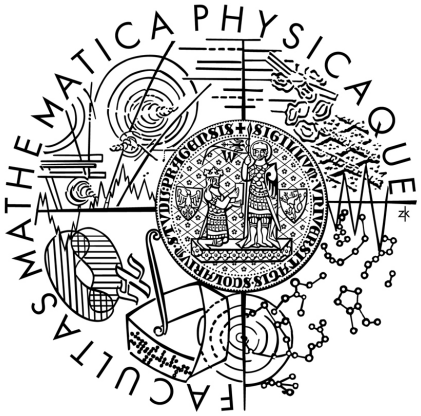
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jiří Horký

Improving Efficiency of HEP Applications

Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D.
Study programme: Informatics, Software Systems

2010

I would like to thank my supervisor RNDr. Filip Zavoral, Ph.D. for his valuable advices.

This work was partially supported by Institute of Physics of The Academy of Science of the Czech Republic (FZU) and the European Organization for Nuclear Research (CERN). Words of thanks are due to my supervisors at these institutions: RNDr. Miloš Lokajíček, CSc (FZU) and Phd. Roberto Santinelli (CERN).

I would also like to thank Mgr. Marek Vinkler and Mgr. Magdalena Čípová for correcting my imperfect English and general support.

I declare that I wrote this thesis by myself and with the exclusive use of the material quotted. I agree the thesis to be lent.

in Prague Bc. Jiří Horký

Contents

Title: Improving Efficiency of HEP Applications
Author: Jiří Horký
Department: Department of Software Engineering
Supervisor: RNDr. Filip Zavoral, Ph.D.
Supervisor's e-mail address: zavoral@ksi.mff.cuni.cz

Abstract: The Large Hadron Collider (LHC) located at CERN, Geneva has finally been put in production, generating unprecedented amount of data. These data are distributed across many computing centers all over the world that form the Worldwide LHC Computing Grid (WLCG). One of the main issues since the beginning of the WLCG project is an effective file access on the site level in order to fully exploit huge computing farms. The aim of this thesis is to explore existing data distribution work flows, standards, methods and protocols. An integral part of the work is the analysis of jobs of physicists to understand input/output workloads and to discover possible inefficiencies. Then, new upcoming solutions are evaluated in terms of performance, sustainability and integration into existing frameworks. It is expected that these solutions will be based on distributed file systems such as NFS 4.1, Lustre and HDFS.
Keywords: grid, profiling, distributed filesystem

Název práce: Zlepšování efektivity HEP aplikací
Autor: Jiří Horký
Katedra : Katedra softwarového inženýrství
Vedoucí práce: RNDr. Filip Zavoral, Ph.D.
E-mail vedoucího: zavoral@ksi.mff.cuni.cz

Abstrakt: Velký hadronový urychlovač (Large Hadron Collider - LHC) vybudovaný v CERNu v Ženevě byl konečně spuštěn a začal generovat obrovské množství dat. Tato data jsou distribuována mezi výpočetní centra po celém světě, která tvoří Worldwide LHC Computing Grid (WLCG). Jeden z největších problémů již od začátku tohoto projektu je efektivní přístup k datům v jednotlivých centrech tak, aby se plně využily obrovské výpočetní prostředky. Cílem práce je prozkoumat, jakým způsobem se distribuují data, jaké jsou použité standardy, metody a protokoly. Důležitou částí práce je dále analýza přístupů k diskům spuštěnými úlohami, aby se zjistily případné problémy a neefektivní chování. Součástí práce je také porovnání nových řešení ukládání dat založených na distribuovaných souborových systémech jako je NFS4.+, Lustre nebo HDSF.
Klíčová slova: grid, profilování, distribuované souborové systémy

Introduction

The Large Hadron Collider(LHC)[16] at CERN[1], situated both in Switzerland and France, is with its 27km of length currently the biggest particle accelerator in the world. There are four major experiments at the LHC: ALICE [2], ATLAS[3], CMS[7] and LHCb[17]. Although the LHC is not yet operating at full energy and luminosity, it already produces 15PB of raw data a year. In other words, it forms one thousandth of all data produced all over the world.

All these raw data from the detectors need to be archived, converted to a suitable format for a physical analysis (reconstructed) and finally processed by the analysis software itself. These data are not kept centrally at CERN but they are distributed to many computing sites that form the Worldwide LHC Computing Grid (WLCG)[33]. It is the largest grid infrastructure in operation today, comprising of more than 200 sites spread over 34 countries on 5 continents. Moreover, most sites also participate in other international as well as national grids. Because of different requirements of the experiments, computing power available for the experiments differs significantly. A typical computing site consists of thousand of CPU cores and several hundred of terabytes of disk space. In the Czech Republic there is also a member of WLCG, the Institute of Physics of the Academy of Science (FZU)[30].

The ultimate goal of the WLCG is to enable physicists to perform their analysis as fast as possible. In order to do so, the aim is to exploit computing power fully by minimizing IO waiting time of the jobs. Current CPU efficiency measured as a fraction of CPU time and wall time typically ranges from 50-70%.

The aim of the thesis is to explore current approaches and solutions in the field of data access. Using this knowledge and further detailed analysis, the goal is to find inefficiencies in how applications access their data and to evaluate and benchmark new solutions of storing and accessing

[1]European Organization for Nuclear Research, `www.cern.ch`

the data in a form of distributed file systems.

Naturally, there are several layers on which data are transfered and handled. The whole picture of the problem is given in chapter 1. The chapter describes the current WLCG data models, all its layers from data acquisition to the actual data analysis and highlights important differences in experiment implementations.

Chapter 2 describes a common structure of the jobs of all four experiments and recent improvements in data handling of the shared framework.

In chapter 3 several means of IO profiling are described. In chapter 4 a specialized IO profiling application is introduced that was developed to IO profile the jobs.

Chapter 5 investigates the access pattern of typical jobs which is a crucial task for understanding requirements on the underlying file systems and observing possible inefficiencies of data access.

Chapters 6 and 7 describe how to use the newly developed tools based on a trace and replay mechanism for benchmarking. Various storage systems that are foreseen to come are then benchmarked and evaluated in terms of stability, life-cycle management, backing up and support available.

Chapter 8 list related works and the last chapter 9 summarizes the results.

Chapter 1

WLCG Data Management

Whole book would have to be written to cover all the aspects of data management in WLCG and experiment specific data management services. It would involve describing hierarchy of computing sites, types of data used, services and protocols to transfer and access data, various catalogs to keep track of data, as well as associated metadata, logical grouping of the files to datasets etc., and also the security model as this is an essential part of the Grid.

To keep the extent of the chapter within reasonable boundaries, only the overview of the whole problem is given in this chapter, omitting the authentication part and putting emphasis on data access as this is the topic of the thesis.

1.1 The Tiers Model

To aid in handling the huge amount of data, the WLCG infrastructure has been divided into Tiers. At the highest level is the Tier-0 at CERN, where the experiments record their data and perform a first-pass reconstruction over it. These data are then distributed to Tier-1 centers which are responsible for its long-term curation and reprocessing to derived formats. The 11 Tier-1s (10 for ATLAS, 7 for ALICE, 7 for CMS and 7 for LHCb) are large computing centers in Europe, the USA, Canada and Taiwan, all connected to CERN through dedicated network links of at least 10Gbps. Most of the centers support more than one experiment. Their role slightly differs across experiments according to their computing model [1] [40] [38] [37], but they are usually responsible for long-time storing of the data on tapes. The difference is mainly in the type of jobs being run at the Tier-1s and also in whether they store a full copy of the

RAW data or just a portion of them. ATLAS, for example, uses Tier-1s to run reconstruction jobs that produce files more suitable for physical analysis.

Reconstructed data files are then distributed to about 100 Tier-2s, where they are being analyzed. Again, a typical Tier-2 center supports 2-3 LHC experiments. The Tier 2 facilities are where most of the users make contact with the data, that are housed on disks. Another important role of Tier-2 centers is a computer simulation of the detectors response as the experiments require for data analysis comparable amount of relevant simulated data. To complicate matters further, CERN also acts as a Tier-1 center and any Tier-1 center can also act as a Tier-2 site.

The exact description goes beyond the scope of the thesis. Please refer to the computing models mention above for further information.

The figure 1.1 shows the computing model of the ATLAS experiment with description of the type of jobs being run in each Tier.

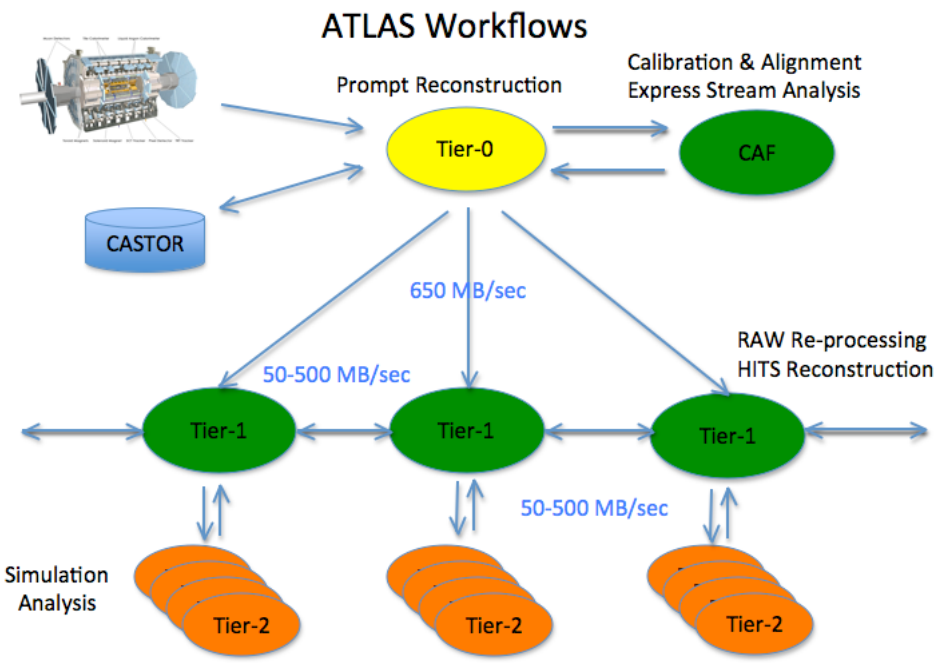


Figure 1.1: ATLAS tier structure, taken from [43]

1.2 Terms Definition

In order to understand following sections, some terminology must be given in advance.

- **Storage Element - SE** - A service that provides uniform access to storage resources within the Grid. Each site has at least one Storage Element that can shield many disk servers or a Mass Storage System (MSS). The SE acts as a storage door to the site.
- **Storage Resource Manager - SRM** - A protocol used to do storage resource management, but not any transfer. It is used to ask an MSS to make a file ready for transfer, or to create space in a disk cache to which a file can be uploaded. It provides authentication, space management, file pinning, SURL to TURL resolving (see below) and shields different implementations of Storage Elements, that provide SRM interface.
- **Grid Unique Identifiers - GUID** - Unique name of the file on the Grid. One file has only one GUID.
  Example: `guid:3a69a819-2023-4400-a2a1-f581ab942044`
- **Logical File Name - LFN** - A human readable name of the file. One file can have more LFNs.
  Example: `lfn:/grid/ATLAS/horky/Dataset.dat`
- **Physical File Name - PFN** - A path to the file in a Storage Element. Also called Storage URL. It points to actual server having the data. The path after the hostname represents only logical path on the SE.
  Example: `srm://gol100.farm.cz/atlas/VJ_F08.aq.root`
- **Storage URL - SURL** - The same as the PFN.
- **Transfer URL - TURL** - A valid URI with the necessary information to access a file in a SE. It has the following form:
  `<protocol>://<path>`, where protocol must be a valid protocol (supported by the SE) to access the contents of the file.
  Example: `rfio://tbed01.cern.ch//dteam/gen/file3e86f-c40`
- **File transfer service - FTS** - A service used to schedule and manage transfers of data between sites.

- **Logical file catalog - LFC** - A service which provides mappings between LFNs, GUIDs and Storage URLs SURLs and keeps track of file replicas in the system.

1.3 Data Distribution

As described above in section 1.1, the data are distributed from top (Tier-0) to the bottom (Tier-2s) via Tier-1s. Each site has one or more Storage Elements, that acts as a door to a site and implements well-defined interface, the SRM protocol. The data are transfered between sites using the File Transfer Service (FTS) that aims to reliably copy one Storage URL from one Storage Element to another. It uses a grid-security-enabled FTP client/server (GridFTP[12]) to achieve this and the FTS retries the transfer if it fails. It also schedules copies along pre-defined network channels to ensure that bandwidth is properly used. It should be mentioned that FTS is not the only tool used to transfer files.

Several other experiment-specific components above FTS are needed to interact with logical file catalogs (LFCs) of the experiments and to actually initialize the transfers. The Logical File Catalogue is a service which provides mappings between Logical File Names (LFN), Grid Unique IDentifiers (GUID) and Storage URLs (SURLs) and keeps track of file replicas in the system.

To transfer a file from one site to another, one has to obtain its LFN using experiment's metadata databases. Once having LFN, the SURL to one of its replicas is obtained using LFC (or an experiment's service built above it). The SURL point to a particular site's Storage Element. To actually transfer a file, the File Transfer Service must obtain the Transfer URL by communicating with the SE using SRM protocol. The TURL points to actual disk server within the site that stores the file. The file does not go through SE, but it is transfered directly from the disk server.

An overview showing how the different components described in this chapter connect is shown in the figure 1.2.

The same mechanism of resolving TURL is used in every file transfer within the Grid, not just using FTS. Users, for example, can download files for analysis to their desktops once authenticated.

