Lund University

Bachelor Thesis

Presented to LTH School of Engineering

Scalability study of database-backed file systems for High Throughput Computing

Author:
Andy Trinh

Supervisors:
Flavius Gruian
Florido Paganelli

August 24, 2017

Abstract

The purpose of this project is to study the read performance of transparent database-backed file systems, a meld between two technologies with seemingly similar purposes, in relation to conventional file systems. Systems such as the ARC middleware relies on reading several millions of files every day, and as the number of files increases, the performance suffers. To study the capabilities of a database-backed file system, a candidate is chosen and put into test. The candidate, ultimately being Database File System (DBFS), is Oracle Database using FUSE to create a transparent file system interface. DBFS is put into test by storing millions of small files in its datafile and executing a scanning process of the ARC software. With the performance data gathered from these tests, it was concluded that DBFS, while performing well on an HDD when compared to ext4 in terms of scalability and read performance, is simply outperformed by XFS with small (from 50 000 files) and large (up to 1 600 000 files) directories.

Key words: *database-backed file system, dbfs, scalability, xfs, ext4, database, file system, fuse, arc, read performance, alternative storage, rdbms, file system interface*

Acknowledgement

I would like to thank all the people that helped to enable and support this study. Some truly stand out and have offered support and advice in good as well as stressful times, and they have my sincerest and most heartfelt gratitude.

I am truly grateful to **Flavius Gruian** for supervising this project. His guidance, support and critical thinking has been invaluable and his constructive comments have greatly improved the report and the project.

My deepest and warmest thanks to **Florido Paganelli**, for being a great supervisor, inspiring mentor and a supportive friend. Working with him has been an educational and joyful experience with many laughs shared. I could not possibly have wished for more.

Contents

Chapter 1

Introduction

File systems and databases are closely related to each other, but differ in some properties which ultimately dictates the area of usage. In very broad terms, the properties they share in common are the fact that they both store data and have a conventional methods of managing this data. The subject of this study lies in a grey zone between these two technologies: the read performance of database-backed file systems. The following sections will discuss the background of file systems and databases with some traditional definitions and how these technologies might benefit the middleware ARC.

1.1 Data storage technologies

Before lunging into historical context and properties of databases, one might wonder what a database is to begin with. A database is simply an organized collection of data, usually stored for a long period of time. Using this definition, the history of databases stretches all the way to ancient civilizations, where people would store literature and other important texts in libraries. It was not until the 1960s where the modern computerized databases began its development. The databases back then derived from file systems, which suffered from numerous problems in terms of accessibility and persistence. To access data in a file system, the user has to know the specific location of the data. Moreover, a file system generally cannot warrant data persistence. These disadvantages led to some criteria a modern database is expected to fulfill. To meet these criteria, the concept of a Database Management System (DBMS) was introduced, a software application which as the name suggests manages the stored data. A DBMS is therefore essential for a functioning database and are often times taken for granted when speaking of just databases (Garcia-Molina et al., 2003, pp. 1-4).

The idea of data location was abstracted with the introduction of relational databases in the 1960s. With relational databases, users no longer needed to know where specific data is located, nor did they need to spend time studying how each and every data storage is structured. Instead, they can rely on a programming language (i e SQL) to retrieve the stored data (IBM, 2003). Although many technological advances and alternatives, such as NoSQL, has been made since then, relational databases are still a popular alternative for data storage.

When using file systems, arbitrary user data is stored in files and the location of these files are an important property of file systems. This contrasts a relational database, which stores structured collections of data and abstracts the notion of data location. Also, when speaking of file systems, details such as blocks, block size and partitions are of importance (Giampaolo, 1998, pp. 7). Some software solutions, such as ARC, are heavily reliant on file systems and its properties.

In terms of the file system development today, a common trend is that they today tend to focus on largely sized data and distributed systems. This can be realized by just observing the amount of distributed file systems released in the past decade (as of 2017). Many popular file systems, ext4 (Mathur et al., 2007, pp. 21-22) and XFS (Sweeney et al., 1996, no pagination) to name a few, was designed to handle very large files. ARC, however, is reading a large amount of small files with sizes ranging a few bytes to a few kilobytes. The file system alternatives for ARC and similar software are remarkably smaller.

1.2 Thesis specifications

1.2.1 Background and problem statement

At CERN, scientists and engineers are conducting an array of different experiments which require heavy computing power. The computing process is carried out by a network of clusters, each cluster running a batch system, which today handles millions of jobs. Maintaining such system with an ever increasing workload presents major challenges to the system engineers, one of the challenges being the many files that are managed by ARC.

ARC, developed by NorduGRID, is a middleware designed to manage such batch systems. The clusters that are affiliated with CERN are equipped with ARC, which also acts as an interface from the user to the batch system. The many tasks of ARC include pre- and post-processing the incoming jobs, preparing them for the batch system, which in turn will carry out the necessary computations. When done, the system will relay the job output back to the sender. As the number of jobs increase, the performance of ARC decreases, efficiently causing a bottleneck for whole job procedure. The underlying problems could be many in theory, and in this report, the focus will be on the file read performance of ARC.

Hitherto, the developers of NorduGrid already has conducted internal investigations in hopes of isolating the problem causing the read performance deterioration. Since some components of ARC are mainly working with very small and structured data in large volumes, the developers are hoping of adapting the architecture to a database solution, but the current design of ARC would require major redesign and reconstruction of the ARC architecture to fit a database, which is why they seek a file system interface on top of a database, simply put a database-backed file system.

To evaluate and somehow relate the performance of the database-backed file system, it will be matched against two popular conventional file systems, ext4 and XFS. By looking at the differences of the performance, it might also shed some light on the reasons of the declining file read performance. The ultimate goal of this study is to test if database-backed file systems are viable alternatives for ARC.

1.2.2 Research questions

This report will study some of the database-backed file systems and test one in relation to some conventional file systems and possibly unveil bottlenecks in the middleware ARC by measuring the performance with different file systems containing a lot of small files. The research questions of this thesis are:

1. What are some possible factors that are bottlenecking ARC in terms of read performance?

2. What is the performance of ARC when reading and processing small files

(a) with a file system with database backend?

(b) with a conventional file system such as ext4?

1.2.3 Study scope

There are possibly other solutions which will solve the read performance problem, but this study will mostly focus on benchmarking off-the-shelf database-backed file systems without altering the source code of ARC. To clarify, the following are not within the scope of this thesis:

- Refactoring ARC source code to fit a database interface.
- Refactoring ARC source code in order to change the ARC middleware software architecture.
- Providing a better performing file system than the current file systems of the computers equipped with ARC.

1.2.4 Contributions of this study

Database-backed file systems are not a common alternative for file storage and therefore not many studies have been conducted on them in comparison to conventional file systems. This study contributes by showing the search process of a database-backed file system as well as its performance in relation to conventional file systems when using ARC, which will contribute to the following:

- The contribution of presenting the search process lies in the feature comparison of the different database-backed file systems available off-the-shelf. There are many properties that need to be considered when choosing a database-backed file system, such as file transparency and underlying technology, and this study will present and evaluate some of these properties of each found file system.
- The benchmarking compares the read performance of ARC when using database-backed file system and conventional file systems, which in this study is ext4 and XFS. This is done by storing different amounts of small files in each of the file systems and executing ARC.

