

MASARYK UNIVERSITY
FACULTY OF INFORMATICS



The effect of aging on filesystems performance

BACHELOR'S THESIS

Samuel Petrovic

Brno, Spring 2016

Replace this page with a copy of the official signed thesis assignment and the 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

1	Introduction	1
2	State of art	3
3	Filesystems and used tools	5
3.1	<i>Filesystems</i>	5
3.2	<i>XFS</i>	6
3.3	<i>EXT4</i>	7
3.4	<i>FIO</i>	7
3.5	<i>Fs-drift</i>	7
3.6	<i>Snapshots</i>	8
4	Implementation	9
4.1	<i>filesystem_ager</i>	9
4.2	<i>recipe_fio</i>	10
4.3	<i>Inspecting filesystem</i>	12
4.4	<i>Aging recipes</i>	12
4.5	<i>Data processing</i>	13
5	Testing environment	15
5.1	<i>HDD and SSD</i>	15
6	Results	17
6.1	<i>Comparing against fresh filesystems</i>	17
6.2	<i>Comparing performance of Ext4 and XFS</i>	17
6.3	<i>Comparing overall state of aged EXT4 and XFS</i>	17
6.4	<i>Comparing rotational and solid state disks</i>	17
7	Conclusion	19

List of Tables

List of Figures

1 Introduction

This thesis will debate the effect of aging and fragmentation on filesystem performance, specifically ext4 and xfs. Explain further the official topic. Performance testing has become stable part of most software developing companies. Nowadays, performance testing is included in most Performance of Red Hat supported filesystems is closely watched by our team, but the effect of aging is not included in the testing matrix and is quite lightly understood in general. Therefore, this thesis will try to shed some light as what the effect could possibly be as well as to design somewhat conceptual way of exploring this field further.

2 State of art

We live in an age of information. Hardware and software technology improves every day. There is a need to work with large databases, multi-media applications and to store great amount of data on a memory device. This place great pressure on performance of storing and retrieving information, e.g. input and output(I/O) operations.

Part of an operation system that handles communication with a physical device is called a file system. Because of such increasing requirement on performance of I/O operations, there been great technological progress in approach to performance issues.

There has been great many tools and tests (e.g. benchmarks) developed to explore behavior and measure performance of file systems.
*talk more about benchmarks

Good file system have to be able to work with different types of devices and should be easy to use and customise. Also, it should show good performance when executing all kinds of different I/O requests quickly.

There are few types of

However actual filesystem performance research and testing is strongly peripetual and unexplored area.

3 Filesystems and used tools

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

3.1 Filesystems

File system is a set of tools, methods, logic and structure to control how to store and retrieve data on a storage, e.g. a device. It is sometimes called 'bookkeeper' of operational system. As analogy to paper-based systems, basic user-accessed units are called files, which could be clustered into directories.

The system stores files either continuously or scattered across filesystem in form of extents. The basic accessed data unit is called a block, which size can be set to various sizes. Block can be either free or used, meaning there is some data stored or not. 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 called metadata, e.g. data about stored data.

Depending on filesystem, these metadata are stored in various ways (different tables or arrays), but on modern file systems, there are objects called inodes (i stands for index). Each file a filesystem manages is associated with an inode, every inode has its number in an inode table.

*este nieco o bitovych mapach volneho miesta.

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

Journaling file system keeps a structure called journal, which is a buffer of changes not yet committed to the file system. After system failure, these planned changes can be easily read from the journal, thus making the file system easily fully operational, and in correct(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 parallel 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 filesystem, as well as its 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 (imrpoving 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.

Extent describes one or more contiguous blocks, which can considerably shorten list of blocks.

Metadata journaling ensures consistency of data in case of emergency situations as f.e. system crash.

Prevention of file system fragmentation consist mainly of *delayed allocation* feature 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, subtracts space from the free-space counter, but won't allocate the free-space bitmap. The data is held in memory, instead, until it must be flushed (to storage) because of memory pressure when calling Unix *sync*, or when flushing dirty buffers. 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 scheme. Such an approach is generally inefficient for large files, on 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, which is more efficient way than one block allocation at time, which is present in earlier ext* file systems. Multiblock allocation has far better performance, particularly when in use with delayed allocation and extents.