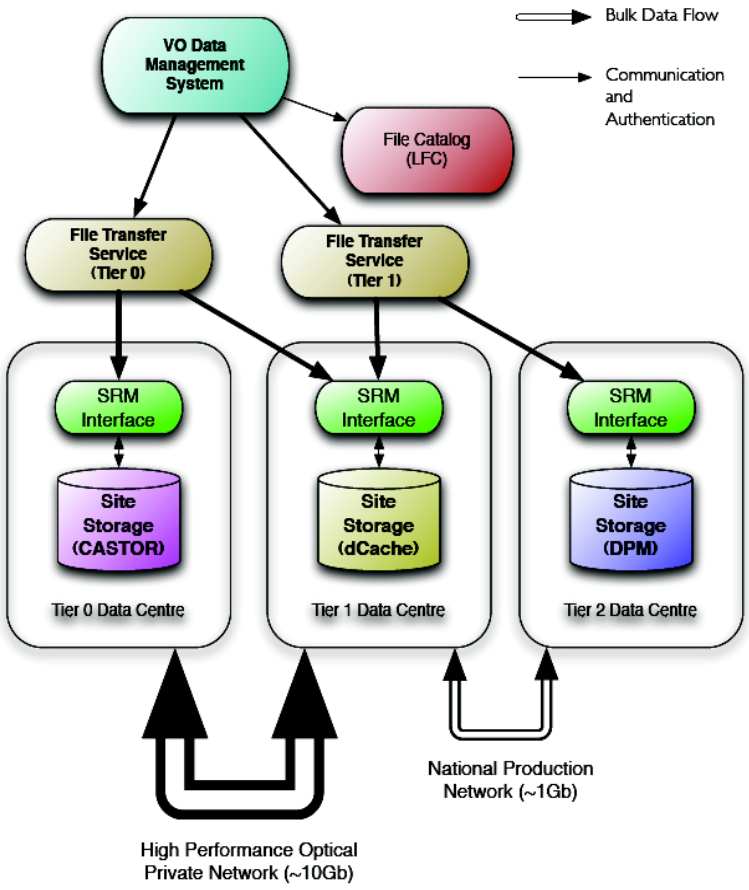


Figure 1.2: The overview of data distribution within the WLCG, taken from [49]

1.4 Storage Elements

Storage Elements play a crucial role in the data management. The main role of SE is to provide a single data entry point to a site from the outside perspective, thus hiding all details of actual data network in the site. A Storage Element provides an SRM interface that shields different implementation of the SEs that have emerged since the start of the WLCG project. The SRM interface can become a bottleneck[36], as multiple requests has to be made for one transfer. The achievable SRM request frequency lies around 100Hz [36]. One Storage Element usually consists of many servers and connected disk arrays. There is always a main server which is visible to the outside world and which manages site's names-

pace. We remark, that one of the task of the SE is to translate SURLs to TURLs which means locating the file on the disk nodes. The SE also authenticates users and checks their privileges.

The list of Storage Element's implementations in production with a short description of each is given. We provide a more detailed description of Disk pool manager (DPM) system as it is the most popular solution deployed in the WLCG in Tier-2s, see table 1.1[1].

- **DPM - Disk Pool Manager[9]** - DPM is a light-weight storage manager developed at CERN. In a typical installation, it consists of one so-called head-node, which provides the SRM interface and manages the namespace and of pool-nodes that store the data and can use any conventional file system such as ext3 or xfs. The DPM uses MySQL[22] database for namespace management and requests tracking, as the SRM protocol is asynchronous. An user contacts a DPM head-node with a request to download a file and with a list of his supported access protocols. The DPM consults the database and finds one or more replicas of the file, forming the TURLs according to the protocols supported by the DPM system that match the user's request. The user then uses the resulting TURL to contact the data server which has the file using appropriate protocol. The DPM does not support any hierarchal storage management, so it can not be used with tape libraries. It also does not support intelligent replication of hot files (this must be done manually). The DPM is the most popular SE implementation on Tier-2 sites mainly because of its ease of use.

- **dCache[8]** - dCache is a more complex Grid storage solution mainly deployed at Tier-1 or big Tier-2 centers. It was developed as a joint venture among DESY[2], FNALFermi National Accelerator Laboratory(`www.fnal.gov`) and NDGF[3]. As opposed to DPM, it supports mass storage systems and various tertiary storage managers. It also supports hot file replication and automatic load balancing. Besides other data access protocols, dCache provides its own native protocol (dCap, or its secured variant gsidCap) to support regular file access functionality. It also exports metadata information of all files

[1]The numbers were obtained by querying each SE listed in the Grid Information service, the BDII, on 14th July, 2010

[2]Deutsches Elektronen-Synchrotron(`www.desy.de`)

[3]Nordic Data Grid Facility(`www.ndgf.org`)

stored in the system via pnfs[4](pretty normal file system), so users actually see the files in their file system.

- **CASTOR - CERN Advanced STORage manager[5]** - CASTOR is the most heavy-weight, yet the most powerful storage solution in production in WLCG. Even though there are just few sites using CASTOR, it manages the biggest portion of data in the Grid. Note that each CASTOR instance provides more SRM entrypoints, so the number of its instances (see table 1.1) is higher. According to its homepage, the CASTOR manages more than 181 million files representing more than 28PB of data as of Aug 3, 2010.

- **StoRM - Storage Resource Manager[25]** - StoRM is another SE implementation developed at INFN[5]. Its main goal is to provide SRM interface to POSIX file systems that supports Access Control Lists (ACLs), such as Lustre[19] or GPFS[10] and thus enable these technologies to the Grid. It does not manage the namespace, as it completely relies on the underlying file system.

- **BeStMan - Berkley Storage Manager[4]** - BeStMan is similar to StoRM in its aims, as it provides SRM interface above distributed POSIX file systems as well. BestMan has been developed by LBNL[6].

SE implementation	**# sites**
DPM	187
dCache	80
StoRM	41
CASTOR	18
BeSTMan	15
Total	445

Table 1.1: Number of SRM doors of different types of Storage Elements in production

The performance evaluation of different SE implementation has been published [36].

[4]not to be mistaken with pNFS(Parralel Network file system), a part of NFS4.1

[5]The National Institute of Nuclear Physics , collaboration of one Tier-1 center and several Tier-2 centers in Italy (`http://www.infn.it/indexen.php`)

[6]Lawrence Berkeley National Laboratory, California (`http://www.lbl.gov/`)

1.5 Data Access Protocols

Data access protocols are used by jobs running on site's worker nodes to read the data available on the site's SE. Typically, this is done using specific, non-standards protocols but the task is similar to accessing the data using a site-global distributed file system such as Lustre or GPFS.

When talking about the data access protocols, the term "protocol zoo" becomes spelled quite often. There is a large variety of protocols, making support and certification process of a new software a difficult task.

- **RFIO** - Remote File IO protocol has been developed at CERN and it has been in production since 1990. It implements almost all of IO POSIX calls (read, write, seek etc.). There are two versions of RFIO protocol, secured (supporting GSI[7] authentication) and unsecured one. RFIO is still widely used for example for the ATLAS analysis.
- **dCap** - DCache Access Protocol is a native protocol implemented within dCache. Similarly to RFIO, there is also a security-enhanced version of the protocol. dCap supports POSIX I/O abstraction using `LD_PRELOAD` mechanism - an user can then use pre-installed commands like `cp, ls` or `mv` to work with files.
- **GridFTP** - GridFTP protocol is mainly used for data transfers between SEs, but it can also be used for data access. In that case, files are first transfered to a local worker node to the scratch area and then accessed locally. This is one of the most commonly used access methods.
- **File protocol** - When using distributed file-systems, or other mechanism providing file-system abstraction, files can be accessed using POSIX IO calls as if they were stored locally.
- **xRootd** - is data access as well as data distribution protocol. The ALICE experiment uses this protocol exclusively to access the data. It is described in more details in 1.6.1.

1.6 ALICE Data Model

The ALICE experiment quite differs from other experiments from the data management perspective. The ALICE experiment does not use the

[7] Grid Security Infrastructure

concept of Storage Elements (and thus it does not use the SRM protocol) but it uses only one data access protocol exclusively. The xRootd procotol is used for both the data distribution and data access. That is why it is more correct to speak about xRootd as a data access system. The main idea of the protocol completely differs from other protocols mentioned above. A detailed description is given in the next section.

1.6.1 xRootd

The xRootd protocol, often referred to as the Scalla suite was developed at SLAC[8] in 2003 and then has been continuously improving until now. The main aspects of the system are:

- **No file database** - xRootd does not keep a record of where the files are, a file is discovered when it's requested
- **No SRM interface** - the system does not use the SRM interface as only one protocol is used for both, data transfers and data access
- **Scalability** - the system is capable of managing thousands of storage servers
- **Fault tolerant** - if the transfer from one source server fails, it could still be recovered using other server that has the file
- **Load balancing** - when multiple clients access the same file, the system can spread the requests among multiple storage servers
- **Multistreaming** - A great effort has been put into efficient WAN[9] transfers with high round-trip times. One file can be transfered using multiple streams.
- **Simple configuration** - configuration of xrootd servers consists of setting a single file with few lines only
- **Modular architecture** - the whole system is highly modular

The figure 1.3 denotes the architecture of the xrootd system. All the servers form B-64 tree with a special server called redirector as a root. Clients that want to open a file need to know only the address of the redirector server. The redirector asks all underlying servers if they store

[8]Stanford Linear Accelerator Center, a research center in San Francisco, USA
[9]Wide area network

the file. If the asked server is a supervisor, it forwards the request deeper in the tree. When the redirector collects all locations of the file, it chooses the best server according to the load (if the load balancing module is enabled) and it answers the client. It also caches the locations of the file (the supervisors cache it as well).

In case of a server failure, the client contacts the redirector node with a request for another location of the file being accessed. The redirector can then invalidate its cache by issuing another request to its underlying servers. This data server outage is invisible to the application itself (besides the slow down of the operation).

The redirector server may look as a single point of failure, but it is not, as it can be configured as a cluster.

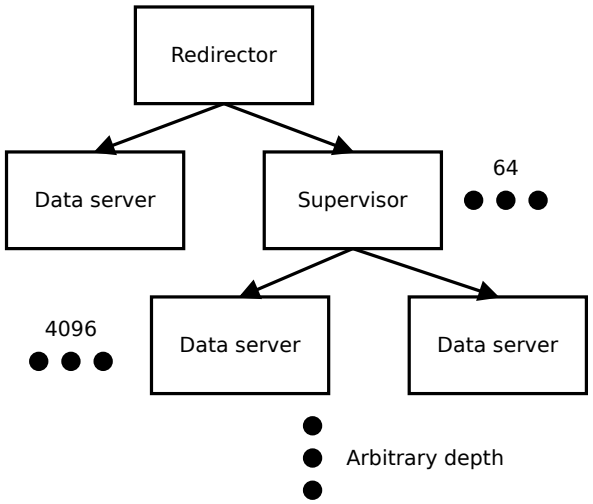


Figure 1.3: xRootd system architecture

As an example of the deployment of the xrootd system, access to the ALICE condition database (a database that every analysis job within the grid must access) can be given. These data are distributed only on five servers located centrally at CERN and accessed exclusively using xrootd protocol.

For further information about the xrootd system, please refer to xrootd homepage[31] and materials linked therein.

1.7 Distributed File Systems usage

As one of the objectives of the thesis is to evaluate new storage solutions, namely distributed file systems, an overview of current usage is given.

There are quite a lot of advantages connected with the usage of standard distributed file systems such as Lustre or GPFS. First it has a large user base, and thus a lot of problems are already understood, there is better support, documentation etc. Secondly, from the performance point of view, these file systems usually benefit from using page cache of underlying Linux kernel, i.e. almost all unused memory is used to cache data. This is a great advantage when reading the same data twice (e.g. backward seeking) as well as when kernel predicts data that will be read next.

Currently, there are two Storage Elements suitable for the use with e a distributed file system, the StoRM and the BeStMan. Their primary role is to provide an SRM interface and authentication to the Grid.

Chapter 7 describes and evaluates the usage of GPFS, Lustre, Hadoop and NFS4.1.

1.8 NFS v4.1

NFS 4.1 [23] is a recently standardized protocol for accessing data on a distributed system. It is described in details in chapter 7.6, but we provide a brief introduction here too.

The minor version 1 of the NFS protocol brings supports for sessions (and thus minimizing traffic needed for individual requests) but most importantly it introduces the pNFS[10] protocol, the protocol to access files distributed on many disk servers. The pNFS cluster consists of a name server, data servers storing actual data and clients. We refer to the name server and data servers as the server part of a pNFS system.

Even though the in-kernel implementation of the protocol is not yet considered stable, the protocol itself is of great interest not only in the high energy physics (HEP) community.

The current model for its usage is, however, not through the in-kernel server implementation (SPNFSD), but rather through implementing the server side in user space as a part of most commonly used Storage Elements, the DPM and dCache. The in-kernel implementation of clients is then used on clients to benefit from the operating system's page caching.

[10] parallel NFS

See [47] and [45] for more information about deploying NFS4.1 in DPM and dCache.

1.9 Conclusion

The brief introduction of the data management in WLCG was given, ranging from the data distribution and data transfers to data access.

Even though there are number of different components and technologies used, there is one unifying factor: almost complete lack of any standardized technology.

It is understood that having many access protocols is sub-optimal, but it is not yet clear which of the protocols will survive and which will not. Nowadays, it seems a lot of work is being invested into NFS4.1 [23] (see chapter 7.6).

The author's vision is that the POSIX-like file abstraction is the solution to be chosen. The idea would be to provide a file system, possibly using FUSE[11] that would hide the underlying access protocols. All of the access protocols more or less support this but the files are still accessed using specialized calls of the given protocol.

[11] File system in user-space, `http://fuse.sourceforge.net/`

Chapter 2

Jobs' Structure

All the four main LHC experiments use the same data analysis framework, ROOT [39]. Usage of the same access layer in the form of ROOT means that every experiment can benefit from improvements in ROOT data handling. We describe the structure of the jobs, how data are organized in files (and thus how they are read back) as well as recent improvements in ROOT data access that very significantly improved the way data are stored and read back.

Starting from 2.3, we supplement the description with our own performance measurements and visualization of the changes in the data format.

2.1 Common Structure

A common structure of an analysis application is shown in the figure 2.1. The top part of a job consists of an experiment specific code. For analysis itself, it uses ROOT, that in turn for data access uses its class `TFile`. This class has then different subclassed providing many different plugins to access inputs files. Among others, `TXNetFile` (for xRootd protocol), `TFile` (POSIX file system), `THDFSFile` (Hadoop), `TRFIOFile` (rfio protocol) and `TGFALFile` (possibility to use LFN, GUID or SURL, see 1.2 for more information) should be mentioned. At the time of writing, there were twenty different plugins for data access.

Usage of the same access layer in the form of ROOT means that every experiment can benefit from improvements in ROOT data handling. The important change in ROOT happened in December 2009, when the way data are stored changed significantly, which is described in following sections.

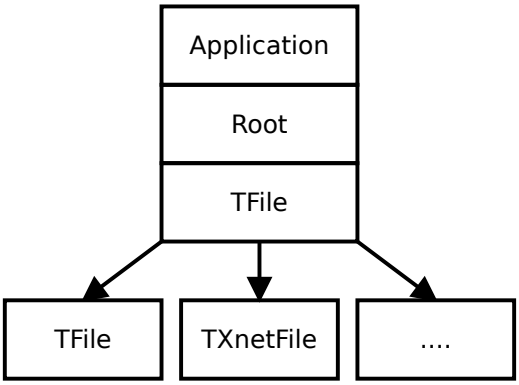


Figure 2.1: Basic structure of the applications

2.2 ROOT Data Files

The ROOT data files are self-describing, they include both data and their description. Although there are several ways how to store the data, most ROOT files use so-called Trees, generic containers that can contain arbitrary data. One Tree (class TTree) usually stores instances of one C++ class. Trees are optimized for storing multiple (million) of instances of the class by saving data description (a header) only once and then having just a pointer to all instances, saving space by omitting multiple class definitions.

The tree consists of branches, each branch usually represents one variable of the class (it can be a simple type variable such as `integer`, an array, a structure or an object), but it can also represent more than one variable, depending on the choice of a designer. Each branch has a branch buffer (basket) that is used to collect the values corresponding to the variable represented by the branch. Once the buffer is full, it is written to the file. All variables of the same branch of all instances are physically stored together on a disk in the baskets. This allows to quickly read a variable (or more coupled variables) of all instances without extensive seeking in the file. Whether the whole branch is stored in one place in the file, or is spread across the file, depends on the size of a basket buffer.