1.2.5 Report overview

In short, several database-backed file systems will be evaluated and one of them will be chosen as a *candidate*. A server mounted with the target file systems (XFS, ext4 and the candidate) is tested by storing a different amount of small files in each file system. The following paragraphs provide a short description of each chapter:

Chapter 1: Introduction Chapter 1 describes the background of this study together with the framework of the study, problem statement and related work.

Chapter 2: File systems and ARC Chapter 2 introduces some relevant file system concepts together with the brief introduction to the conventional file systems ext4 and XFS. An overview of ARC, the subcomponent which is being tested and how files are read is also given here.

Chapter 3: Method Chapter 3 explains the approach and methods used to (1) find a database-backed file system candidate and (2)to test the database-backed file system.

Chapter 4: Search process outcome and test results In the fourth chapter the process of searching for a candidate is described together with the performance results of the test runs.

Chapter 5: Discussion In the final chapter, discussions such as source criticism and speculations of the test results can be found along with the conclusion of this study.

1.2.6 Related work

As earlier mentioned, not many studies have been conducted on database-backed file systems. The following lists show some relevant file system studies, the first one being a performance study of an actual database-backed file system and the two latter being a study of FUSE, a common tool used when interfacing a database with a file system.

1. *The Design and Implementation of The Database File System* by Murphy et al. (2002): This report shows the design and implementation of a database-backed file system called DBFS. DBFS has adapted the POSIX file system interface on top a Berkeley DB, allowing software to access files with little code change. While not directly related to this report, it shows a proof of concept of a well-performing database-backed file system already back in 2001. In the report, the read and write performance of DBFS tested and it fares well in comparison with Berkeley Fast File System (FFS).

2. *Comparison of kernel and user space file systems* by Duwe (2007): Using File System in User Space (FUSE) is one way of interfacing a database with a file system. The report tests the performance of FUSE by comparing it with other file systems such as memfs. Although FUSE requires an extra overhead when accessing the file system (Krier and Liska, 2009, pp. 6-7), the report shows that it still can perform well under some circumstances. Looking at the test environment of the study, it mainly focuses on larger file sizes and observing kernel operations. This shows that if a database-backed file system uses FUSE to create a file system interface, it might be a viable option even if there is an overhead caused by FUSE.

3. *To FUSE or Not to FUSE: Performance of User-Space File Systems* by Vangoor et al. (2017): Continuing with FUSE, the report tests the capabilities of an optimized FUSE by comparing the read and write performance of ext4 with and without FUSE on top. The results are promising, showing that FUSE can perform almost as well as a native ext4 in some cases.

Chapter 2

ARC and file systems

One of the many tasks of ARC is to manage the metadata of the jobs submitted to the cluster. The reading of the job metadata files, which is the main subject of this study, is done in a rather simple way: metadata files are stored in a directory and then read periodically using Perl scripts. However, the set up of the benchmarking is rather delicate, since the tests are purely done through ARC. This means that the file systems, files and test execution need to function with ARC.

Section 2.1 will give an overview of ARC, introduce the ARC information system called *infoprovider subsystem* and explain how files are read by this subsystem. Since this is not a raw benchmarking study, it is important to know how ARC handles the metadata files to understand what the results actually show. Some technical details are omitted for a more streamlined reading, these details can be found in Appendix B.1. The information about ARC is taken from *ARC Computing Element, System Administrator Guide*, written by Paganelli et al. (2016).

To help understand what happens under the surface when data is accessed in a file system, section 2.2 will go through file systems fundamentals along with a short introduction to ext4 and XFS with a focus on the block structures.

2.1 The ARC middleware

2.1.1 ARC overview

ARC, short for Advanced Resource Connector, is an open-source grid computing middleware used to create grid infrastructures and enables services such as sharing, federation of computing and resource storage distributed across different administrative and application domains. It is developed by NorduGrid and has been in active use by organizations since 2002.

ARC consists of three main components, namely the Computing Element (CE), the Storage Element and the Indexing Service. CE per se works as a layer between the computing resource (typically a local cluster) and the designated client tools. These tools can be used to retrieve information about a resource, query, submit and manage computing jobs. On a higher level, ARC also serves as an abstraction layer between the clients and a local cluster, concealing the advanced computing architecture and processes involved when handling a job.

The two other components of ARC provides storage and indexing (linking) of several computing resources. These are not used in this study and will not be further discussed.

CE is a component which manages jobs in a number of different ways and consists of several subcomponents which all provides job services. As a whole, tasks of the CE includes advertising the local cluster's capabilities and location to clients, accepting job execution requests, processing jobs through the execution service (A-REX) and forwarding a submitted job to a local batch system.

2.1.2 A-REX

The main component of ARC CE is the A-REX (ARC Resource-coupled EXecution Service). A-REX executes jobs in the underlying local batch system and also handles the pre- and post-processing of the jobs. The pre-processing prepares the job for the batch system, which involves downloading the necessary files and data to execute a certain job. The metadata for a job is created and stored in the directory called the *Control Directory* (hereby referred as CD) of A-REX. Each job is assigned a unique ID, and every associated file is conventionally named after a schema containing the ID. Listing 2.1 shows a sample of this naming convention of job files containing metadata in the

CD. The extension of the metadata files describes its contents.

Listing 2.1: A sample of job metadata files created by A-REX when a job is submitted. The files are located in the CD. Each job generates 11 metadata files.

```
job.2fQtGR1TurLF3XB8vJf58J26HPZUAUJIu2NF3UPOvRq9UxrTaYcUJi.description
job.2fQtGR1TurLF3XB8vJf58J26HPZUAUJIu2NF3UPOvRq9UxrTaYcUJi.diag
job.2fQtGR1TurLF3XB8vJf58J26HPZUAUJIu2NF3UPOvRq9UxrTaYcUJi.errors
job.2fQtGR1TurLF3XB8vJf58J26HPZUAUJIu2NF3UPOvRq9UxrTaYcUJi.grami
job.2fQtGR1TurLF3XB8vJf58J26HPZUAUJIu2NF3UPOvRq9UxrTaYcUJi.input
job.2fQtGR1TurLF3XB8vJf58J26HPZUAUJIu2NF3UPOvRq9UxrTaYcUJi.input_status
job.2fQtGR1TurLF3XB8vJf58J26HPZUAUJIu2NF3UPOvRq9UxrTaYcUJi.local
job.2fQtGR1TurLF3XB8vJf58J26HPZUAUJIu2NF3UPOvRq9UxrTaYcUJi.output
job.2fQtGR1TurLF3XB8vJf58J26HPZUAUJIu2NF3UPOvRq9UxrTaYcUJi.proxy
job.2fQtGR1TurLF3XB8vJf58J26HPZUAUJIu2NF3UPOvRq9UxrTaYcUJi.statistics
job.2fQtGR1TurLF3XB8vJf58J26HPZUAUJIu2NF3UPOvRq9UxrTaYcUJi.finished
```

2.1.3 The infoprovider subsystem

The infoprovider subsystem is a collection of Perl scripts which gathers information about the batch systems, jobs and users. When a job is submitted, the metadata for it is generated and the infoprovider subsystem periodically run scripts that read the metadata files located in the CD. This is done in order to gather information about the job status. The most important scripts for this study are the following:

CEinfo.pl The main script is used to initiate the job information collection by running and coordinating scripts.