Similarity as xfs, ext4 use delayed allocation design, 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 filesystem.

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 according to options. 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 filesystem grows, *create* requests began to fail and other requests succede more. Finally, filesystem will reach a state of equilibrium, when requests are equally likely to execute. From this point, the filesystem will not grow anymore, and the test will run until one of the *STOP* conditions are met (specified with parameters).

3.6 Snapshots

From time reasons, but to ensure stability of results as well, I needed a solution, how to test an aged filesystem with possibility to recover to the pre-test state. For this matter, I made use of kernel native tools for inspecting filesystem in case of corruption or other forensic reasons.

Unfortunately, I had to use different tools for either filesystem. For EXT4, the tool is `e2image[]` and for XFS `xfs-metadump[]`.

Both of these tools have possibility to store only metadata of files, not the actual data, which is a great option for this thesis, as it does not need the randomly generated data inside of files. Moreso it reduces the size of a snapshot to a mere fraction of the acutal volume, and the process of storing the snapshot si very quick.

4 Implementation

4.1 filesystem_ager

This aging tool is a simple approach to write and remove many files of random size.

The tool consist of three scripts and one common library called *functions*. The scripts are named *filesystem_ager.py*, *fio_config_generator.py* and *random_deletor.py*.

The workflow consist of calling *filesystem_ager*, with desired parameters. Script manages triggering *fio_config_generator*, calling *fio* tool on generated config and triggering random deletor. These three actions are repeated given number of times. Parameters of *filesystem_ager* are:

1. Total desired size do be written in one cycle
2. Denominator of total desired size (Total desired size will be divided by this number)
3. Range of size of written files
4. Number of cycles

Although FIO tool has some parameters to randomize the size of files which are written, the management of file sizes and randomisation, as well as naming of files is handled by *fio_config_generator* instead, to provide more control over those qualities. Parameters of this script are:

1. Total deisred size to be written
2. Range of size of written files

The script will generate global settings of a workload, then proceeds to generate jobs for every file that will be written. File size is always the name of that file, and these are gathered to a list, then list of generated files is returned and script ends. Including file size in its name, as well as indexation of files will help effectively search and delete files in the random deletion process, without need to search for files on the disk and examine them for size. Simplistic approach in *fio* config will hopefully result in compatibility and reliability in use with any *fio* version.

After config file is generated, *filesystem_ager* will run *fio* tool on generated config and therefore, files are written on the device.

4. IMPLEMENTATION

The removing of files is handled by `random_deletor` script. Its parameters are:

1. Total written size
2. Denominator of total size
3. Range of size of written files
4. Number of existent files

If denominator equals zero, `random_deletor` won't remove any files and will return empty list. Otherwise, desired range of deletion is estimated. `Random_deletor` then proceeds to remove files while desired volume is not deleted. Files are randomly selected through choosing random integer from zero to number of existent files. This step may seem inefficient, but with large amounts of generated files, the time to perform successful selection will not change dramatically. Selected file name is then parsed for size information, and if it fits into desired volume to remove, it is deleted, through `subprocess` command. Names of removed files are gathered in a list and returned.

Number of deleted files is subtracted from number of existent files. `filesystem_ager` then sums up deleted volume, log it as well as other information and triggers the cycle again.

However, after few runs, I decided not to use this approach for actual aging, because the time needed to fill and appropriately age the filesystem simply took very long. Instead, I was looking for other, already created tools I could use.

4.2 `recipe_fio`

Measuring a performance is done by a tool I developed, `recipe_fio`. Similar to `filesystem_ager`, `recipe_fio` uses `fio` tool to handle desired IO operations, but instead of focusing on filling the filesystem, the script uses 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.

4.3 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 logarithmic 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-contiguous 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 textual form. Script will store this output and then easily parse the histogram and aggregate the data into a graphic form.

4.4 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` parameters 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.

4.5 Data processing

5 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 SCECIFS 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.

5.1 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.

6 Results

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

6.1 Comparing against fresh filesystems

6.2 Comparing performance of Ext4 and XFS

6.3 Comparing overall state of aged EXT4 and XFS

6.4 Comparing rotational and solid state disks

7 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.