As an example, lets assume that class Event describes one collision of particles and it consist of one hundred variables, `float x` and `y` among them. If one wants to histogram function of `x` and `y`, within all Events saved in a file, he only needs to access 2/100 of the file. If, however, the instances were saved one by one in a file, it would need to seek through whole file, which would be inefficient.

The data files could be (and most of them are) compressed using gzip compression.

2.3 Improvements in ROOT File Handling

In December 2009, new production release v5.26/00 of ROOT was released. It has introduced many important improvements in IO handling that are described below. In fact, these changes were so dramatic that it almost invalidated any benchmark performed on previous production version v5.22/00.

The most imporant change is, however, missing. The new ROOT version has not yet been adopted by all the LHC experiments.

2.3.1 Buffer Size Optimization

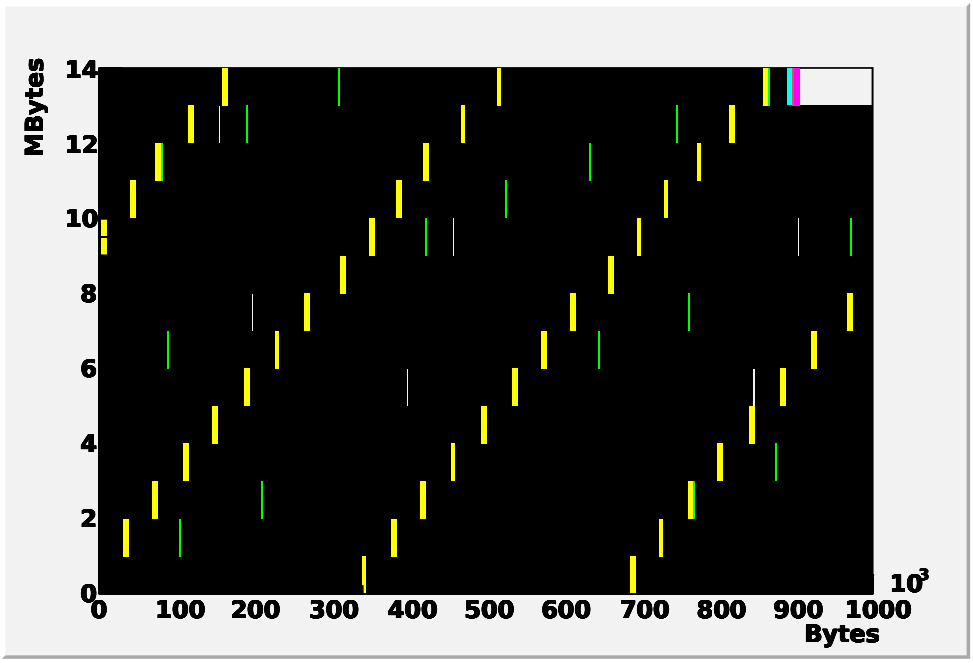


Figure 2.2: Distribution of branches in an unoptimized file

As described above, a buffer size of a given branch influences when data are actually written to the disk. The figure 2.2 illustrates this fact

by showing a map of physical positions of branches in a test file. Two (yellow and green colors) out of twenty branches were colored in the sample 14MB ROOT file containing one Tree representing Event class. Black color represents data of other branches, white color represents free space and blue color represent headers. The file contains 400 instances of the class. As can be seen, the branches are spread all over the file, the yellow branch distributed to 40 baskets, the green one into 25 baskets. The basket size was set only to 16kB for each branch.

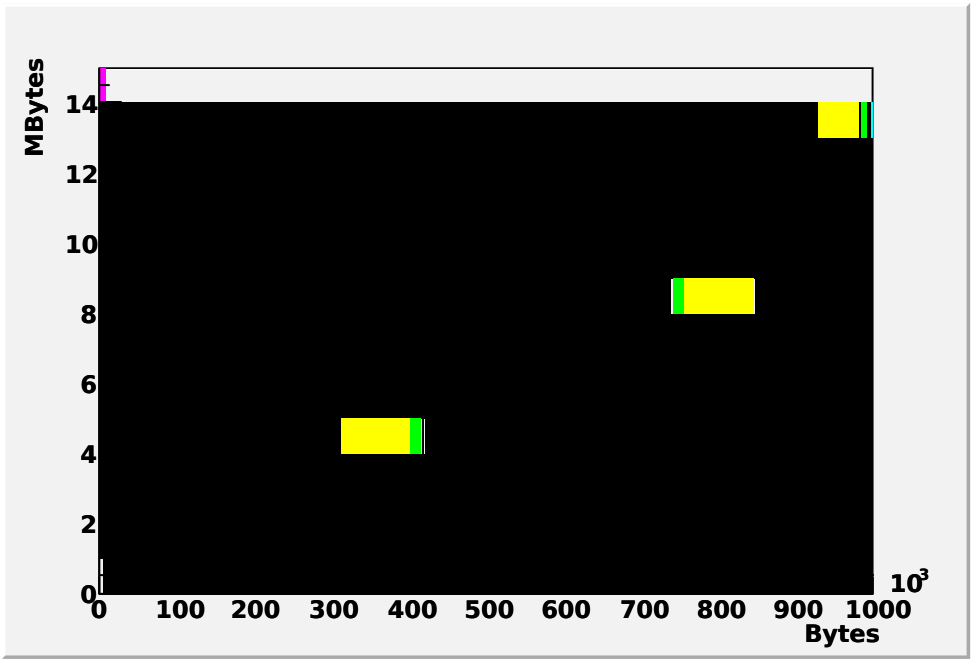
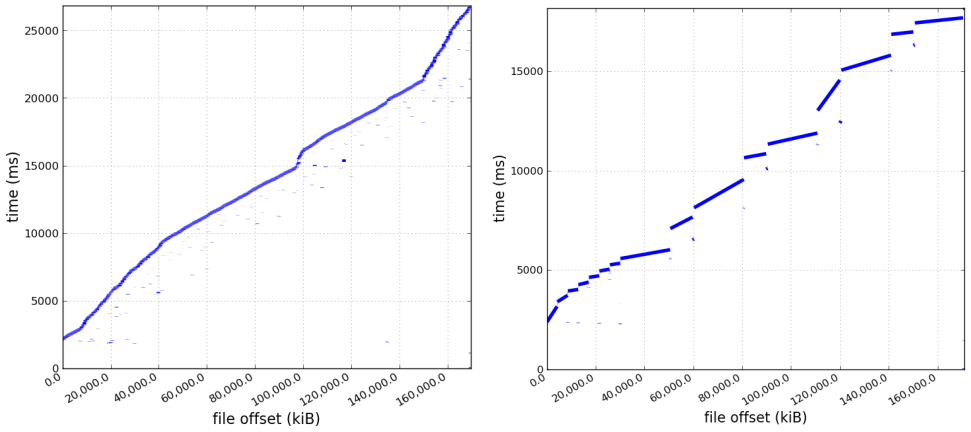


Figure 2.3: Distribution of branches in an optimized file

Since ROOT v5.26/00, basket size for each branch are optimized taking into account the population in each branch, all branches are also flushed to the file together, when one of them reaches its size limit. As an immediate effect, this highly reduces the number of seeks required to read a branch. This is illustrated in the figure 2.3 showing the file from the previous example rewritten with the optimization enabled (maximum size of a branch was set to 10MB[1]). Only three baskets of both branches are now used, with sizes of 234kB for the yellow basket and 145kB for

[1]This buffer is probably too large for real life scenarios, where trees contain several thousand of branches



(a) Unoptimized baskets (b) Optimized baskets

Figure 2.4: Influence of basket size optimization during read

the green one. The reason why there is not only one, or maximally two baskets (the file has only 14MB) is that the file is compressed and the biggest branch in the file holds 30MB of uncompressed data. So for that particular branch, at least three baskets had to be used, which triggered saving of other branches' baskets as well. Because the branches are always flushed together, data of one instance of a class are always saved physically close together.
The interval (in megabytes) at which the baskets were flushed on the disk is also saved in the header to help reading the file back efficiently (see 2.3.2).

The effect is once again shown in the figure 2.4. The plots show the read pattern when reading all data from a file containing 5000 of the same events as in previous example. For the unoptimized file, 5967 reads were performed, whereas only 159 reads were needed for the optimized one. The number of read requests almost exactly reflects the number of baskets in the file. The huge difference is also in average speed of reading the file, the optimized file structure brought 60% improvement.

Please also refer to the section 5.3.1 to see the influence of this change for a real life ATLAS analysis job.

2.3.2 TreeCache

TreeCache has been in ROOT for several years, but it has been hard to use until recently when a new interface was introduced. Moreover, new releases of ROOT include better caching by taking advantage of basket

flushing as described in section 2.3.1. TreeCache provides a mechanism to read-ahead branches before they are actually needed. It groups all blocks from the used branches into one buffer. The blocks are sorted in ascending order and consecutive blocks merged such that the file is read sequentially.

The important fact is, that this cache really operates on the branch level and thus can behave more efficiently in files where branches are scattered across the file than a conventional read-ahead cache could. The cache size is always a multiple of a frequency at which baskets were flushed to the file. The TreeCache also merges individual, possibly non-consequent, read requests into bigger requests. That means, if there is a small gap between two reads, only one read is performed reading also the gap.

2.3.3 Sub-optimal Implementation of TreeCache

Even though the TreeCache brought additional improvement, we identified some sub-optimal behavior using the strace output and statistic functionality of the `IOreplay` application (described in 6.2). The table 2.1 summarizes the number of read and seek system calls performed when reading an unoptimized 180MB real life ATLAS data with TreeCache first enabled and then disabled. The input file was cached in the page cache of the operating system. The data read from the file were silently discarded (they were not processed).

	reads count	time(ms)	**seeks** count	time(ms)
cache on	5629	1399.5	614461	8843.6
cache off	206601	4778.4	206849	3356.9
cache fixed	5529	780.3	9879	345.0

Table 2.1: Unnecessary seeks with TreeCache

As can be seen, the number of read operations is approximately 36x lower, while number of seeks increased by the factor of three when using the TreeCache. We discovered the root of the problem by inspecting the strace output by hand; there were a lot of consequent seek requests that were not followed by read or write system calls. The problem was reported

to ROOT developers and was fixed in three days [48]. As of writing of this thesis, the fix has not been included in any stable release. The numbers obtained using the fixed version are also presented.

Chapter 3

Profiling

Understanding of application's behavior is essential for being able to improve its performance. In particular, when speaking of a file access, one is typically interested in a disk access pattern of the application, i.e. whether it reads data sequentially or randomly, how big are typical IO requests, how many of these requests the application does and how many files does it open per run. Another very important question to answer is then: how the application buffers data?

We evaluate possible approaches for profiling of the IO behavior of an application in this chapter.

3.1 Profiling Methods

The idea behind profiling is to catch all IO calls and inspect their count, offset within a file and duration. There are many options how to do it, but not all of them are suitable for the purpose of profiling HEP experiments' applications, given the restrictions listed in the next section.

3.1.1 Requirements

A typical application of an experiment requires very specific and complex settings of environment and relies on various services. It also needs input data files of the experiment, which are not available for non-members of the collaboration. Given this fact, the author had to ask other people to profile the applications on their production machines, as he has access to neither the experiments' applications nor their data (with the exception of the ATLAS experiment). For that reason, requirements of no OS settings changes and easiness of use were one of the most important factors

when deciding the IO profile solution.

The list of requirements on the profiling is provided:

- Being able to reliably trace behavior of HEP applications
- Usage of a distribution kernel
- No changes to the OS settings (i.e. root access not needed)
- No source code modification needed
- Ease of use

Although the first point might seem trivial, it is not. The important fact here is that all the profiled HEP applications are possible to be configured to access input files using file system, instead of network protocols. Also, none of the applications uses `mmap` system call, which is important for profiling on the system call layer.

3.1.2 Profiling Methods

There are many possible solutions of how to track the IO calls, differing on the layer they operate at (completely in kernel, user-space or combinations of the two), the complexity of its usage, ability to track hidden IO calls (caused by page faults of memory mapped files) and they also vary in a performance overhead they introduce. The comparison of possible profiling methods is given in the table 3.1 and individual methods are decribed below.

	Kernel changes	**SW installation**	**Root privileges**	**mmpap IOs**	**Overhead**	**Dev. complexity**
Kernel hacks	Yes	Yes	Yes	Yes	Very Low	Very High
SystemTap	No	Yes	Yes	Yes	Low	Low
Lib. preloading	No	Yes	No	No	Low	Medium
Strace	No	No	No	No	High	None

Table 3.1: Comparison of the profiling methods

SystemTap[28] is a relatively new and actively developed project for tracing both kernel and user-space functions and calls. It provides its

own scripting language for defining actions to perform when given event occurs. The event could be a kernelfunction call (which is traced using kprobes), system call as well as a user-space function. Current project members include Red Hat, IBM, Intel, Hitachi, and Oracle. Tracing by SystemTap is always system-wide, although it is possible to filter events by the pid of the process which caused it.

Because of the requirement of no OS setting changes, it was not possible to use SystemTap, as it needs installation of kernel debug symbols and root privileges to run. If there were not such requirements, it would probably be the best tool to use because of its flexibility and low performance impact. With this software, it is also possible to trace memory mapped IO calls, that are not visible on the system call layer.

Another possible approach would be to use library preloading through `LD_PRELOAD` environment variable and thus instruct dynamic linker to preload our own implementation of IO functions, enabling us to reimplement every such call. We could then inspect every parameter of the call and then call the original function. However, when using `LD_PRELOAD`, it is impossible to track statically linked applications (which is not a problem for tracing of the selected HEP applications). It also does not work for setuid programs when the user running the program is not the same as the owner of the executable. This is because being able to load code into a privileged executable would be a security flaw. Depending on the Unix version used, there might be also problems with programs that call `dlopen()` explicitly.

As examples of the `LD_PRELOAD` approach, the PlasticFS [24] and IOProf[53] can be given.

Another solution is to exploit `ptrace(2)` system call that interrupts execution of a given program whenever a system call is performed. The solution does not suffer from previously described problems `LD_PRELOAD` has. However, usage of this call is rather complex and depends on both machine architecture and Unix version. Fortunately, there is already a well-known standard tool using `ptrace(2)` mechanism, `strace` [27]. Besides having the expected problem with memory-mapped IO calls, it can have a significant performance overhead of more than 100% of running time depending on the number of calls traced. See table 6.1 for detailed numbers for real-life ATLAS analysis application. It was measured that approximately 20 thousands of syscalls per seconds cause 1% of overhead on Intel Xeon E5520 2.27GHz.

Taking into account the requirements and drawbacks of the various methods, the author decided to use the `strace` method mainly because of its wide availability and ease of use when obtaining profiling information. The overhead problem that affects exact timing of individual IO calls is not considered as a big issue, as we are mainly interested in the access pattern of the applications, which should not be affected. All the physical applications can be configured to use POSIX-like method (using TFile class, see chapter 2) for accessing data to enable us spotting all IO system calls.