GMJobsInfo.pm The module provides the subroutine `collect()`, which gathers job information by reading the metadata files stored in CD. As earlier mentioned, 11 files are generated for each submitted job, however, `collect()` attempts to read 6 of them and only 5 succeeds. One of them is missing because the job is never actually executed.

2.1.4 Reading metadata files

The whole reading procedure is started by running `CEinfo.pl`, which will start running other necessary scripts and start the `collect()` subroutine of `GMJobsInfo.pm`.

`collect()` scans by firstly retrieving the ID of all the jobs in CD and storing it in a list as strings. For every ID in this list, the metadata filenames are constructed by appending the metadata extensions to the ID. The metadata file is then directly accessed and its contents scanned with respect to what extension it has. This is repeated for every ID and every extension. Listing 2.2 shows this algorithm in pseudocode. For the source code of `GMJobsInfo.pm`, see A.2.

Listing 2.2: The listing shows the algorithm used to read and process the metadata files.

```
Store all IDs in idlist
for each id in idlist
        Append id with ".local"
        Open id.local
        Scan contents of id.local

        Append id with  ".diag"
        Open id.diag
        Scan id.diag

        ... Repeat until all 6 metadata files has been processed
for end
```

2.1.5 Performance deterioration

The infoprovider subsystem is stateless, which means that every scan of the CD has no memory of the previous scan. The scripts only read the metadata files in the CD and do not modify (write) them. The infoprovider subsystem reads the job status in a timely basis, so if a client were to query a submitted job status, the infoprovider would return the latest status report of the job. While being very simple in design, this poses several challenges to the subsystem and its developers, such as managing the large amount of metadata files when there are many jobs submitted. Moreover, inexplicable slowdowns in reading the CD has been noted by the developers. In some cases, the performance gradually becomes even worse than the initial read speed.

2.2 File Systems

Basic structures of file systems are introduced here to describe what theoretically happens under the surface when a file is being accessed in a file system. Since the infoprovider subsystem works heavily with files, with the information from this section it will be easier to reason and understand the trends of the test results.

The source of the file system theory presented in this section is gathered from the second chapter in *Practical File System Design with the Be File System* by Giampaolo (1998), if not specified otherwise.

2.2.1 Basic structures of a file system

Files and inodes

File systems abstract data in form of files. A file is in its simplest form a collection of data, stored in a specified location of the disk. While a file itself contains user data, there is normally an inode associated with that file. What data an inode exactly contains varies from file system to file system, but it normally contains metadata such as file size, owner, date of creation and the last date of modification. The most important task of an inode is to keep track of the file block locations since the blocks composing file data are not necessarily physically close to each other.

Directories

Directories are an important structure of a file system. For ARC, this is especially important since the CD is, in fact, nothing but an ordinary directory. A directory provides a mean to manage and collect files into a single structure. The implementations of directories differ depending on file system, but essentially a directory contains a list of file names. The name of the file name is the key whereas the associated inode is its value.

There are different ways of arranging the file names in a directory. Some file systems keep the keys completely unsorted, whilst others (such as ext4 and XFS) use more sophisticated data structures like B-trees.

Most DBMS and file systems today use B-trees variations to organize files in a directory. B-trees are very popular due to the decreased need for performing expensive I/O operations on a disk.

In short, B-trees are a generalization of a binary search tree (BST), which means that a B-tree allows more than two paths from a node, expanding the amount of data that can be stored on a node. This is especially beneficial when used in disk storage because if a file system loads a large node into the memory, it can reduce necessary I/O operations on disk, assuming that the subsequent reads only requires data of the same node. The larger node size also decreases the depth of the tree, which means that a data query in worst-case only needs a few I/O operation on disk.

In a B-tree, key values and references are stored in the nodes. When querying data, the process is similar to that of a BST. Due to the larger nodes, the differences lie in the decision making at each node. Since there are more key values stored in each node, more decision making has to be done before proceeding to the next node if the queried data was not found in the current node (Comer, 1979, pp. 123-124).

There are other variants of the B-tree, such as the B+-tree and HTree. They typically follow the same idea of a generalization of BST, but they differ in other aspects. One of the reasons B+-tree was developed was due to the relatively poor performance of sequential reads of B-trees (Comer, 1979, pp. 128). The B+-tree addressed this by changing the node structure. In a B-tree, each node contains keys to a corresponding data value. In a B+-tree, the data values are all moved to the leaf nodes. The keys are still present in each node, but the internal nodes of a B+-tree do not contain any data values. The keys in these nodes serve as an intermediate step for reaching the leaf nodes. In a B+-tree, the leaf nodes are also conveniently linked to each other, and since all data is stored in the leaf nodes, all the key values and data values of a B+-tree are linked to each other. In theory, this improves the performance of sequential reads of spatially close data since they are stored as a linked list (Comer, 1979, pp. 129-130).

2.2.2 Initializing a file system

Before using ext4, XFS or the candidate, they need to be somehow initialized and mounted. For conventional file systems, deciding how large the components of a file system, such as block size, partition size and superblock size is a part of this initialization. After these parameters have been decided, a top-level directory, commonly known as root, will also be created. Finally, the file system needs to be mounted in order to access its contents. What the OS does when accessing the mount point depends on which OS that is being used, but generally the OS will read the metadata provided by the file system, thus unveiling its contents.

2.2.3 ext4

Ext4 (ext short for "extended file system") is a journaling file system and it is the fourth version of the extended file systems, which was first introduced in 1992 as simply ext.

Looking at the block structure, ext4 is divided into smaller groups of blocks, simply called block groups. To reduce fragmentation and achieve faster access times, the block allocator makes an attempt into putting the blocks of the same file in the same block group. Each block group has a group descriptor, containing metadata for that particular block group, such as which blocks included in the group and the range of inodes (Fairbanks, 2012, pp. 123-124).

To list the files in a directory, ext4 uses a modified B-tree called HTree, which hashes its key values and has a constant tree depth.

2.2.4 XFS

XFS is a high-performance journaling file system developed by Silicon Graphics, Inc. It was designed with parallel I/O and large data sets in mind. This gives XFS an advantage over other file systems since it was designed for large systems from day one. Many features, such as extents and delayed allocations, were pioneered by XFS and are used in other modern file systems such as ext4 (Hellwig, 2009, pp. 10-12).

XFS uses allocation groups, which are equally sized chunks of blocks, similar the block groups to ext4. Although they might look the same externally, they work differently internally. Allocation groups can be seen as a smaller file system, managing their own space allocations and dynamically allocates inodes. This was designed to support large multi-threaded file system operations. An allocation group is typically larger than an ext4 block group, ranging from 0.5 to 4.0 GB.

The large groups of blocks could cause problems when dealing with either very large files or small files. When it comes to the small files XFS tries to chunk files of the same directory in the same allocation group. To achieve this, a directory is allocated on a different allocation group from its parent. This allows the file system to cluster blocks and inodes of the same directory (Sweeney et al., 1996, no pagination). To list the inodes in a directory, XFS uses B+-tree structure (Sil, 2006, pp. 19-29).

Chapter 3

Method

This study is separated into two phases: (1) searching, studying and picking a database-backed file system to test (a candidate file system) and (2) installing, configuring and benchmarking one of the found software solutions together with ext4 and XFS. Section 3.1 will explain the approach and search criteria when looking for a database-backed file system and Section 3.2 describes how the test environment, such as managing the metadata files, is set up.

