues.

In past years, great many tools and tests (e.g. benchmarks) were developed as means for users and researchers to explore behavior and to measure performance of file systems. New versions of file systems are released very often (almost weekly) and therefore there has been an pressure on quality engineers to evaluate new versions fast enough.

The standard work flow consist of running test on freshly installed operation system and newly created file system without running other applications, which might cause noise in measurements and eventually invalid results.

Although this work flow shows great results, it only gives us a general idea about how does file system perform at early stages of its life cycle. Results of these tests are therefore quite unrealistic and won't address the behavior of file system after long-term usage.

Prolonged usage of file system impel it to do more and more optimisations with impact on its perfmormance. The more files has been created, deleted or expanded, the free space of file system is progressingly more fragmented. Such fragmentation means, that blocks of data are scattered through physical device which results in degradation of performance.

The ideal approach for research pointed on effect on age of file system would be to put file system under stress conditions for a few years or months, gathering information in the process, however, such approach is apparently impractical or even impossible, because the demand has to be satisfied in matter of weeks to respond to new versions of file systems.

2 State of art

One of the earliest ideas about how to deal with this problem would be to capture a snapshot of a file system that had already been used for a prolonged time.

This approach of testing on such a snapshot, however, would probably lead to optimising for a very specific instance of old version of given file system.

If researchers want to predict behavior of a new version of file system after prolonged time of usage, they should be able to create a simulation, which would mimic the aging process, but in short period of time.

This can be technically implemented, because the idea is to run the simulation continuously under very heavy workload, which would not have any delays. We believe, that file systems used in real life are not really used continuosly, therefore researchers should be able to compress the time of creating such an aged file system in matter of days.

Unfortunately the demand from users and developers has yet not been met by standard studies. There has been few studies executed in last millenium[3], that shed some light on this topic, but overall, modern research is in hands of regular users, which remain in amateur sphere, lacking professional equipment to execute such research properly.

My general idea is to design a set of heavy workloads, simulating aging process, to run on different types of devices and at least two types of file systems and then to execute a series of performance tests. Results of these tests could be then compared to fresh file systems, showing performance differences between them. Comparing different aged file systems between each other would clearly be an interesting topic of research as well.

3 File systems and used tools

Possible other sections to explain terms: journaling filesystem, allocation groups, B+ trees

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 a storage, e.g. device. It is sometimes called a 'bookkeeper' of operational system. As an analogy to paper-based systems. Basic user-accesed units are called files, which could be clustered into directories.

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 either as free or used.

Files which are non-continous 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 inodes (index nodes). 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 unrelated to any specific file, such as information about bad sectors, free space or block availability.

*este nieco o bitovych mapach volneho miesta.

In this thesis, targeted file systems will be UNIX XFS and EXT4, which are main Red Hat supported file systems. These file systems belong to the group of 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.

*IO scheduler

3.2 XFS

XFS is a 64-bit journaling file system created by Silicon Graphics, Inc(SGI) in 1993. It is known for great performance in execution of paralel I/O operations, because of its 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, threads, 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 for the highly efficient location of free extents for file system operations.

Prevention of file system fragmentation consist mainly of a feature called *delayed allocation* as well as online defragmentation(*xfs_fsr*), that can turururu

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 (such as *sync*). 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.

3.3 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.

Traditionally, ext* systems use an indirect block mapping sheme. Such approach is generaly inefficient for large files when using operations like deleting or truncating. Ext4, as well as XFS use approach of

extents, which positively affect performance and encourage ?continuous layouts?.

When allocating, ext4 use multiblock allocation. **?what is multiblock allocation?**

Similary as xfs, ext4 use delayed allocation to increase performance, especially when in use with multiblock allocation and extent-based approach, also reducing fragmentation on the device. For cases of fragmentation that still occur, ext4 provide support for online defragmentation and *e4defrag* tool to defragment either single file, or whole file system.

3.4 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 contributors and users.

3.5 Fs-drift

fs-drift is a workload aging test written by Ben England. It relies on randomly mixed requests generated by inner heuristic (according to parameters). These requests can be writes, reads, creates, appends or deletes.

At the beginning of run time, the top directory is empty, and therefore *create* requests success the most, other requests, such as *read* or *delete*, will fail because of lack of files and small probability of randomly choosing existing one.

Over time, as the file system grows, *create* requests began to fail and other requests succede more. Finally, file system will 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 until one of the *STOP* conditions are met (specified with parameters).

Fs-drift is very flexible and can be used to simulate lots of different workloads by operating with various file sizes, request types and different kinds of random distribution.

3.6 File system images

To achieve consistency of results and to shorten testing time, file system images are used. Once the image is created, it can be stored for later use and replayed back on device. To save space, only metadata of created file system are 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. These images can be created by using tools developed to inspect file systems in case of emergency. For ext based file systems, there is e2image tool and for xfs, there is xfs_metadump. Both tools create images as sparse files, so compression is needed.

E2image tool can save whole ext based 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.
Creating compressed image using e2image:

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

Such images can be later replayed back on a device. From that point, file system can be mounted and revised.
Replaying 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 (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 from where it can be passed to a compression tool. Creating compressed image using xfs_metadump:

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

Such images, when uncompressed can be replayed back on device by tool xfs_mdrestore. File system can be then mouned and inspected as needed:

```
xfs_mdrestore $NAME $DEVICE
```

4 Storage media

4.1 SATA

4.2 SAS

4.3 HDD

4.4 SDD

5 Implementation

5.1 filesystem_ager

5.2 Performance testing of images

Performance testing of created images is done by a package recipe_fio_aging. Upon instalation, 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 can be replayed on the device by using presented tools (e2image, xfs_mdrestore).