3.2 Strace

As stated above, `strace` is a standard tool for tracking system calls caused by a given application. It is possible to trace application from the beginning by executing it from the `strace` itself as well as to attach to already running program. The output of the `strace` is user-friendly; it prints out names of the parameters instead of the actual integer constant (`O_RDONLY` for example). An example of its output is given in listings 3.1.

```
1765 1279445178.319030 open("/usr/bin/root", O_RDONLY) = 21 <0.000088>
1765 1279445178.319261 read(21, ""..., 512) = 512 <0.000081>
1765 1279445178.320078 close(21) = 0 <0.000078>
1765 1279445178.320284 writev(1, [...], 1) = 2 <0.000083>
1764 1279445178.320429 <... read resumed> ""..., 128) = 2 <0.005448>
1765 1279445178.320462 exit_group(0) = ?
1764 1279445178.320512 read(21, "", 128) = 0 <0.000032>
1764 1279445178.320668 --- SIGCHLD (Child exited) @ 0 (0) ---
```

Listing 3.1: An example of strace output

3.2.1 A Strace Bug

Even though the `strace` has been available on virtually every Linux in production since decades, it has its issues. When tracing complex programs that involve many processes with complicated process tree, one probably hits a strace bug: the output gets corrupted, some system calls are not captured and the parameters of others might be output in unusual formats. Because of time constraints, the author didn't try to fix the issue but he at least reported it to the developers [50]. Unfortunately, nobody answered at the time of writing.

3.3 Conclusion

Several methods of IO tracing were evaluated in the chapter, listing their advantages as well as drawbacks. Given the aforementioned requirements, the `strace` was chosen as a source of the traces for IO profiling.

Chapter 4

IOprofiler

Because of the shortage of options for IO analyzing of a `strace` output, a new tool called IOprofiler has been developed. Description of its architecture and features follows.

4.1 Introduction

The only script that can be found concerning `strace` outputs and IO analysis is `strace_analyzer` [26]. Unfortunately this tool does not handle problems described in 4.2, which makes it of no use when tracing complex programs that spawn children processes and that are using calls like `dup` or `socket`. It also does not include any easy way of plotting the data.
For that reasons, our own application called IOprofiler has been developed. The goal was to provide fast and user friendly way of analysing IO from a strace output. The idea was to provide a GUI application from which users could generate access pattern diagrams (individual read-/writes plotted with file offset and call time as axes) of individual files, with some statistics in a few clicks. Because the strace output files produced by long running jobs that process tens of gigabytes of data could be very big (up to few gigabytes), there was also a need for an option to convert the text file into a more suitable architecture-independent binary format, that would be smaller and faster to read, as the text would not have to be parsed. All these requirements were fulfilled.
Because of a low level nature of IOreplay (see 6.2) that shares part of the code and logic with the IOprofiler and also because of the requirement of fast processing, pure C was chosen as the programming language.

The choice of the framework for the GUI part was mainly influenced

by availability of the plotting frameworks. After evaluation of many solutions, the Matplotlib library [21] has been chosen. The GUI itself is therefore written in Python using PyQt framework. Because the analysis application and IOreplay, an application for replaying syscalls described in chapter 6 both need to parse an input file and handle file descriptors to file names mapping (see 4.2), it was an obvious choice to have a common layer for it. All file parsing and file descriptor *fd* mapping machinery is done through a compiled python module based on the same C source files that both applications use.

4.2 File Descriptors Handling

The basic idea of using `strace` output is to simply go through all `read`, `write` and `lseek` system calls and plot them with file offset and time as the axes to see whether a file is accessed sequentially or not. It turns out that it is quite more complicated.

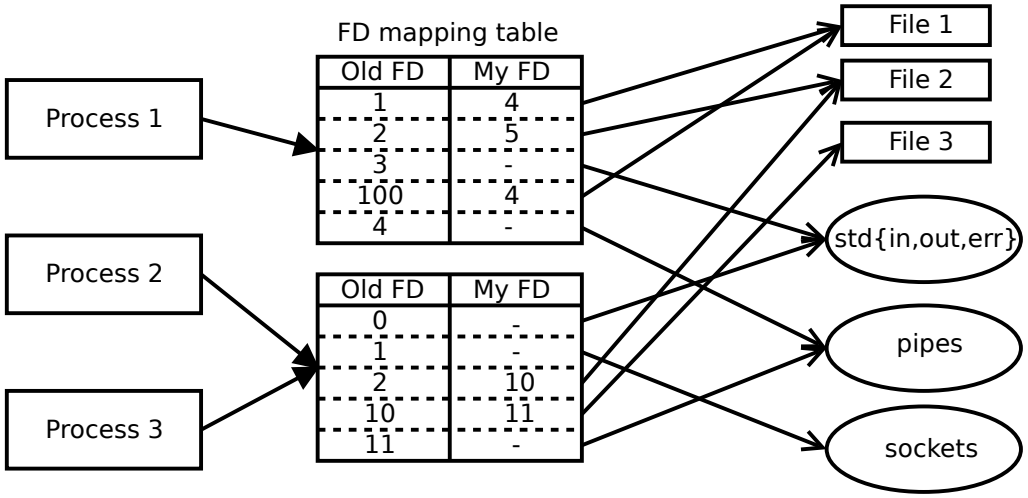


Figure 4.1: Handling of file descriptor mappings

The issue is that every such system call operates with a file descriptor instead of a file name, so one has to keep track of the mapping of the file descriptor to the file name it is referring to, which is surprisingly not a trivial task. The typical job involves many processes, where a *fd* mapping is unique only inside each process. Moreover, not every file descriptor corresponds with a regular file, it could also be a pipe or a socket, in which

we are not interested. Reads and writes to such *fds* are then ignored, as we are only interested in requests that hit a hard drive. When a new process starts, it can clone the file descriptor table of the parent process, but it also can just share it. That means that every operation performed by the child process on a *fd* also affects the corresponding parent's *fd*. Sharing of file descriptors is possible not only on a file descriptors table layer, but also individual file descriptors can be shared via `dup` system call. As the result, 18 different system calls including `clone` or `socket` must be traced and then processed.

When analyzing a recorded strace file, the application simulates the run of the original application by taking into account all system calls that could alter the file descriptors-to-names mapping. For example, the `open` call creates a mapping, while `dup` creates a new file descriptor to the same file and `close` deletes the mapping. The same file descriptor (its number) can belong to different files on the disks during the execution of a program, as the numbers are being reused. The figure 4.1 shows how recorded (old) file descriptors are mapped to file names when reconstructing the run of traced programs. The column "My FD" is only used when replaying the straces files by IOreplay, see chapter 6 for more information. When simulating, only unique numbers with no relation to the actual file system are used.

4.2.1 GUI

To enable a user-friendly way of analyzing traced data, a simple GUI written in PyQt framework have been developed. It consist of two screens. The first one (figure 4.2) is showing a list of files read or written by a traced job with basic statistics for each file (number of reads, summary of read requests, summary of duration of requests). The list is sortable and the shown files can be filtered by using regular expression filter in the bottom of the screen.
The detailed information could be displayed for each file. This triggers a detail dialog (figure 4.3) with the list of all operations on the left side and the plot on the right side. There are three types of plots. The default one, the pattern diagram, shows individual requests performed with a file offset as x axis and time elapsed from opening the file as y axis. The other plots are histograms of requests' size, duration and speed.
Taking advantage of using Matplotlib, the program enable users to zoom the plots and highlight individual request by either clicking in the pattern diagram, or selecting the requests from the list. The plots could be

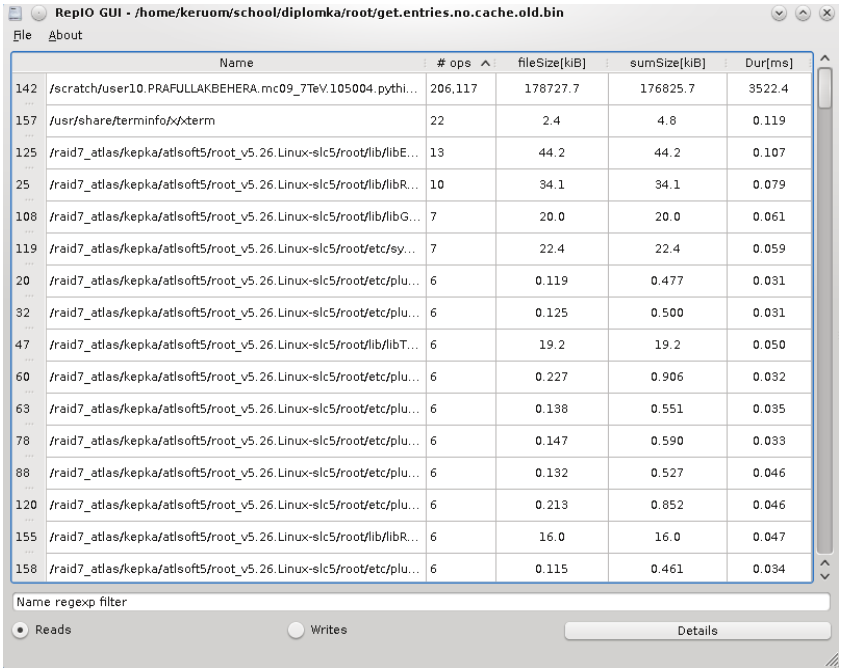


Figure 4.2: IOprofiler - The main window

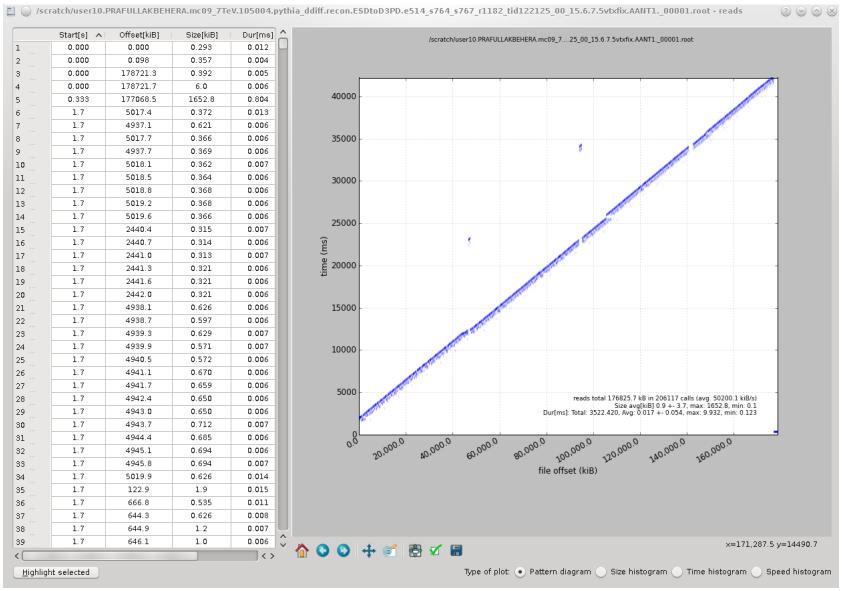


Figure 4.3: IOprofiler - The detail window

exported to both vector and scalar graphic formats.

Although rather simple, the GUI makes really fast and efficient analysis of a job's access pattern possible.

4.3 Conclusion

IOprofiler, a tool for IO profiling based on `strace` was introduced in this chapter. It enables fast and easy-to-use analysis of IO behavior of applications. The tool is freely available at [42].

Chapter 5

Experiments' Access Pattern

In-depth understanding of the access pattern of the jobs is essential for explaining their behavior on different storage solutions. This is particularly useful when benchmarking using these jobs.

Once we had a tool for IO profiling, the IOprofiler (see chapter 4), we were able to take a closer look at what the experiments' jobs actually do in terms of storage accesses.

This chapter provides IO analysis of representative samples of LHCb, CMS and ATLAS jobs. For the CSM and LHCb experiments, two types of jobs were profiled, an analysis and a reconstruction job. These represents two basic types of workloads. The reconstruction jobs are used to reprocess raw data format to a format more suitable for user analysis and thus they read whole data files mostly in a sequential manner. The analysis jobs, on the other hand, typically picks only parts of the files that are important from a user's point of view.

For CMS and LHCb experiments, the author asked members of the collaborations to provide a strace output of their jobs. The jobs for ATLAS were actually run by the author.

Due to the different data model used by the ALICE experiment (explained in chapter 1.6), no ALICE jobs were traced.

5.1 LHCb

The strace outputs for analysis and reconstruction jobs of LHCb collaboration were provided by Vincenzo Vagnoni and Stefano Perazzini (INFN). Both types of jobs run on 64-bit version of Scientific Linux CERN v5 and ROOT v5.26.00b. By looking at the access pattern of the analysis job (figure 5.2), it is evident that the source files have been written by the

non-optimized version of ROOT. DaVinci, the LHCb analysis software, version used was v25r4. The version of Brunel, the LHCb reconstruction software, used was v37r2.

5.1.1 Reconstruction

The reconstruction job read 788MB raw file. At summary, 495 unique files were opened (only 34 of them local, the rest was mostly experiment software accessed remotely). The whole job lasted 4 hours and 3 minutes, whereas the raw file was opened for one minute less. As shown in figure 5.1, the raw data file was read strictly sequentially, with 52101 read calls in. 1.9s in total was spent by reading the file. This indicates that the file was cached. The average read request size was 15.5Kb, with standard deviation of 16.9kB and its maximum of 81Kb. From the list of calls (not presented here), one could see that every bigger read request (15kB+) is followed by one small request of 0.2kB. The same statistics for request duration are 37μs$\pm$ 607μs, maximum 97634μs. Interestingly, the highest number of calls were performed to the /proc/PID/stat and /proc/PID/statm system files. These were opened, read and closed more than 26 thousand times, total of 3.3 seconds was spent in these calls.

Because of the sequential nature of reading the raw file and a very small fraction of time spent in IO calls, this type of job is not that suitable for optimizing.

5.1.2 Analysis

The analysis job lasted for ten and a half minutes and opened 501 files in which it is very similar to the reconstruction job. The input ROOT file had 875MB and contained 26050 events. It was read with 16739 read calls, with average size of 49.1kB $\pm$ 754kB, maximum request size was 19388kB (there were tens of such big requests, see figure 5.2a). The source file was opened for 8 minutes and 55 seconds, which means that initialization of the job took more than one minute. Once again, the /proc/PID/stat and /proc/PID/statm files were accessed extensively, 82 times per second, resulting in 5 seconds 'wasted' for memory monitoring.

There were three other interesting files in terms of IO. The 60.5MB big config.tar (41540 of 8kB reads, so the file was read five times!), and two SQLite databases DDDB.db and LHCBCOND.db, 50 and 80 megabytes big, respectively. Around 10MB was read from each database with strictly 1kB big reads.