3.1 Phase one: Searching for a database-backed file system

Searching for a database-backed file system is done in three different ways: (1) by looking at different tools and libraries that could enable such technology, (2) searching in different source code repositories and (3) using general and academic search engines.

It was noted early on that the term *file system* is a broadly used term and can refer to many different types of file systems. On one end there are conventional file systems such as ext4, XFS and NTFS, whilst on the other end there are distributed file systems. Some file systems fall outside of this spectrum, among them are the ones that are called file systems but are technically APIs. This broad definition complicates the search process, since simply searching for a file system will give a huge variety of different software solutions. To narrow the search domain, the needs and requirements of ARC were identified and the following criteria were established:

1. **Transparency**: In this study, transparency refers to the ability to use system calls for I/O file operations, meaning that there will be no need of altering the ARC source code to fit the database-backed file system. The scripts running the infoprovider subsystem needs to be able to transparently read the files without need of code alteration, without knowing the specifics of the underlying file system.

2. **Active development**: If the developers of NorduGrid were to consider a database-backed file system, active development of the database-backed file system is highly advantageous due to available support and stability. However, for this study it would suffice if the file system candidate worked with the latest kernel but if it is no longer in development it will less likely to become a candidate.

3. **Independent of other services**: The file system should not rely and be maintained on any other third party organization or company. The file system should be able to be installed and run on a single computer.

4. **Deployable**: The file system must work on Linux, since the servers running ARC are using Linux. If the file system setup and configuration is simple and a desktop environment is not needed, the file system would be higher prioritized. Other things to consider are the libraries needed for a found file system. Too many libraries could cause complication when installing it on servers running ARC.

While not a criterion, the fact that the ARC CD lies on a single computer must be considered when choosing a file system candidate. What this means for the search is that other than the file system working on a single computer, it should also be viable for simple storage of small files.

3.2 Phase two: Test specification

3.2.1 Overview

The file systems are tested by creating CDs containing different amount of job metadata files, storing them in each file system and then running the infoprovider subsystem. The subsystem is patched with a performance logger, which calculates the reading time for each file read. This process involves several preparations, which all are explained in this section. In short, the subtasks of the preparation are:

1. Deciding amount of jobs to be read.
2. Generating metadata files to a predetermined amount by copying the metadata files of a dummy job.
3. Storing different amount of metadata files in different CDs.
4. Storing the aforementioned CDs in every file system to be tested.
5. For every CD in each file system:
   (a) Setting the target CD in the ARC configuration file.
   (b) Running the infoprovider subsystem to scan the files of the target CD.
   (c) Gather the performance log generated by the infoprovider subsystem.

Technical details, such a description of the scripts used and metadata generation can be found in Appendix 3.

3.2.2 Amount of jobs and metadata files

Each cluster typically handles sub-40K jobs at any given point of time, but to put the file systems to test, it was decided that the files systems were going to be tested with up to 160K jobs. Note that here, 160K jobs refer to generating metadata file of 160K jobs, not reading or executing 160K jobs. To see how the performance correlates with a number jobs, the volume is doubled for each test run, meaning that starting with 5K jobs, the file systems are tested with 5K, 10K, 20K, 40K, 80K and 160K jobs.

To further study the capabilities of the file systems, for each file system the tests are run on a solid state drive and a mechanical drive.

Table 3.1 shows the number of files in the CD of each test run. In the table, the rows describe how many jobs that generated the metadata, whereas the columns describe storage type and file system for each test run. This totals in 36 test runs. In terms of organizing the directories, each test run

Table 3.1: The full test specification. *Candidate* refers to the candidate file system chosen in Phase one.

	Amount of jobs (in thousands)					
	SSD			*HDD*		
Run	*XFS*	*ext4*	*Candidate*	*XFS*	*ext4*	*Candidate*
5K runs	5	5	5	5	5	5
10K runs	10	10	10	10	10	10
20K runs	20	20	20	20	20	20
40K runs	40	40	40	40	40	40
80K runs	80	80	80	80	80	80
160K runs	160	160	160	160	160	160

has its own CD, meaning that all things considered there are in total 36 CDs distributed among the different file systems and hard drives.

As for how many files that are actually stored and read in each test run, for the 5K job, 11 files are generated, whereas only 6 are attempted to be read. For example, for the 5K run, the corresponding CD is stored with roughly 55K files, and around 36K of these files attempted to be read by the infoprovider subsystem.

3.2.3 Generating a job and metadata for the test run

It is important to stress that this study revolves around fitting and testing file systems with ARC and is not a raw benchmarking. The metadata must conform to a syntax which the infoprovider subsystem can parse.

To generate such metadata, a tool called *arctest* was used, which is bundled with ARC. This was used to generate one job, which will in turn cause ARC to generate the metadata. The metadata files for this single job are then copied to generate metadata for a new job. Therefore, the contents of the generated metadata files are all the same, the only property that needs to be different is the ID of the job, which is expressed in the filename.

The metadata generation was done in an incremental fashion, meaning that the CD containing 5K jobs is a subset of the CD containing a larger amount of jobs. This was done to ensure that the infoprovider subsystem reads the same set of files, regardless of volume.

Chapter 4

Search process outcome and test results

In Chapter 3, the search criteria for the file system candidate is established in Section 3.1 and the test environment is described in Section 3.2.

In this chapter, the outcome of the file system candidate search is presented and Section 4.1 and hardware specifications for the test runs can be found in Section 4.2. Finally, the results from the test runs are shown in Section 4.3.

The technical configurations of the database-backed file system and the properties of the generated CDs, such as size, can be found in Appendix B.2.

4.1 File system candidate

4.1.1 Search process

The open-source community has created a handful of database-backed file systems, however, the majority of these are small-scale projects developed by a single developer and are no longer maintained or further developed. Looking at the enterprise-level database-backed file systems, some of them are not transparent for the OS or are focused on distributed systems. Since the size of the CD is relatively small and is locally stored on a single computer, the distributed file systems were considered too grand-scaled for this study. Other databased-backed file systems do not handle small files very well.

While searching for a database-backed file system, it was noted that many databased-backed file system uses FUSE. FUSE has the property that it can represent files in a transparent way, which means the OS will treat the underlying data that interface with FUSE as if it was any other file stored in a conventional file system. This file system property is one of the most important ones in this study since the developers of NorduGrid wishes to avoid larger code alterations of ARC.

In short, the database-backed file systems that ultimately were discarded as a candidate file system had the one or more of the following drawbacks:

1. Outdated and no longer in further development.

2. Missing I/O functionality, such as writing or reading.

3. Lack of transparency.

4. Integrating to ARC requires a source code or architecture overhaul.

The list of considered database-backed file systems can be found in Table 4.1. The file systems listed are the ones that were initially considered before researching deeper into their capabilities and functionality. In the table, each column represents a criterion described in Section 3.1, where *TR* stands for transparency, *AD* for active development, *IN* for independent of other services, and *DE* for deployable. If a database-backed file system fulfills the criterion, the corresponding checkbox is ticked. Due to the nature of software development, the information provided may be prone to change.

The database-backed file system marked with a star (*) utilises FUSE as a file system interface.

4.1.2 The chosen file system candidate

The chosen candidate file system was ultimately decided to be *Database File System* (DBFS), developed by Oracle. It was chosen due to it fulfilling all of the criteria described in Section 3.1 and due to the popularity of Oracle products. DBFS uses FUSE[1] to create a file system interface with Oracle DB, an RDBMS, as its backend. In this study, the free version of Oracle Database 12c Enterprise Edition was used.