Measuring a performance is done by a tool I developed, recipe_fio. Similar to filesytem_ager, recipe_fio use fio tool to handle desired IO operations, but instead of focusing on filling the filesystem, the script use measurement features of fio, which consist of performing IO operations and reporting results.

*talk about used recipe

For purposes of this thesis, I let recipe_fio to report bandwidth and operations per second(IOPS).

The main script receives slightly enhanced fio configuration file, enriched of some non-fio parameters, which are used by test only. These parameters are:

1. used filesystem
2. number of test repetitions. For statistical stability, I decided to run the test several times under same conditions.
3. specifying a snapshot to be tested
4. flag which represents whether or not to rsync data on a result-storing server

After compiling, tool parses the parameters and gathers information about system, which consist of: version of kernel, time and date, hostname, RHEL compose, memory, kernel, mount, system info, system variables and fio version.

Then it proceeds to set environment for testing by:

1. Installing fio tool
2. Creating directory for results

3. Loading given snapshot on a device
4. Randomly removing volume from the device to make room for test

Volume is randomly removed using random_delete_volume.py script. This scriptt globs all files in the filesystem, retrieves information about used volume as well as it's overall size. Then proceeds to randomly choosing files to delete and stops when desired volume is freed. The approach of recursively globbing all files may be inefficient, but this way, we can be sure, that volume is deleted from whole device evenly.

Python script run_tests.py, manages to parse recipe parameter, resulting in creating one or several fio configuration files in the directory, further adding logging parameters to them. Then for every created file, directory on the desired medium is created and fio tool is triggered. If the script succesfully ends, file OK is created in the results directory.

When the measurement is over, bash script generate the name of results, which consist of:

1. time
2. date
3. used snapshot
4. version of kernel
5. version of RHEL compose

Then proceeds to compress results into tar archive with generated name, and according to -g flag will, or will not, rsync the result onto the data server.

5.3 Inspecting filesystem

For determining some overall idea about an extent to which is the filesystem aged or dirty, I wrote scripts that generate histograms representing fragmentation of used space as well as fragmentation of free space. Both scripts use common linux tools and pyplot to generate the graphics. Both scripts can display linear or logarhytmic Y scale.

Script extent_distribution.py makes use of xfs_io fiemap tool, which is a tool to display extent distribution of a given file and works correctly even for ext* filesystems.

The script will first recursively crawls the whole filesystem from given top folder and makes a list of all files. Fiemap is then run over every file separately.

The only data, that are then parsed from the output, is how many non-contigous extents does the file have. These integers are aggregated to a single list, from which are then counted, and final histogram is made.

Script free_space_fragmentation.py use the tool e2freefrag, which runs over a device, and outputs the histogram of free space fragmentation in texutal form. Script will store this output and then easily parse the histogram and aggregate the data into a graphic form.

5.4 Aging recipes

To determine which fs-drift settings will be the most fitting for purposes of this thesis, I wrote a small python script fs-drift_matrix.

It is capable of taking matrix of possible fs-drift paramteres from *json* file and then run it on a device.

After every run, histograms-generating scripts are triggered to store histograms of *free space* and *used space* fragmentation. Also outputs of the *fs-drift* script and *df* command are logged.

As the creator states in README, to fill up a filesystem, maximum number of files and mean size of file should be defined such that the product is greater than the available space.

For the purpose of this thesis, desired usage is 60%-100% with enough fragmentation to consider the device aged.

5.5 Data processing

6 Testing environment

The aging process took place on an ibm3250 machine with following parameters:

1. Intel(R) Xeon(R) CPU, X2460 (2.80Ghz, ??? Cache,8 cores)
2. RAM 10GB ???Mhz DDR? ???
3. 4x300GB SAS disks + 1x50GB Sas disk ???

THIS IS AN EXAMPLE, HOW SHOULD SCECIFICS OF A MACHINE LOOK

2 x Intel Xeon E5-2620 (2.0GHz, 15MB Cache, 6-cores) Intel C602 Chipset Memory - 16GB (2 x 8GB) 1333Mhz DDR3 Registered RDIMMs CentOS 6.3 64-Bit 100GB Micron RealSSD P400e Boot SSD LSI 9207-8i SAS/SATA 6.0Gb/s HBA (For benchmarking SSDs or HDDs)

The system installed on machine was —— with kernel 3.10.0-229.el7.x86_64

I created a two volume groups, G1 and G2. Each group have a pair of 300GB disks with striping of 2. This setting allows to double the speed of IO operations.

On each volume group i created logical volume of 100GB.

6.1 HDD and SSD

HDD is a rotational disk, which requires specific approach from kernel, to ensure the lowest possible seek time. Seek time is a time for moving parts of the device to find next relevant block of data. This affect overall performance greatly, because with large fragmentation, seek time becomes quite high.

As for SSD, this type of device does not have any moving parts, which make perform really well. One of the problems, however, is limited lifecycle of memory cells. SSD manufacturers deal with this problem by adding controler with its own scheduler, which make sure, no parts of the device are used significantly more than other parts.

When aging the filesystems, I expect for those grown on HDD to perform significantly slower after aging process, and I expect SSD filesystems not to be affected at all, or maybe significantly less.

7 Results

The output of result generator is a htlm report summarising all information about system, links to raw data and charts of measured values. *talk about highcharts and how do you represent data

7.1 Comparing against fresh filesystems

7.2 Comparing performance of Ext4 and XFS

7.3 Comparing overall state of aged EXT4 and XFS

7.4 Comparing rotational and solid state disks

8 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.