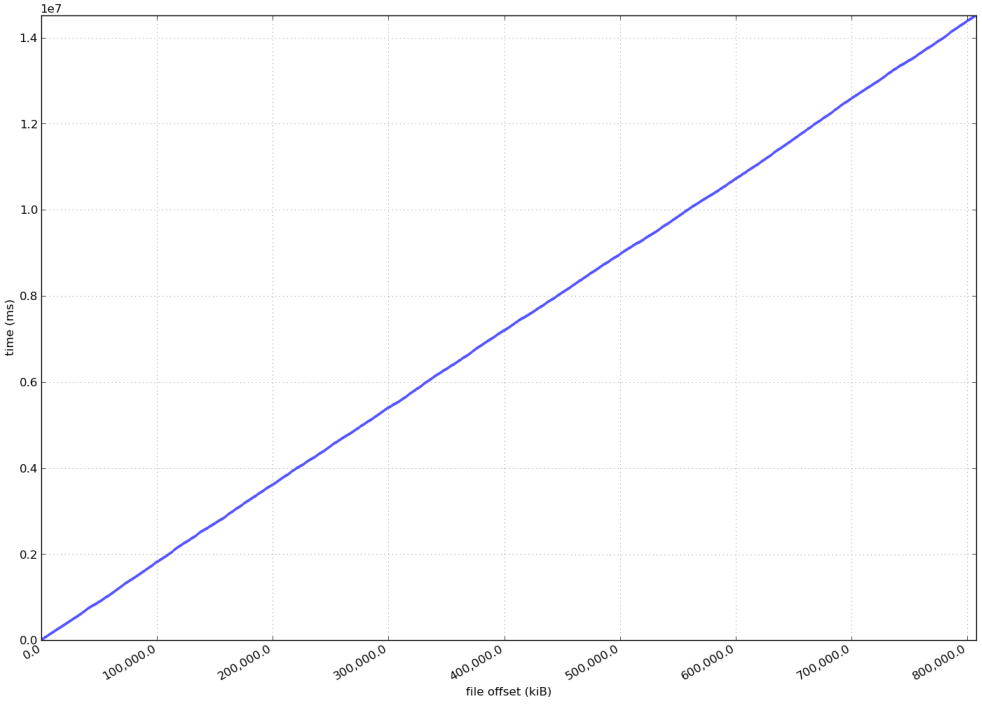


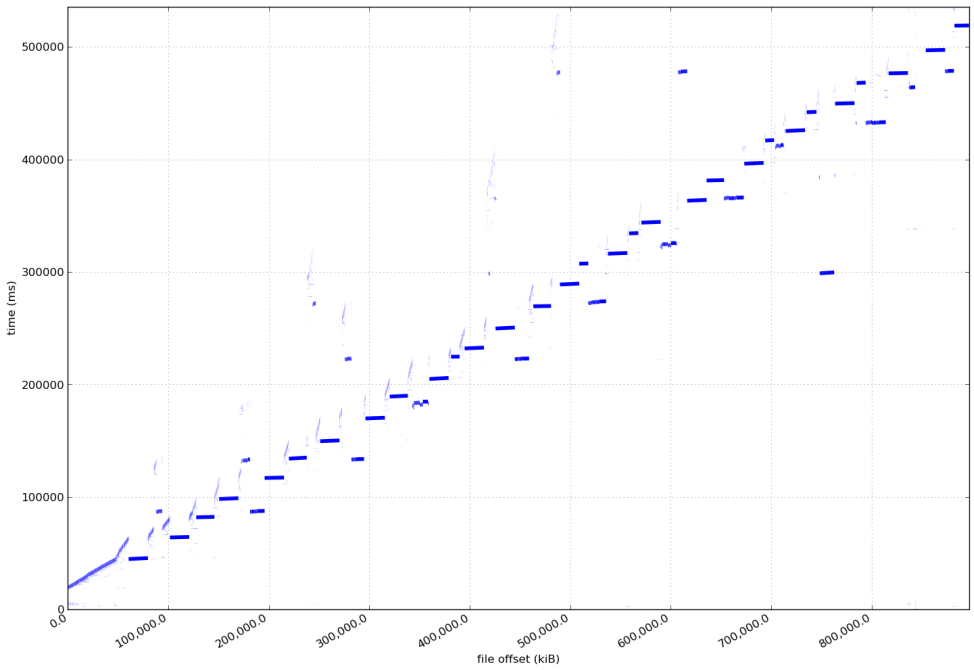
Figure 5.1: Read pattern of an LHCb reconstruction job

5.2 CMS

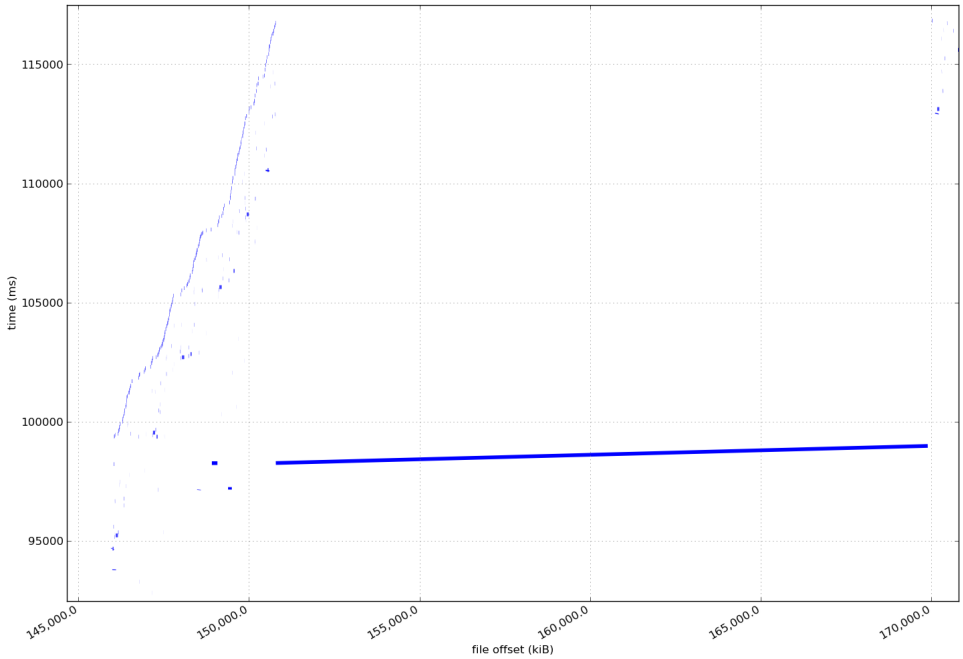
Profiling data for CMS analysis and reconstruction jobs were provided by Leonardo Sala (CERN) with help of Brian Bockelman (Nebraska University). More versions of the CMS software were tested, but only the latest production version is presented here. The CMS experiment uses ROOT v5.22.00d with their own patches for IO subsystem, to cite their author: "I patched some of the worst sins of 5.22, but there's still huge, gigantic changes in the I/O layer in 5.26 to make it even better'. Used version of CMSSW was 3.7.0. The ROOT's TreeCache was enabled with a size of 20 megabytes. All the information were caught using 32-bit Scientific Linux 5.

5.2.1 Reconstruction

The traced reconstruction job opened 2905 files in total, only 31 out of them were not libraries or executables of the executed software. The job was processing 4642MB big RAW source file, but only first part



(a) Whole picture



(b) Detail

Figure 5.2: Pattern diagram of an LHCb analysis job

representing 1000 events was read. In numbers, 424MB was read by 5077 calls, average read size 85.7kB with standard deviation of 157.6kB and maximum read request size was 4893.4kB. The file was opened for 19 minutes whereas the job lasted for half a minute longer. The read pattern of the job is displayed in figure 5.3, which shows great effect of caching in a form of almost horizontal lines. Although not visible in the figure, the job constantly accessed few kBs scattered in around 10MB block in the end of the file. In the view of rather sequential access pattern and small fraction of time spent to read it, the CMS reconstruction job could be again considered as not that suitable for optimization. Because of the use of the TreeCache (and its bug, see 2.3.3), 91.3% out of 196 thousand seeks were unnecessary.

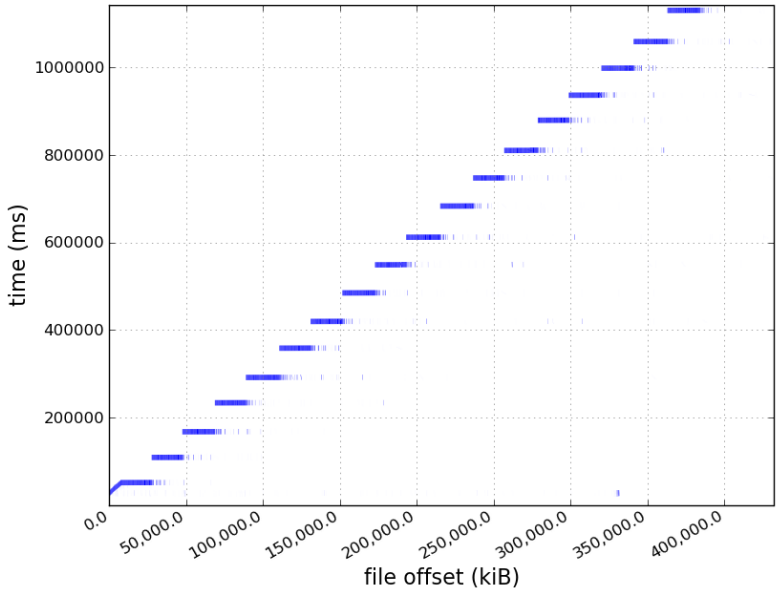
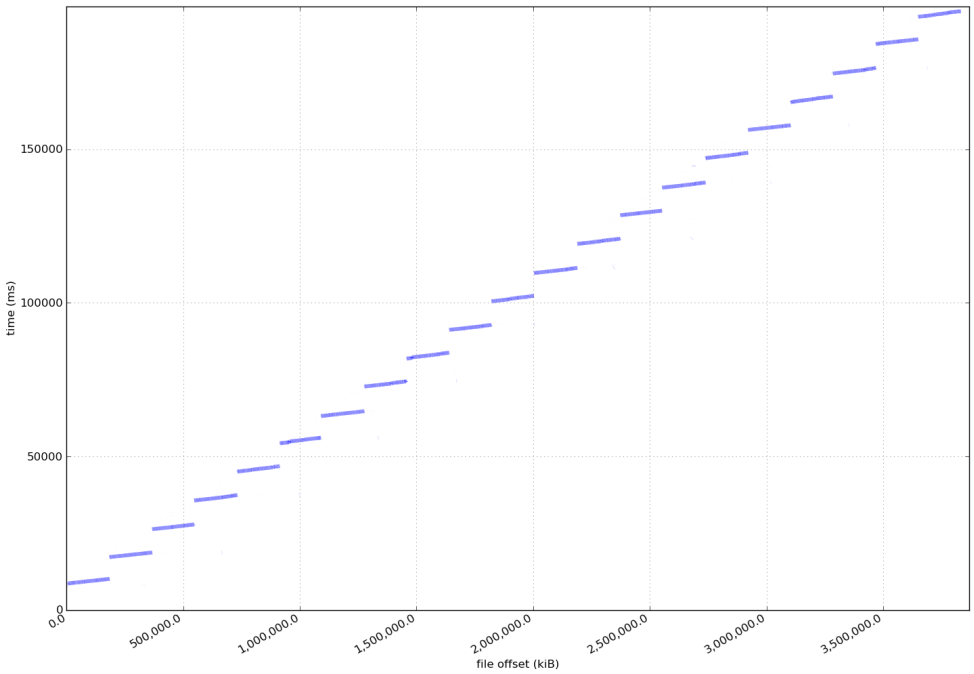


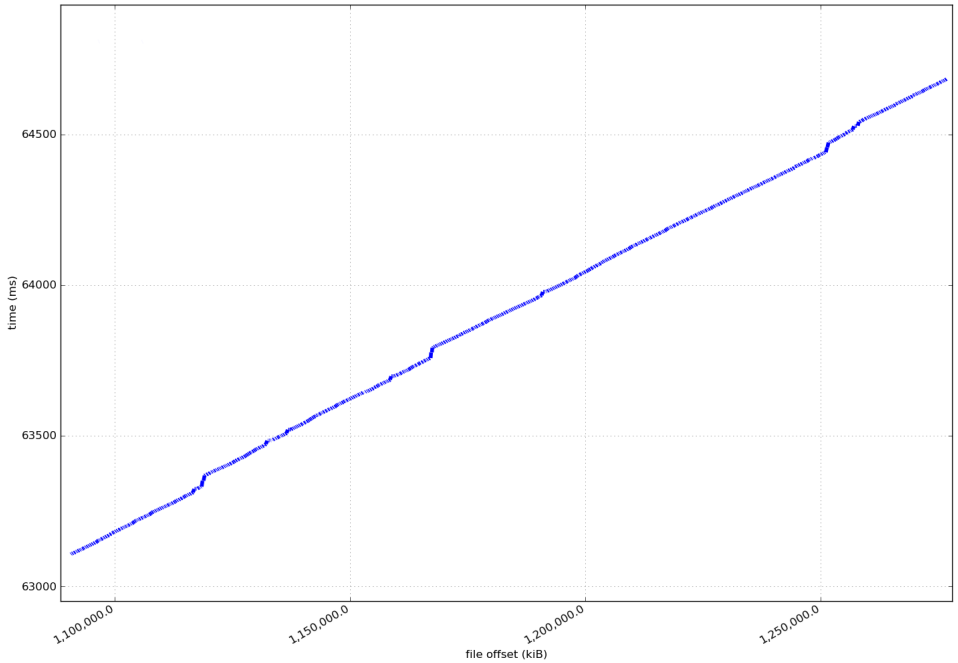
Figure 5.3: Read pattern of an CMS reconstruction job

5.2.2 Analysis

The analysis job used 532 files (501 of them located were libraries and other files needed by the used software area). 1984.1 megabytes were read out of the source ROOT file (4624MB) that contained 10000 events using 12655 operations (average read size 157.1 ± 88.0, max. 4893.4kB). Although the pattern might look similar to reconstruction one at the



(a) Whole picture



(b) Detail

Figure 5.4: Pattern diagram of an CMS analysis job

first glance, it is not. Zooming in the plot (figure 5.4b), one can see that the single reads are not consequent, there is always a small gap between them. That's why the figure 5.4a shows reads in offset up to 3800MB, whereas only half of it was actually read. The source file was opened for 197 seconds and the whole job lasted for 203 seconds. Because of the use of the TreeCache (and its bug, see 2.3.3), 88.5% out of 112 thousand seeks were unnecessary.

5.3 ATLAS

The ATLAS experiment is the only experiment of which data the author has an access to as it is the main supported LHC experiment in the Institute of Physics AS CR (FZU), the current workplace of the author. As described in 1.1, mainly user analysis and no reconstruction jobs are being run in Tier-2 centers. Taking advantage of having access to the actual code of the analysis jobs, it was possible to measure the effect of various ROOT data format improvements. Moreover, the ATLAS jobs served as an example for replaying comparison study in the chapter 6.