FUSE (Filesystem in Userspace) is a two-part interface which helps developers to develop filesystem in userspace. The benefits of developing with

[1]Libfuse repository: https://github.com/libfuse/libfuse

Table 4.1: Considered database-backed file systems

Name	*TR*	*AD*	*IN*	*DE*
GridFS	☐	☑	☐	☐
Comment: While initially looking promising candidate, GridFS is in fact an API despite it being named a file system.				
Link: https://docs.mongodb.com/manual/core/gridfs/				
CouchDB-fuse*	☑	☐	☑	☑
Comment: Uses very old libraries that would not compile with CentOS 7.				
Link: https://code.google.com/archive/p/couchdb-fuse/				
Libsqlfs*	☑	☑	☑	☑
Comment: Libsqlfs was another potential candidate. DBFS was chosen due to having the more popular database.				
Link: http://www.nongnu.org/libsqlfs/				
Postgresqlfs*	☑	☐	☑	☐
Comment: A small project which is no longer in development.				
Link: https://github.com/petere/postgresqlfs				
BerkeleyDB /w FUSE*	☑	☐	☑	☐
Comment: Uses old libraries and is thus outdated.				
Link: https://git.kernel.org/cgit/fs/fuse/dbfs.git/about/				
Database File System	☐	☐	☑	☐
Comment: Database File System seems to be a GUI application to access files in a database, thus there are many uncertainties regarding its transparency.				
Link: http://dbfs.sourceforge.net/				
Mongofuse*	☑	☐	☑	☐
Comment: This project was developed in 24 hours as a part of a competition and was thus not considered reliable.				
Link: https://github.com/asivokon/mongo-fuse				

FUSE is that it provides a way for a user to develop a file system without editing the kernel code, effectively hiding the complex kernel architecture. The first part of FUSE is the fuse kernel module and the second part is the libraries which are an API for the file system. The fuse kernel module registers a fuse device which will serve as an interface for the FUSE userspace libraries. When a user makes a FUSE file system operation, the *Virtual File System* (commonly referred as VFS) will receive this call and direct it towards the fuse kernel module. This call is then put on a FUSE queue by the module and will eventually be computed by a FUSE daemon (Rajgarhia and Gehani, 2010, no pagination).

DBFS itself is actually a tool used to create a conventional file system interface on top of Oracle DB. The database data can be accessed either through PL/SQL or by mounting through FUSE.

When using DBFS, data is stored as segments in a tablespace, which is a logical representation of the data. Segments are any database objects, such as tables and indices. The data stored in a tablespace is physically stored in a data file located on the disk (Ora, 2017, no pagination).

When storing files through DBFS, the data is stored with a schema just like any other data in the database. This schema describes the file metadata and the contents of the file are stored as a BLOB.

4.2 Server specifications

The computer used is a server-level computer with 4 GB of RAM, two HDDs and one SSD. One of the HDD hosts the system OS and the Oracle RDBMS, whilst the other HDD and SSD were each partitioned into three roughly equally sized partitions. Two of these were formatted with ext4 and the last one with XFS. One of the ext4 stored the datafile for DBFS. Table 4.2 shows the full server specifications. When creating the partitions, the block size was set to the default size of 4096 B.

4.3 ARC performance results

The performance data gathered from the infoprovider subsystem are presented in this section. The main goal is to highlight the potential factors that affect the read performance of the target file system, in particular DBFS.

Table 4.2: The specifications of the server used to run the tests.

Server part	*Model name*
OS	CentOS 7
Motherboard	Intel Desktop Board DX79TO
CPU	Intel Core i7-3820 CPU @ 3.60GHz
RAM	Samsung DDR3 4096 MB
Storage (System OS)	Seagate ST31000340NS 1TB
Storage (Test HDD)	Western Digital WDC WD10EZRZ-00H 1TB
Storage (Test SSD)	Intel SSDSA2BW12 120GB

The results will abstract the notion of files to jobs instead, meaning that the performance is expressed as ”jobs per second” rather than ”files per second”. This way, it is easier to correlate a job submission with the performance.

4.3.1 Average read rate

The average read rate is measured in jobs per second, shown on the Y-axis, and includes the performance data gathered from the transient state of the system.

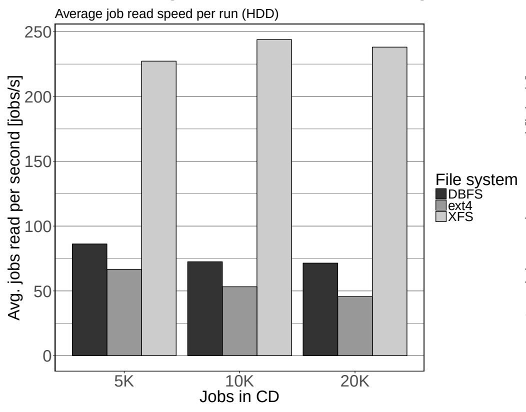
When using an HDD, the graphs shown in Figure 4.1 shows that XFS outperformed ext4 and DBFS for all the given amounts of jobs. Ext4 was especially susceptible to an increased job volume, suffering about a considerable performance degradation in relation to the other file systems. DBFS and XFS, on the other hand, performs relatively well regardless of file volume. While DBFS performed well in comparison with ext4, it is still slow with a read rate of roughly 80 jobs per second.

As for the test runs performed on the SSD, the graphs in Figure 4.2 shows that when using SSD, the performance of ext4 boosted significantly from its HDD counterpart, performing roughly at the same speed as XFS. While both ext4 and XFS benefited from SSD, no considerable performance gain can be observed on the DBFS.

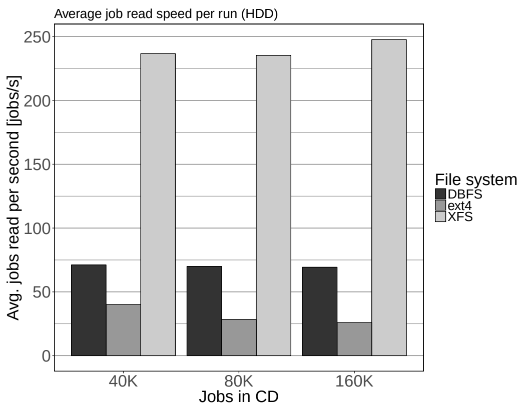
4.3.2 Performance graphs

This section contains the graphs showing the performance trends during a test run. Each graph of this section shows a number of jobs read for a unit

Figure 4.1: The average speed of a full test run on an HDD.

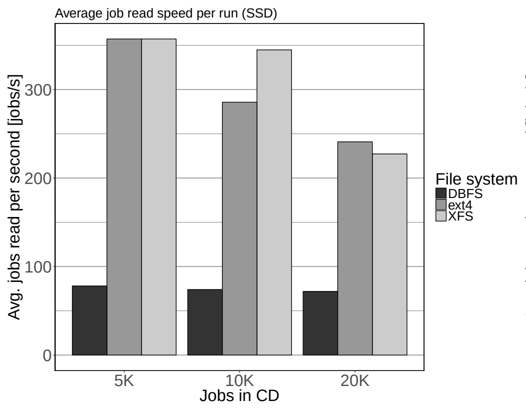


(a) 5K, 10K and 20K jobs

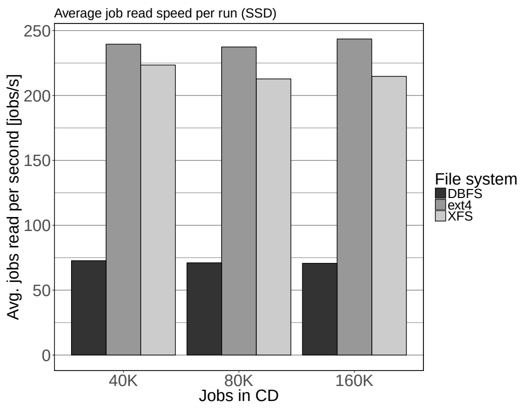


(b) 40K, 80K and 160K jobs

Figure 4.2: The average speed of a full test run on an SSD.