The author also tried to obtain system call traces from ATLAS jobs submitted through WLCG to FZU's computing farm. The client part of a batch system (Torque[32]) was patched to insert call to `strace` before executing the actual script that was submitted on the farm. As the execution of jobs involves calling of various wrappers inserted on different levels in different languages and thus resulting in a quite complicated chain of processes, the strace bug described in 3.2.1 has been hit. If that occurred, the outputs were unusable. Only manually executed jobs were successfully traced.

The analysis application was provided by Oldrich Kepka. The jobs were run under 64-bit Scientific Linux v5.4, using ROOT 5.26.00 on a machine with Intel Xeon E5520 @ 2.27GHz with 300GB 15k RPM SAS hard drive.

Unfortunately, no reconstruction job for ATLAS experiment was traced. It is believed that it would not differ much from the reconstruction jobs of CMS and LHCb.

	# reads	time(ms)	kBytes read	avg request size(kB)
Old, cache off	191209	25895	247203	1.3 ± 16.8
Old, cache on	3205	7927	541572	169 ± 148
New, cache off	14658	5338	212428	14.5 ± 50.4
New, cache on	797	4348	236759	297.1 ± 562

Table 5.1: Comparison of TTreeCache and basket reorganization during ATLAS analysis

5.3.1 Analysis

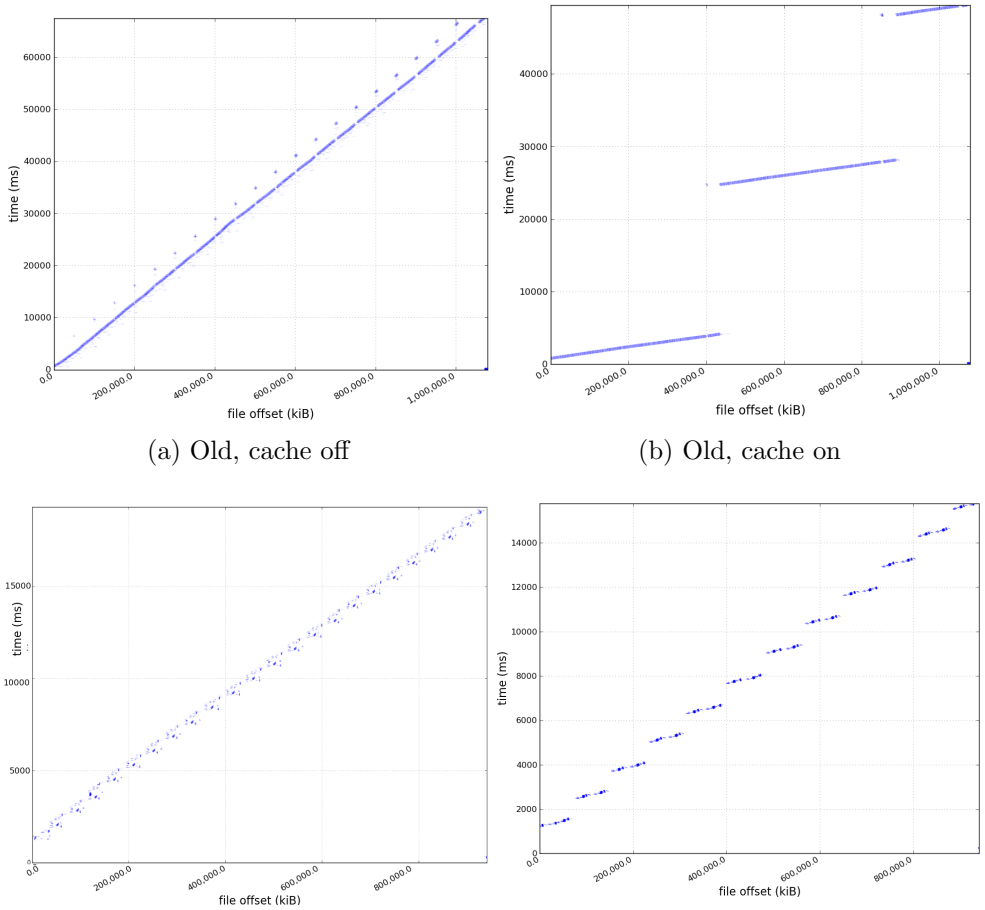
Because of the possibility to run an ATLAS analysis job with different settings of TreeCache and also with input files written with different ROOT version (with basket size optimization), all four combinations are presented here. The IO access patterns differ significantly. The default one used by the user was an older, unoptimized file structure with TreeCache enabled.

The table 5.1 shows number of read request, its average size with standard deviation and also the time spent reading alone. The "new" rows show numbers obtained by running the analysis job over a new format of ROOT file rewritten using ROOT v5.27. The "old" one shows numbers for input files created in ROOT v5.22. The TreeCache size was set to 100MB.

The interesting part of the table is the number of bytes really read from the file. The use of TreeCache brings also some read-extra overhead, that could bring performance problems when accessing data over network. The bytes read without using the cache is the minimum number of bytes needed by the application without any overhead. These differ for "new" and "old" version, as the files written in "new" version are slightly smaller. The read-extra overhead for the "old, cache on" was more than 115%, but it was still much faster then reading half of the data without the cache. The overhead for the "new" version of the input file was only 15%.

Also the access pattern of the four combinations profiled differs a lot. The figure 5.5 shows pattern diagrams for each case. Note that when using the cache, all reads are sequential, as the TreeCache reorders the read requests.

The figure 5.6 shows one part of the whole access pattern of the jobs in detail to point out the huge difference. Although the access pattern of



(a) Old, cache off

(b) Old, cache on

(c) New, cache off

(d) New, cache on

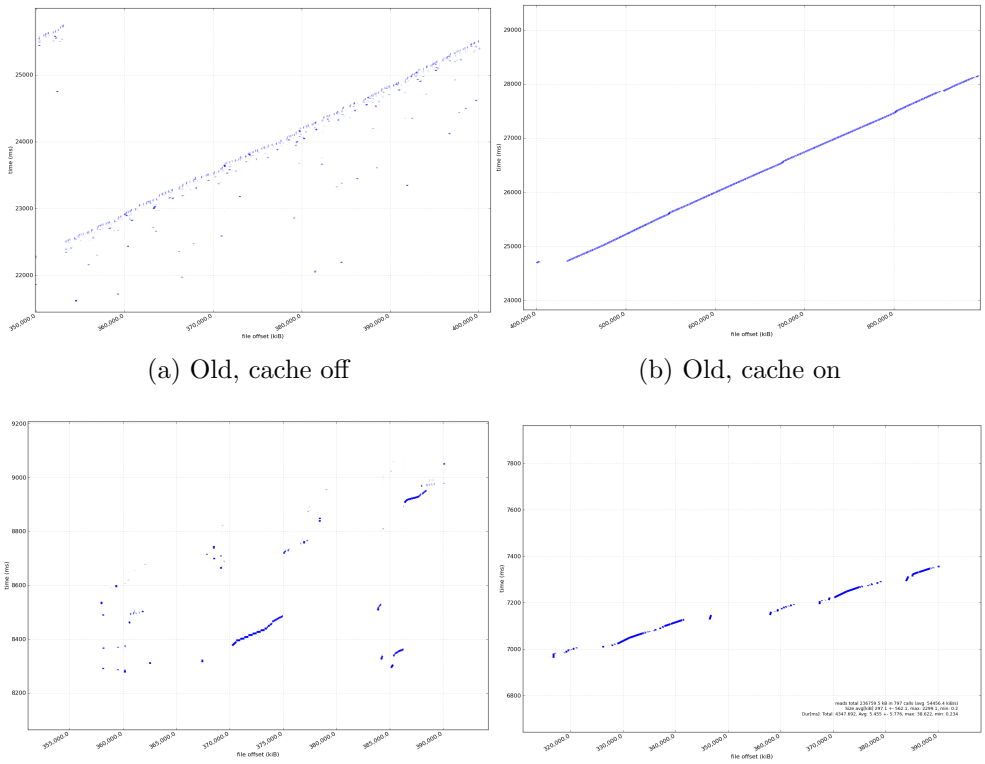
Figure 5.5: Pattern diagram of the ATLAS analysis job with different settings

the "Old, cache" case (figure 5.6b) might seem reasonable, one can see in a closer look that the line actually consists thousands of small reads, which is sub-optimal.

5.4 Conclusion

Different types of the typical LHC experiments' jobs were profiled and described in the chapter. The access pattern significantly differs across the experiments, type of a job (reconstruction/analysis) and the ROOT version used when reading as well writing the data files.

It was shown that the changes in the ROOT data format thoroughly



(a) Old, cache off

(b) Old, cache on

(c) New, cache off

(d) New, cache on

Figure 5.6: Details of access pattern of the ATLAS job with different settings

described in the chapter 2.2 have an important influence on both the access pattern of the jobs and the overall speed of data reading.

The knowledge of IO behavior of the jobs was exploited when benchmarking distributed file systems in chapter 7.

Chapter 6

IOreplay

In order to fulfill one of the goals of the thesis, to evaluate distributed file systems for use in high energy physics community, the right methodology and tools have to be chosen for benchmarking the evaluated systems.

There are three approaches one might adopt, based on the trade-off between complexity of a benchmark setup and fidelity of it. a) Run the real application on the test system and measure some application metrics; b) Collect traces from a running application and replay them back on to the storage systems to be evaluated; or c) Use synthetic benchmarks to generate workloads and measure the IO system performance for different parameters of the synthetic workload. The first method is ideal as it measures the performance of the system at the point that is the most interesting: one where the system is actually going to be used. However, it is also the most difficult to set up because of the cost and complexity involved in setting up real applications. Moreover, this approach is not always possible. The other two approaches, replaying traces of the application and using synthetic workloads, though less efficient, are commonly used because of the benefits of lower complexity, lower setup costs, predictable behavior, and better flexibility. Trace replay is particularly attractive as it eliminates the need of understanding the application in detail. The main criticism of these approaches is the validity of the abstraction, in the case of synthetic workloads, and the validity of the trace in a modified system, in the case of trace replay.

As already described in chapter 3, execution of an analysis or a reconstruction job of an experiment requires complex setup of software and access to the data files. For example, for ATLAS, one needs the right versions of gcc, python, ROOT, Athena (ATLAS analysis software) and

several other depending libraries.

Because the author had access only to data and working environment of the ATLAS, the author decided to simulate the behavior of other experiments using the trace and replay mechanism.

An application called IOreplay has been developed for this purpose and is described in detail in this chapter. It was shown that it can be used for a very accurate benchmarking in regard to the original application.

6.1 Replaying

The idea is to replay every IO operation that was previously recorded by the `strace` program. Doing so, the results obtained should be very similar to running the actual job as all requests that hit the hard drive are performed in the exactly same order with the same size. The main difference is in the timing of the calls as `strace` itself has quite a big overhead. As described in [35], special care must be taken to overcome this issue.

It is important not only to replay the read and write requests, but also the metadata operations like `stat` and `access` system calls that are also very common and can represent a non-trivial part of the IO operations especially for distributed file systems with a remote metadata server.

6.2 IOreplay

An application that is able to replay every IO-related system call named IOreplay has been written in the C programming language. It shares a part of its source code with the analysis application as described in section 4, mainly the routines for parsing of a strace output and handling of the file descriptors to file names mapping. Only system calls that hit the hard drive are replayed. That means that calls like `socket` or `dup` are ignored with the exception of tracking of the file descriptor involved in such call.

It was decided to make the application single-threaded mainly because of the complexity of debugging multi-threaded programs and because of the fact that the physical applications that were straced are strictly single-threaded even though they involve more than one process. Naturally, this approach has its drawbacks that are described in section 6.2.1.

The following sections describe the whole process of replaying in more details.

6.2.1 Drawbacks

One of the main drawbacks of replaying the traces is the inability to reliably simulate multi-threaded applications that access a hard drive simultaneously. The system calls are replayed in a strict order in which they were recorded. In multi-threaded programs or when tracing more than one process, some system calls can be interrupted by another call from other process/thread. The other call(s) can finish before the first one is resumed. The result is two interleaving calls that can even rely on each other. When replaying, the interrupted call is performed after the call that caused the interruption. That is because not all parameters of the call are written in the strace output, when it is interrupted. For example (see 6.1), write call in process 11109 was interrupted by a read call from process 11108 which was again interrupted by the resuming write call from 11109. Therefore, the order of operations can be broken. Even worse, the result of the interrupted call could depend on the call(s) performed during its interruption. It is worth to mention that none such situation was observed by the author during using the tool so far.

```
11109 1279734664.748838 write(1, ""..., 117 <unfinished ...>
11108 1279734664.748887 read(25, <unfinished ...>
11109 1279734664.748901 <... write resumed> ) = 117 <0.000055>
```

Listing 6.1: Strace output with interrupted system calls

6.2.2 Preparing the Environment

In order to reliably replay all operations and to obtain the same results as in the original run of an application, it is important that the environment in which we replay is the same as it was when we recorded the run. In other words, every `open`, `unlink`. `stat` and any other system calls that originally failed should now fail as well, while the calls that succeeded should succeed.

For that purporse, two mechanism were developed. The first one provides a means how to define files that should be ignored during replaying and also enables mapping between original files and files in a new environment. The second mechanism efficiently checks for problems that would occur during replaying and suggests changes.

The first one consists of two lists. First list, the ignore list, is for the files on which operations should be ignored. The other one is for the mapping of original file names to new file names. Every operation that should be performed on the files in ignore list is not performed and every operation that involves a file in the mapping list is actually made on the substituted file. The motivation is that it is very common that the file structure of the original system is completely different from the system where the replaying is being run. In applications that open thousand of files that are almost not accessed, the ability to ignore such requests instead of creating the files could be a big advantage. The program can be instructed to list names of all files accessed to help creating these lists.

The second measure taken is more sophisticated and also takes the two lists described above into account. The program enables the user to run the replaying in a simulation mode to discover whether the environment is ready or not. It uses a module called "simfs" for that purpose - own in-memory directory structure implemented in a form of prefix tree. It goes through the list of the system calls and consults every syscall that contains a file name as a parameter with the prefix tree. If the tree contains it, depending on the call and the original return value, it either succeeds or triggers an error (e.g. file is in the simfs but the previous open call failed). If it is not in the prefix tree, it tries to consult underlying file system for the file or directory. In case of different return value, it outputs its recommendation (e.g. please delete file X, as the previous open call failed but it would not now). In every case, the result of the operation as recorded in the source file is reflected to the simfs, so other calls depending on the failure or success of this one will not be affected. This way, only the cause of the problem is reported. It also measures how big the files should be in order to make sure that read operations succeed.

It is important to stress that the module is not perfect as it only checks whether the file exists and its size, not taking into account its permissions at all. It could happen that the traced program running under non-root account had tried to delete `/etc/shadow` for which it did not have permission and thus failed. The simfs, however, always assumes that the previous call failed because the file had not been there, so if the replayed application is run under the root user, it suggests deleting the file `/etc/shadow` so that it will fail too. This would be fatal for the system.

On the other hand, the original program could have been run under

root and had really deleted the file. The simfs would not argue about it as it would remove it as well if run with appropriate privileges. It is therefore always wise to list all files that are being access to ensure that it does not harm the system.

6.2.3 Timing Modes

As stated in [35], it is crucial to keep the delivery of the calls in exact timing as in the original application. Skew of more than few hundred microseconds causes significant difference in IO planning. Three different modes of time keeping are compared in the paper: using `usleep()` and `select()` calls, and spinning with periodic CPU counter reading. The most accurate by the order of magnitude is using the CPU spinning. However, it is important to note that using any of the mechanism, it is impossible to hold exact timing on a modern non-real-time operating systems as it will be always influenced by OS interrupts and process scheduling.

Another important point is that calling the `gettimeofday()` to find out whether to wait or not is very expensive. It can cause overhead of tens or hundred of percents for traces with high rates of system calls per second.

Because of the above mentioned reasons, the spinning with CPU instruction counter (representing the count of CPU cycles done since reset) approach was chosen. Using of CPU time stamp counter has several trade-offs that one should be aware of: it shrinks program's portability as not every processor supports such a feature (basically every x86 processor newer than Intel Pentium Pro does). It is also not guaranteed that the counter is the same on all cores in a multi-core or multi-CPU environment. That is why we bound our application to one core using `sched_setaffinity` system call. It can also be sensitive to power saving modes on some CPUs, when the clock rate is decreased. On some processors the register is incremented in the actual clock rate frequency, while on another it is incremented by the maximum resolved frequency at which the processor is booted regardless on current clock rate. The user must ensure that the system he uses is configured to not change the CPU frequency on such processors. Instructions reordering can also have marginal impact on the exact instruction counting.

Reading the CPU time stamp register is a very fast way of getting current time when the frequency of processor is known. The program determines the frequency by calling `sleep()` for one second and measuring the actual time and cycle count passed.

File version	ROOT version	Cache on	time(s)	time strace(s)	overhead
5.22/00	5.26/00	yes	234.6 ± 2.2	563.2 ± 9.9	140.1%
5.22/00	5.27/05	yes	220.2 ± 0.5	224.0 ± 2.4	1.7%
5.27/05	5.27/05	yes	159.3 ± 1.7	160.5 ± 0.4	0.7%

Table 6.1: Strace overhead

The application provides three modes in which it can replay the calls:

- **AFAP mode** - The calls are performed as fast as possible, ignoring any delays caused by processing the read data in the original run.
- **Exact mode** - The calls are made as close as possible to the time they were previously made (measuring from the start of the application). Spinning in the for loop is performed until it is time to make the call.
- **Diff mode** - The calls are made with the smallest difference between them in respect of the previous run as possible. This mode usually gives the most reasonable results as it holds all think-times of the original application and the only difference is in the duration of the system calls. Moreover, the application allows to specify a multiplier by which the gaps should be shrink/widen. This is handy for simulating difference in CPU speeds, when the ratio between the reference system and the tested one is known.

6.2.4 Scaling

In order to mark benchmarking results obtained by replaying as valid, it must be shown that it behaves similarly to the original job even on different hardware and under different load of the system. In other words, that it scales well.

Table 6.1 shows the execution times of running ATLAS analysis job that processes nine one-gigabyte files with different version of ROOT and different version of the file format and with and without the `strace`. All numbers were obtained by running the job five times. We remark that ROOT version 5.22/00 is the old one without basket size optimization (see 2.3.1) and that ROOT version 5.26/00 suffers from the bug with useless seeking (see 2.3.3), which is fixed in v5.27/05. The huge overhead

measured in the first row of the table is caused by the enormous number of seeks performed - there were more than 5 million of them for 30 thousand reads, while for the later two, only tens of thousand were done.

The job with 5.27/05 file version was used to confirm the scaling of the replaying. Every job processed nine files of 1GB. Each job has its own copy of the same data files to avoid influencing the page cache of the operation system. Moreover, the page cache was flushed after each run using `echo 1 > /proc/sys/vm/drop_caches` command to not skew further runs. The machine used for the comparison study was equipped with two Intel Xeon E5520 CPU (8 cores total), 16GB RAM and one 300GB SAS drive of 15k RPM on which the input data were stored. The `diff` timing mode was used in order to simulate processing of the data by the application.

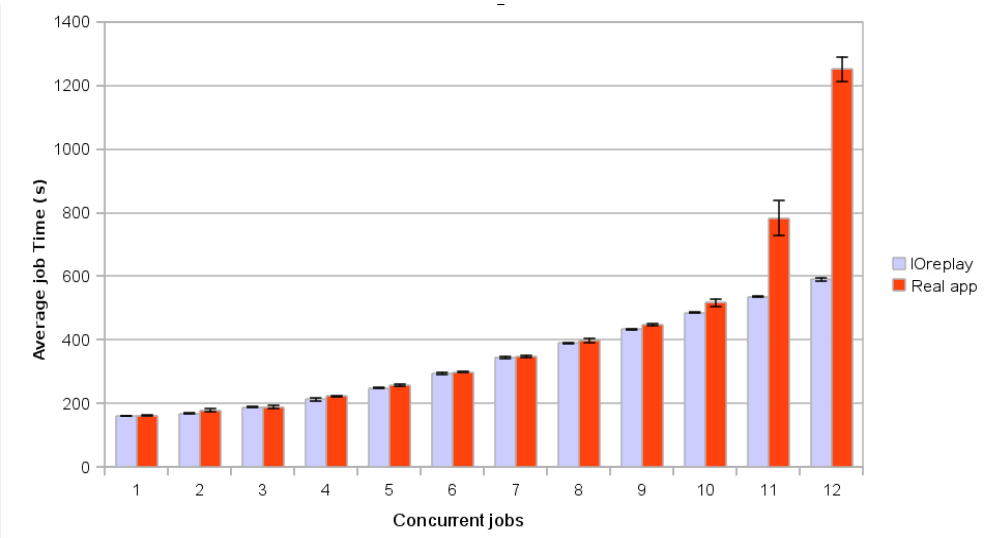


Figure 6.1: Scaling of the IOreplay application

The results are shown in the figure 6.1. The figure shows the sum of time (with standard deviation) spent to run different number of concurrent jobs. The absolute numbers and ratio between the IOreplay run and the real application run are also depicted in the table 6.2. The numbers are roughly the same up to nine of concurrent jobs. This was expected because of the way the IOreplay works and number of CPU cores in the machine. When overloading the node with more than eight jobs, the bottleneck starts to be the data processing (CPU). Contrariwise, the

# Jobs	IOreplay(s)	Real App(s)	Ratio(%)
1	160.4 ± 0.8	161.9 ± 0.4	100.93
2	168.4 ± 2.0	177.9 ± 4.5	105.67
3	188.1 ± 1.9	188.4 ± 4.7	100.21
4	212.7 ± 4.4	223.2 ± 1.3	104.96
5	248.7 ± 1.6	256.5 ± 3.6	103.15
6	294.8 ± 3.9	297.9 ± 1.5	101.04
7	345.3 ± 3.2	348.0 ± 3.5	100.78
8	390.2 ± 2.1	398.9 ± 6.3	102.25
9	433.9 ± 2.2	447.1 ± 3.6	103.03
10	485.4 ± 2.2	516.6 ± 12.7	106.42
11	536.3 ± 1.3	782.7 ± 54.5	145.95
12	590.6 ± 5.5	1251.3 ± 39.6	211.85

Table 6.2: Scaling of IOreplay application

IOreplay still tries to keep the time difference between the calls, so it just do less cycles in the spinning and thus runs faster than the original application.

The study clearly shows that the IOreplay can scale within a few percent of difference in comparison with the original application provided the recording of the traces had reasonable overhead and that we do not overload the node.

6.3 Conclusion

IOreplay, the application for benchmarking using a trace and replay mechanism, was introduced in this chapter. Several features as well as drawbacks of the tool were described. Most importantly, it was shown that it can be used for a very accurate benchmarking in comparison with running the original application. The tool is freely available at [42].

Chapter 7

Distributed File Systems

Once we have a detailed understanding of the workload (described in chapter 5) and having the desired tool for benchmarking, the IOreplay (described in chapter 6), we can finally evaluate selected distributed file systems. We have chosen Lustre, GPFS, Hadoop as they are the most used systems in high performance computing and NFS4.1 as it is a very interesting, emerging a promising protocol not only within the HEP community.

This chapter includes general guidelines for benchmarking that we try to follow as well as a description of the used testbed and benchmarking methodology. Then, individual file systems are described and evaluated.

7.1 Benchmarking Is Hard

Prior to describing testing environment and test results, it is important to point out that benchmarking file systems is a hard task. A great study summarizing nine years of file system and storage benchmarking [51] has been recently published. The authors had taken more than one hundred papers and looked for the common mistakes people do when evaluating storage systems. Surprisingly, almost all of them suffered from not following the well justified guidelines (see below) depicted in the paper.

The list of the most important reasons why the storage benchmarking is demanding follows.

- Multi-variate space
- Moving target
- Easy to take results out of context

- Hard to make tests comparable
- Extremely complex system

We cite the benchmarking guidelines for benchmarking as provided in the aforementioned paper.

- Explain What Was Done in as Much Detail as Possible - For example, if one decides to create one's own benchmark, the paper should detail what was done. If replaying traces, one should describe where they came from, how they were captured, and how they were replayed (what tool? what speed?). This can help others understand and validate the results.
- In Addition to Saying What Was Done, Say Why It Was Done That Way - For example, while it is important to note that one is using ext2 as a baseline for the analysis, it is just as important (or perhaps even more so) to discuss why it is a fair comparison. Similarly, it is useful for readers to know why one ran that random-read benchmark so that they know what conclusions to draw from the results.

Among the two important but rather generic advice, one should also honor following recommendations.

- Choose right benchmark - The chosen benchmark should be repeatable, ie. testing the same operations as in previous runs and should be I/O bound[1]. The workloads should also be well understood and correspond with the metric one wants to measure.
- Have the state of the system under control - Major factors that can affect results are cache state, ZCAV[2] effects, file system aging, and nonessential processes running during the benchmark as well as other computers sharing the network equipment in case of distributed systems. Keeping the desired state of caches is crucial, as it could have huge effect on the results.

[1]Interestingly, not all of them are. For example, as shown in the [51], a popular compile test was run against the file system that was intentionally slowed down by the factor of 32, which resulted in a wall-clock slowdown of only 3%-5%.

[2]Zoned Constant Angular Velocity, a technology that stores more sectors per track on outer tracks than on inner tracks of a hard drive. That causes faster speeds when accessing outer tracks than the inner ones.

- Use multiple runs - This is significant to identify errors in benchmark setups and also recognize how confident one can be about the results (see next point). One has to ensure that the runs are identical.
- Make tests comparable - For example, if testing different file systems, one should test it under the exactly the same conditions.
- Provide statistical information - They can help readers to see how stable the tests were.
- Use scripts - A lot of mistakes associated with manual repetitive tasks could be prevented this way.
- Make tests repeatable - One can than validate the results. There is always someone who doubt the correctness of the test. It also concerns the next point.
- Provide as much information as you can - It includes software and hardware used, its parameters, how the tests were run.

We tried to follow these guidelines, to ensure the results obtained are valid.

7.2 Testbed

The testbed consisted of twenty servers with identical configuration: Hewlett-Packard Proliant DL 140, 32bit Intel Xeon CPU 3.06GHz, 2GB RAM, 80GB Seagate Barracuda 7200.10 HDD. In each test, one of them was used as a metadata server. With an exception of Hadoop where all nodes besides the metadata server were used as both data server and clients, there were always nine data servers and ten clients, where tests were run. All servers shared one network switch that was connected with 1Gbps line to each server. The architecture of the testbed is depicted in the figure 7.1. Prior to any tests, all nodes were re-installed with Scientific Linux v5.4, that comes with 2.6.18-164.11.1 version of kernel.

7.2.1 Evaluating the Testbed

In order to understand the results obtained from benchmarking distributed systems and to predict the limits of the system, performance

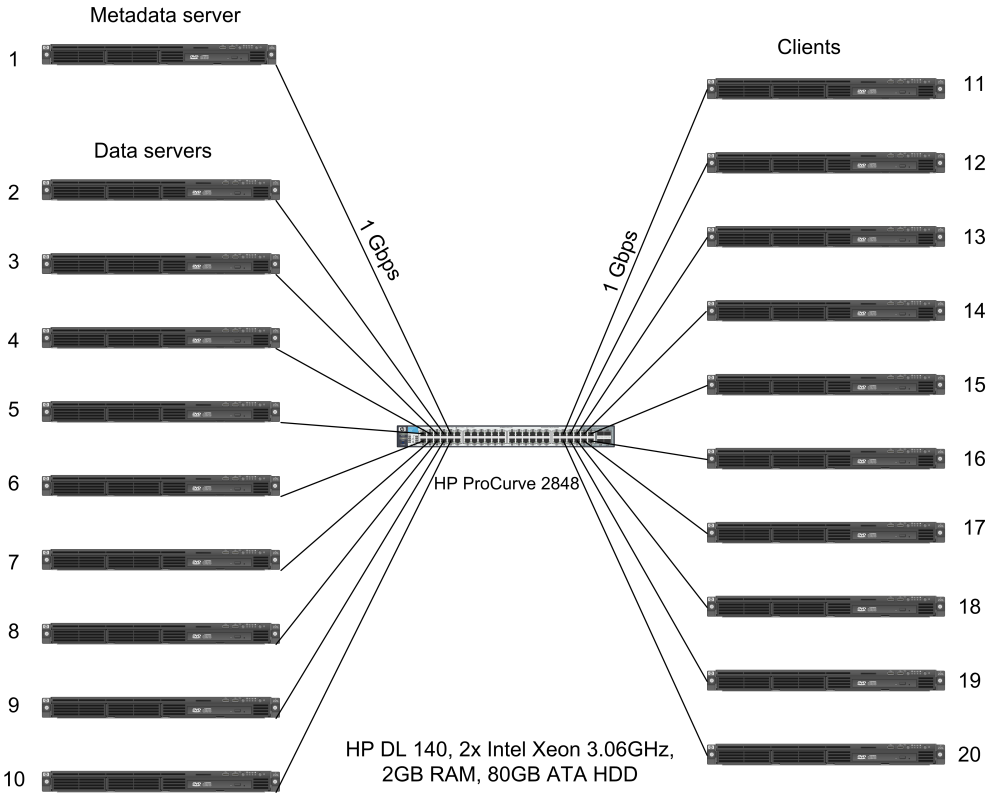


Figure 7.1: Testbed architecture

tests of individual servers have been done. This is also important to ensure that tests are not biased by a particular server which is significantly faster or slower than the others.

The figure 7.2 shows performance (axis y, kB/s) of individual storage servers (numbers in legend) when using sequential mode of IOZone [15] benchmark with different number of parallel threads (axis x). Every test wrote 2GB of data in summary (i.e. 500MB per thread for the 4-thread test). One gigabyte of RAM was by used by a small application to ensure leaving another one for the page cache of the operating system. This was done to ensure that the tests hit the disk while still measuring influence of RAM access speed. Except for the one thread test, where two nodes appeared to be faster than others, all of the nodes performed similarly.