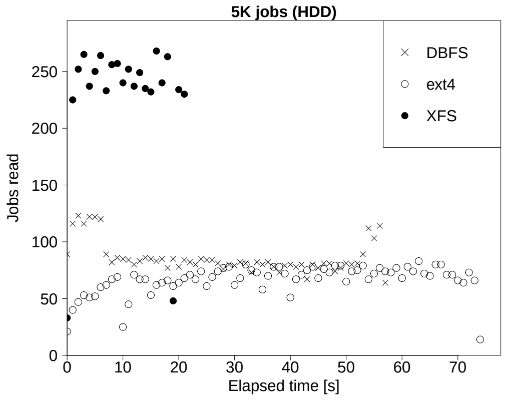


(a) 5K, 10K and 20K jobs

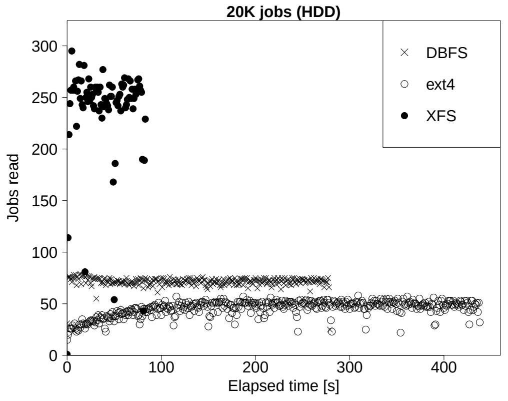


(b) 40K, 80K and 160K jobs

Figure 4.3: The performance data of a test run with 5K and 20K jobs stored on an HDD



(a) 5K jobs



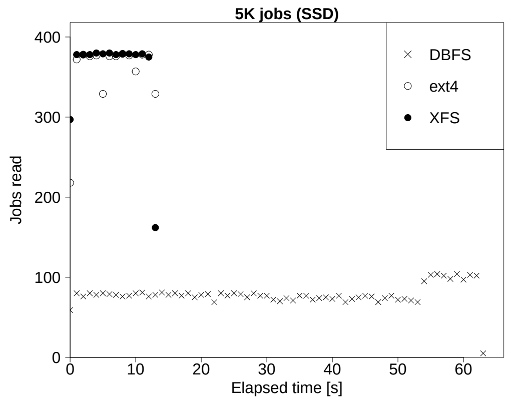
(b) 20K jobs

of time. In the graphs that show results from the lower volumes (5K, 10K, 20K), the Y-axis describes the jobs read each second. In the graphs that shows the results from the higher volumes (40K, 80K and 160K), the Y-axis shows average jobs read per second for a given minute. This aggregation of performance data was done to avoid data point cluttering due to the large amount of performance data. The focus in this section will be on the lower volumes, graphs of the larger volumes can be found in Appendix C and were moved there because the performance trends of the larger volumes test runs were similar to the lower ones.

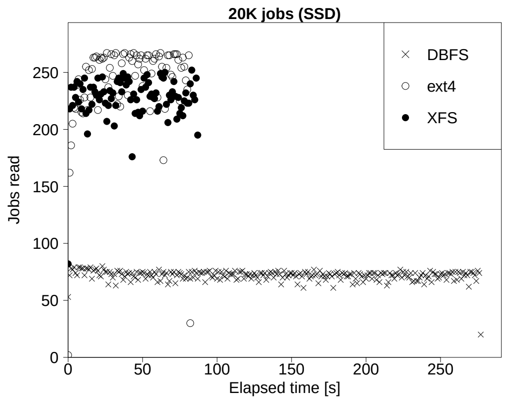
The graph in Figure 4.3a displays the results from the test run of 5K jobs on the HDD. It shows that while all the target file systems has a stable performance after reaching the steady state, XFS shows a slightly turbulent performance in relation to DBFS and ext4. DBFS has a decreasing performance in its transient state while ext4 has an increase in performance and remains longer in its transient state. Even though the size of the CD is increased by twofold for each run, the transient response does not change much. Similar performance trends can also be observed on the test runs of a larger CD, as seen on the graph in Figure 4.3b, which shows the results from the test run of 20K.

Looking at the performance of the 5K and 20K jobs on the SSD in Figure 4.4a and Figure 4.4b, the graphs shows us that the relatively turbulent performance of XFS is still persisting in the SSD test runs and that ext4

Figure 4.4: The performance data of a test run with 5K and 20K jobs stored on an SSD



(a) 5K jobs



(b) 20K jobs

no longer suffer from a long transient response. Moreover, the ext4 shows the relatively turbulent readings as seen on the results from XFS, but it is slightly more consistent than XFS. As for the DBFS, no notable performance trend differs from its HDD variant.

Chapter 5

Discussion

5.1 Source criticism

The sources used are from proceedings of different conferences, academic reports and scholarly literature. The reliability is considered to be high, although there is a risk of some information being outdated. This is mostly due to the volatile open-source development of software. For instance, the official documentation of XFS was lastly updated over a decade ago. Surely, some components have changed. However, a complete architecture revamp would most likely generate new reports that describe the changes.

Some of the discussed technologies, such as FUSE, completely lack an official documentation of how it works internally. This was solved by reading the background section of different academic reports that study this technology.

5.2 Finding a databased-backed file system

It has become more clear that even if the high level concept of storing data in a database or a file system are somewhat similar, they are difficult to compare on a one-to-one basis, let alone comparing databases with other databases and file systems with other file systems. The reason for this is mostly the differences in the underlying technology. For instance, XFS uses a blocking scheme similar to ext4, but groups blocks together to form allocation groups. How they work internally is quite different: the allocation groups of XFS works more like an independent file system and manages their own space, while the structure design of ext4 has space locality in mind.

Deducting performance outcome of each property of a file system would be too time-consuming for this study, but because of the criteria set up for the search (as described in Section 3.1, the amount of potential database-backed file systems were greatly narrowed down. This is not necessarily advantageous in search of a well-performing solution since many of the popular databases such as PostgreSQL, MongoDB and Cassandra were excluded.

5.3 Analysis of the performance data

Comparing the performance of ext4 with DBFS on an HDD, DBFS has a slight advantage when it comes to read performance. In the conducted tests XFS outperforms both in any case. The causes of this are difficult to exactly pinpoint with the theory presented and the experiments conducted, but since directories and files were some of the main subjects of the tests one could argue that the directory structure of XFS greatly benefited its read performance. One of the main design goals of the B+-tree was the fact that the sequential reading of B-tree was poor (Comer, 1979, pp. 128). In theory, the B+-tree structure should give XFS a greater advantage over ext4 since the XFS attempts to store files with the same parent directory into a common allocation group. When reading sequentially, access times might have been reduced due to the linked leaf nodes. Simply put, XFS might use available memory better than ext4.

As for the DBFS, the underlying storage technology has little effect on the performance. While performing well in comparison with ext4 in the HDD case, it is outclassed by XFS. Another aspect that has not been discussed earlier is the write performance of DBFS. Poor write performance has been observed when storing the many job files into the DBFS mount points. Nevertheless, DBFS has shown good scalability properties, with little performance degradation with respect to the CD volume. There is a possibility of the FUSE system calls causing a bottleneck for DBFS but that is difficult to prove without testing the performance of the database without FUSE. Whether FUSE was automatically optimized for reading small files in the experiments is unclear.

5.4 Proposed solution for ARC servers

This study shows that a database-backed file system does not necessarily improve the read performance of ARC. The process of trying to find other database-backed file system might be too time-consuming and thus not be feasible, due to the many different flavours of this technology.

Without changing the architecture of ARC, one way that in theory should increase the performance of the servers using XFS would be to separate the different kinds of metadata files into smaller directories. The results show, the increased file volume in the directory has a negative impact on the performance of the conventional file system.

As a result of this study, these proposals have been accepted by the ARC developers team and are planned for future releases of ARC.

5.5 Conclusion

When considering a database-backed file system as an alternative storage, one has to have a look at many factors due to the varying underlying technology. In this study, DBFS, a file system with a FUSE interface on top of an RDBMS, was chosen as a database-backed file system candidate for testing due to its active development and ability to transparently present files to the OS.

When reading many small files in a directory, the database-backed file system DBFS does not perform well when compared to a high-performing file system such as XFS. The results show us that XFS performed about 4-5 times better than DBFS with directories containing up to 160 000 jobs, which is roughly 1.6 million files. In spite of its lacking performances in relation to XFS, the read performance of DBFS on an HDD showed good results when compared to ext4, showing a consistent performance even with a highly increased file volume.

The storage technology used had a great impact on the read performance of the conventional file system, especially for ext4. The read performance of DBFS however, remained unchanged even when changing from HDD to SSD. The reasons for this are not clear and it would require more benchmarking to narrow the possible reasons, but one possible bottleneck could be the overhead caused by FUSE kernel calls each read.

Appendices

Appendix A

Code repositories

A.1 Test run scripts

Repository link: `https://bitbucket.org/Anditron/thesisrepo/src`

The scripts stored in this repository were used to set up the test environment, this includes generating the metadata files. The scripts stored in the repository are:

- **arc.conf**: The minimal configuration file used to run the infoprovider subsystem (CEinfo.pl).
- **genCDentries.sh**: The script used to generate metadata files by copying an existing one and changing its filename.
- **runtests.sh**: The script used to run the tests, is used in conjuction with changeline.sh.
- **changeline.sh**: The script used to change the lines of arc.conf. This was done in order to automatise the many test runs.

A.2 ARC source code repository

- **ARC source code repository**: `http://svn.nordugrid.org/`
- **Infoprovider subsystem scripts**: `http://svn.nordugrid.org/repos/nordugrid/arc1/trunk/src/services/a-rex/infoproviders/`

- **Modified GMJobsInfo with performance collection**: `http://svn.nordugrid.org/repos/workarea/florido_paganelli/performance/`

Appendix B

Technical details and configurations

B.1 Chapter 3

B.1.1 Metadata generation

The ARC middleware provides a test application called *arctest*, which is a tool to generate a job tasks for test purposes. As of April 2017, there are three possible realistic test jobs that can be generated by ARC.

The metadata is copied using a BASH script provided by the developers of NorduGrid. This script, called *genCDentries.sh*, can be found in Appendix A.1. The script creates more metadata, but only does so superficially by copying the contents of existing metadata and then changing the name of the generated metadata.

B.1.2 Running the infoprovider subsystem

Before running the infoprovider subsystem to scan a target CD, the ARC-configuration file, `arc.conf`, needs to be set accordingly. This needs to be done before every test run.

A minimal configuration file provided by NorduGrid was used for the runs, which can be found in Appendix A.1. The most important settings were the keys `controldir` and `perflogdir`. The value of `controldir` is the target CD, and the value `perflogdir` is the target directory of the performance logs.

There are two ways of running the infoprovider subsystems: either by starting A-REX or simply running `CEinfo.pl`. To automatize the test run process, BASH scripts were developed to automatically configure `arc.conf` and start the test runs. They are called `runtests.sh` and `changeline.sh` and can be found in Appendix A.1.

To reduce the risks of depleting all the available inodes, the 160K test runs are done lastly, after finishing the test runs on the smaller CDs, which can then be deleted to leave more space for 160K jobs CD.

B.2 Chapter 4

B.2.1 Configuring the RDBMS

Oracle's RDBMS was installed on in the System HDD, more specifically in `\u1` and `\u2` of the root directory. To query and configure the database, the database command-line utility `sqlplus` was used. A bigfile and non-autoallocating tablespace was created alongside a data file of roughly the same size as the hosting partition. In the SSD, the datafile size was set to 33 GB and in HDD it was set to 120 GB.

The SSD data file was stored in `/dev/sda3` and HDD data file in `/dev/sdc3`. To create a file system out of the data file, the script `dbfs_create_filesystem.sql` was run on the tablespace. The script is provided by the Oracle DB Installer. Using `dbfs_client`, the contents of the data files were mounted on `/arctest/test_ssd_dbfs` and `/arctest/test_mk_dbfs`, respectively.

B.2.2 The target CDs

As described in Section 3.2, the CDs were first created before copied to the mount points of the target file system. To avoid confusion and mixing of the different CDs, several directories were created where the top-level ones were the mount point of the target file systems. Each of these mount points contained CDs of different sizes. Figure B.1 shows the mount points of each device and Figure B.1 shows the hierarchy of the test directories. In the `arctest` directory, there are in total 8 mount points, where 6 of them contains CDs of different sizes and 2 contained their own data file for the DBFS. For each test run, the configuration file `arc.conf` was set to one of the CDs in Figure B.1 as target CD. For instance, a test run for a CD

Table B.1: The mount configurations of the target file systems.

Device	*Type*	*Mount point*	*File system*
/dev/sda1	SSD	/arctest/test_ssd_ext4	ext4
/dev/sda2	SSD	/arctest/test_ssd_xfs	xfs
/dev/sda3	SSD	/arctest/orcldb_mk	ext4 (Data file host)
/dev/sdc1	HDD	/arctest/test_mk_ext4	ext4
/dev/sdc2	HDD	/arctest/test_mk_xfs	xfs
/dev/sdc3	HDD	/arctest/orcldb_ssd	ext4 (Data file host)

of size 10K jobs on a ext4 mechanical drive, the target CD would be set as `/arctest/mk_test_ext4_controldir10K`. The folders `orcldb\_mk` and `orcldb\_ssd` contains the data file used for the database.

The metadata files generated are typically a few bytes to 4 KB. The size of the CDs are listed in Table B.2 and the size of the files scanned by the infoprovider subsystem is listed in Table B.3. Note that files with a `.failed` extensions are actually not generated. This is because the actual job execution is never carried out. The infoprovider subsystem will still attempt to read it.

Table B.2: The size of the generated CDs.

CD Size	*Size*
5K	221 MB
10K	421 MB
20K	822 MB
40K	1.6 GB
80K	3.2 GB
160K	6.4 GB

Table B.3: The size of the metadata 6 files scanned.

Extension	*Size*
`.local`	1.1 KB
`.status`	8 B
`.failed`	0 B
`.grami`	1.5 KB
`.description`	2.3 KB
`.diag`	437 B

Figure B.1: The test folders hierarchy.

```
/
└── arctest
    ├── mk_test_ext4
    │   ├── controldir5k
    │   ├── controldir10k
    │   ├── controldir20k
    │   ├── controldir40k
    │   └── ⋮
    ├── mk_test_xfs
    ├── mk_test_dbfs
    ├── ssd_test_ext4
    ├── ssd_test_xfs
    ├── ssd_test_dbfs
    ├── orcldb_mk
    │   └── datafile_mk.dat
    └── orcldb_ssd
        └── datafile_ssd.dat
```

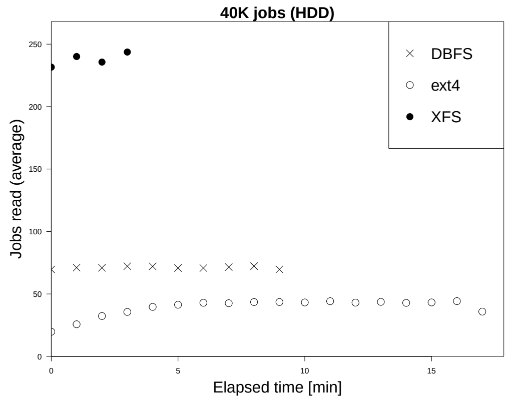
Appendix C

All results

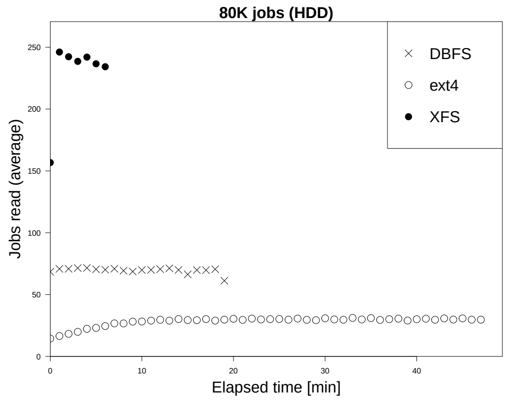
This section shows the performance graphs of the CDs containing larger volumes (40K, 80K and 160K jobs). The Y-axis describes the average jobs read per second for a given minute, which can be read from the X-axis.

The three graphs in Figure C.1 shows the results from the the test runs performed on an HDD and the three graphs in Figure C.2 shows the runs performed on an SSD.

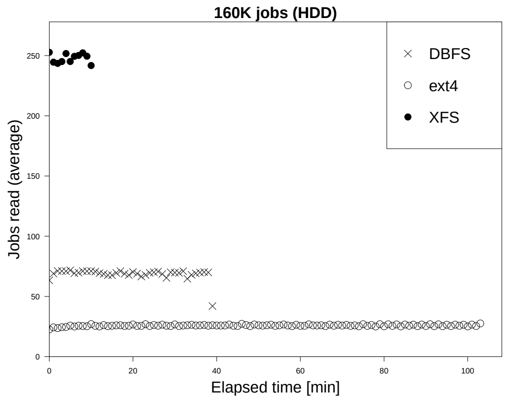
Figure C.1: The performance data of a test run with 40K, 80K and 160K jobs stored on an HDD



(a) 40K jobs

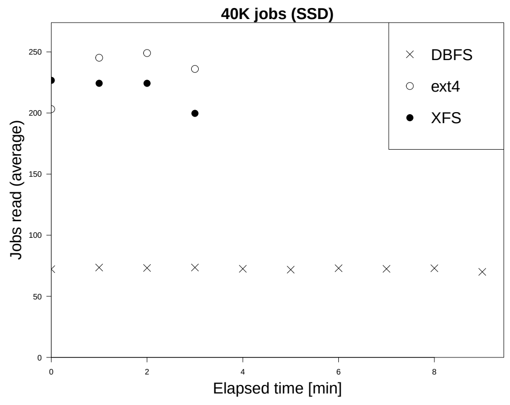


(b) 80K jobs

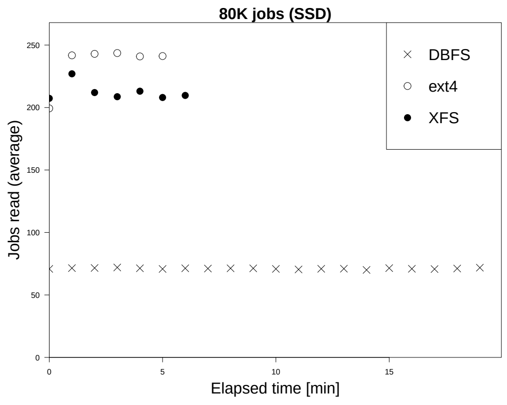


(c) 160K jobs

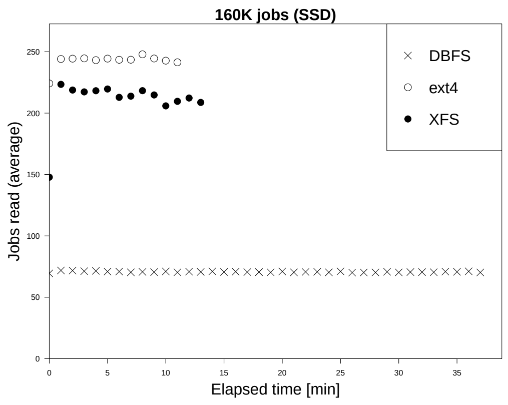
Figure C.2: The performance data of a test run with 40K, 80K and 160K jobs stored on an SSD



(a) 40K jobs



(b) 80K jobs



(c) 160K jobs

Bibliography

Douglas Comer. The ubiquitous b-tree. *Computing Surveys*, Vol. 11(No. 2), 1979.

Kira Isabel Duwe. *Comparison of kernel and user space file systems.* Bachelor thesis, University of Hamburg, 2007.

Kevin Fairbanks. An analysis of ext4 for digital forensics. In *The Digital Forensic Research Conference*, 2012.

Garcia-Molina, J. H. Ullman, and J. Widom. *Database Systems: The Complete Book*. Pearson, 2003.

D Giampaolo. *Practical File System Design with Be File System.* Morgan Kaufmann, 1998.

Christoph Hellwig. Xfs: The big storage for linux. *;LOGIN:*, 2009.

IBM. Ibm archives: Edgar f. codd. *IBM Research News*, 2003.

W. Krier and E. Liska. *FUSE Design Document.* Sun Microsystems, 2009.

A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plan. In *Proceedings of the Linux Symposium*, 2007.

N. Murphy, M. Tonkelowitz, and M. Vernal. The design and implementation of the database file system. Technical report, Faculty of Sciences and Art, 2002.

Oracle Database Concepts. Oracle, 2017.

F. Paganelli, Zs. Nagy, and O. Smirnova. *ARC Computing Element, System Administrator Guide*. NorduGrid, 2016.

Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010.

XFS Filesystem Structure. Silicon Graphics Inc., 2006.

Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, 1996.

B. Vangoor, V. Tarasov, and E. Zadok. To fuse or not to fuse: Performance of user-space file systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.