The network performance of the servers were also measured using the `iperf` tool. All nodes proved to be able to fully saturate its gigabit link.

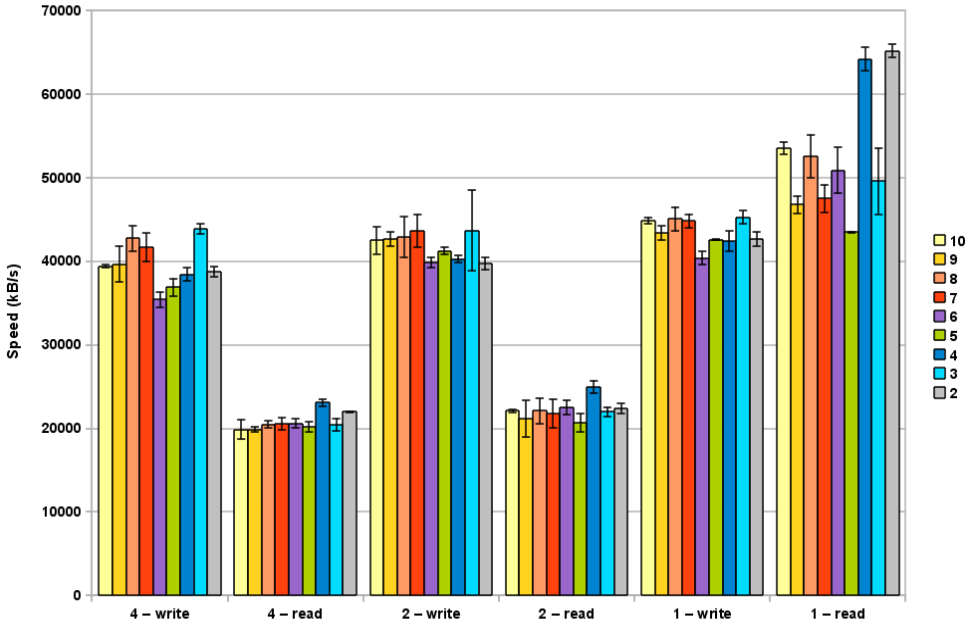


Figure 7.2: Testbed evaluation

7.2.2 Testing Methodology

Six different tests were run on the system. These tests correspond to analysis and reconstruction jobs of LHCb and CMS experiments, and two types of ATLAS analysis job, that were described in chapter 5. We provide a brief description of each test. Please refer to 5 for more details.

- **AtlasOld** - 9x 250MB out of 1GB files read, a lot of useless seeks (not followed by read or write calls). The effective seeks were usually within few megabytes.
- **AtlasNew** - 9x 250MB out of 1GB files read, no useless seeks. Strictly sequential with caching.
- **CMSAn** - 1984MB read from a 4GB file. Strictly sequential, caching, small gaps between individual calls.
- **CMSReco** - 424MB read from a 4GB file (only beginning of the file). Strictly sequential, some backward seeking usually within 20 megabytes. Almost 3 thousand files opened in total.

- **LHCbAn** - whole 875MB file was read with approximately 50 thousands of read requests. There were seeks back and forth in the file with distances of hundreds of megabytes.
- **LHCbReco** - 788MB out of 788MB file was read, strictly sequential in small requests (15kB) resulting in approximately 52 thousands reads. No effective seeking in the file.

The traces were replayed using the IOreplay application (see 6.2) with different time modes for different tests. Although we had access to real life jobs of the ATLAS experiment, it was not possible to used them in our testbed, because the ATLAS software is 64-bit only and the nodes were only 32-bit. The time modes used per test are listed in the table 7.1. The reason for not using the same time mode is that the reconstruction jobs lasted for a long time and were generally CPU and not IO bound. Differences in results using `diff` mode would be minimal, and thus `AFAP` mode was used. Moreover, for AtlasOld test, the scaling factor of 0.4 for think-times was used to compensate for the overhead when recording the traces.

Test	**Time mode**	**Scaling**
AtlasOld	Diff	0.4
AtlasNew	Diff	-
CMSAn	Diff	-
CMSReco	AFAP	-
LHCbAn	Diff	-
LHCbReco	AFAP	-

Table 7.1: Testing methodology: different replaying time modes

Each test was run three times to spot any errors and also to get information of the test stability. The manner in which tests were run is provided in listings 7.1.

```
for Count in 1 2 4 8 10 20; do #number of concurrent jobs
  for run in 1 2 3; do # run each test three times
    for Test in atlasnew atlasold lhcbreco lhcbanal cmsreco cmsanal; do
        RunTest $Test $Count $run
    done
  done
done
```

Listing 7.1: Test Methodology

It is important to note that the inner-most loop was changing the test case, which ensured that caches on both, client and server side, were rewritten before repeating the same test again. We intentionally did not flush the caches manually to also benchmark their behavior. The only tests that were influenced by not flushing the caches were the 1-job tests of the ATLAS experiment, where the second and third runs were faster than the first one as part of the data files stayed in the cache of data servers.

The average execution time of one job for the six tested workloads and different number of concurrent jobs was measured. Each job accessed its own copy of the data files, but shared all other files (such as libraries). The horizontal lines on each bar in the following plots display standard deviation of the measurement.

We also measured amount of data transfered to and from clients. Outbound traffic is shown only for Hadoop where it is significant, but not for the other systems as it was negligible. Inbound traffic per one host is shown for each system.

The numbers in legends mean count of concurrent jobs. For network plots, zero means data transfered from a disk by running the job on a local node.

7.2.3 Other Metrics

The performance of a file system is clearly not the only parameter by which one decides which file system to use. Other, maybe even more important factors are its stability, redundancy options, maintenance, availability of monitoring tools, life cycle management, disaster recovery options and availability of documentation and support. The stability of a system was hard to evaluate due to the limited testbed size, but at least the behavior of a data server loss was tested. The other aspects were evaluated from documentation and technical reports available.

7.3 Lustre

Lustre is the most commonly used open-source distributed file system in HPC[3] environments. It is very popular because of its performance.

The following section evaluates Lustre v1.8.4.

[3]High Performance Computing

7.3.1 Architecture

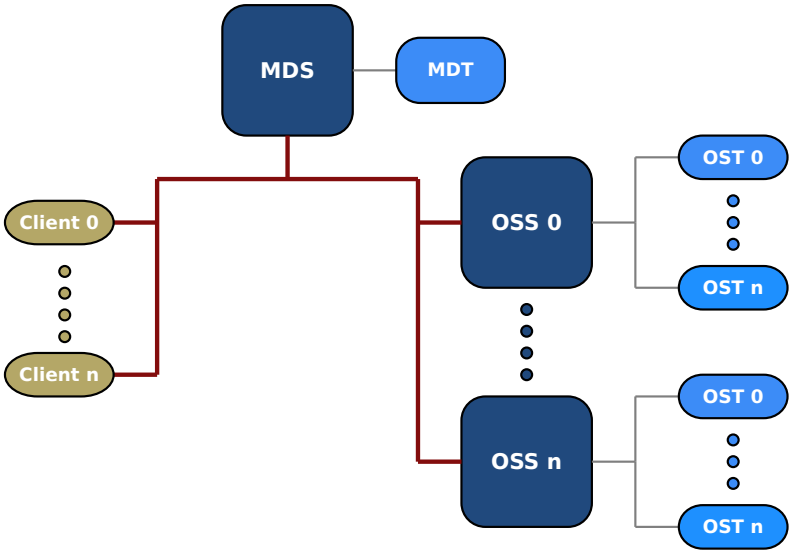
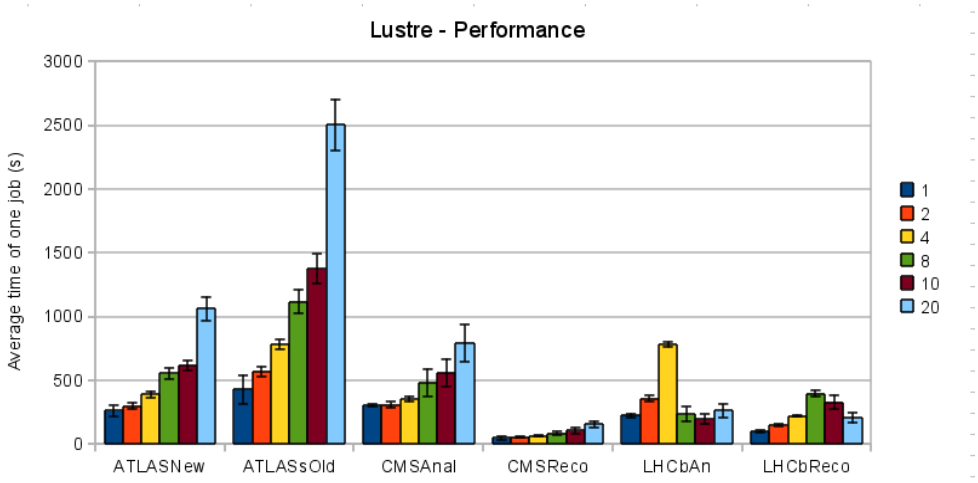


Figure 7.3: Lustre architecture

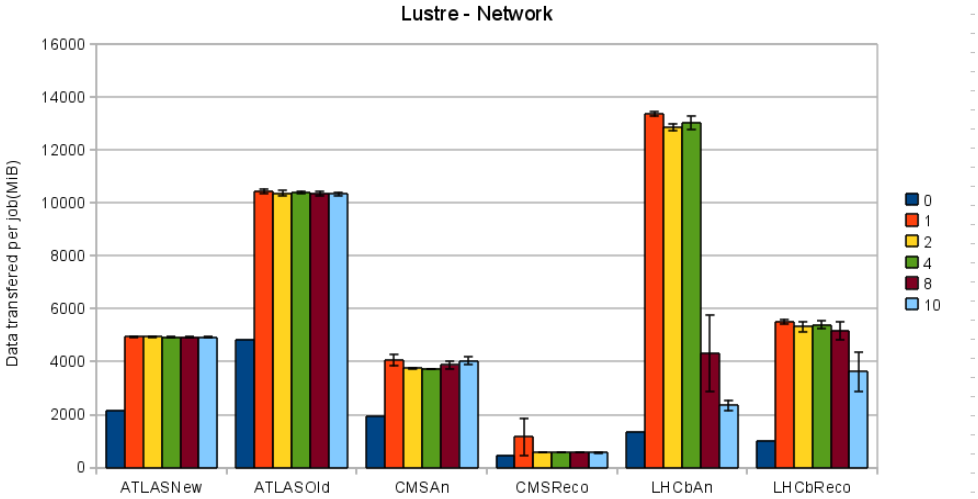
Lustre file system consists of two types of servers, the Metadata servers (MDSs) and Object Storage Servers (OSSs), see 7.3. The MDS is connected to a MetaData Target (MDT), which is typically a storage device that actually stores the metadata of the Lustre system. If the MDS fails, the service can failover to second MDS server that is connected to the same MDT. The same concept is used with OSS servers. OSS servers are connected to one, or many Object Storage Targets (OSTs) which are the storage devices that store the files in the system. The OSTs can be reconnected to other OSSs, if it is physically possible.

One of the main factors leading to the high performance of Lustre file systems is the ability to stripe data over multiple OSTs. The stripe count can be set on a file system, directory, or file level.

On the other hand, file replicating is not possible in the Lustre and thus loss of an OST results in an immediate data loss. That is why it is recommended to run at least RAID-5 or, preferably, RAID-6 on the storage servers. Sudden nonavailability of the MDT is fatal for the whole file system, and that is why it is critical to run this service on reliable hardware with regular backups.



(a) Average job duration



(b) Data transferred per one job

Figure 7.4: Performance of the Lustre file system

7.3.2 Performance

Our instance of the Lustre file system has been configured to stripe every file to 100MB chunks to maximally three servers.

The performance plots (see 7.2.2 for description of the tests) are depicted in figure 7.4.

The plots show expected behavior for ATLAS and LHCb jobs. Both types of LHCb jobs show unexpected pattern, when running more jobs in parallel results in faster average job execution. The network usage plot

confirms that, showing a lot of data transfered for 1, 2, 4 and partly also for 8 instances. We first believed that there was an error in the test scripts but this reason was ruled out by results of benchmarking other systems. A temporary dysfunction of the network or other hardware was also ruled out by re-running the same tests more times.

The phenomenon has not been entirely understood, but one of the possible explanations is probably an effect of client caching together with a quite large stripe size of 100MB. The LHCb jobs involved usage of several thousand of files, one data source file and two condition databases that were accessed randomly in the first and last minute of the job. These are shared for all jobs, so accesses to them for second and later jobs were cached by the system. It could also happen that the clients' kernel learned from the previous runs that this data are likely to be accessed so it kept them cached.

The effect is more visible for the LHCb analysis job as it accessed the data file in a random fashion.

It would be interesting to see whether shrinking the stripe size would improve the behavior and performance as there seems to be indication that choosing smaller stripe size could help. Unfortunately, these tests were not performed due to time constraints.

The detailed comparison with other file systems can found in section 7.7.

7.3.3 Maintenance

Following sections were written with help of the official manual [18] and the report of Lustre evaluation at CERN [52] which provides a neutral view of Lustre as a general purpose file system.

Requirements

Lustre can be installed on any Linux system, but it requires a special kernel on servers and at least kernel-modules on clients. None of Linux distributions the author is aware of provides the patched kernels in their repositories, so it is up to administrators of a cluster to maintain it and keep it up to date. Kernels for some distributions are provided on Lustre official web pages, and the guidelines for building a new kernel from source code are provided in the Lustre manual.

Documentation

The Lustre is provided with an excellent manual. The comprehensive manual describes both architecture of the system and administrator's guide on almost six hundred pages. Basically everything is included: from installation and configuration to disaster recovering, tuning, backing up and debugging. There was no need to consult any other information sources when installing and configuring the system. Lustre also comes with its active mailing list.

Redundancy

There are two types of redundancy to care of, for data and metadata.

The metadata server is by default a single point of failure and Lustre itself does not provide any mechanism to solve this issue. However, the manual describes a way how to use a second passive instance of MDS and a heartbeat service that would enable failover of the MDS server. However, the storage of metadata, the MDT, must not ever go down.

Lustre does not provide any redundancy for data. That is why the use of RAID-5, or preferably RAID-6 arrays is recommended in the manual. Because of the stripping of the stored files, outage of a storage server can cause unavailability of more files than expected.

Lustre can be configured in two failover modes for data servers (OSTs). Clients can either block until an OST is available again, which is the default option, or immediately receive an EIO error after timeout.

Backup and Recovery

Again, there are two types of data that one needs to backup. The metadata and the data itself.

Backing up metadata is crucial, as their loss results in the whole file system crash. The manual proposes to use LVM[4] as a storage device for metadata. This enables usage of snapshots for regular backups with a cost of partial performance hit.

Because the Lustre file system looks like a local file system from a client's point of view, the data can be backed up using any standard software such as Networker or Tivoli Storage Manager.

As noted in [52], the fact that these two kind of backups are decoupled can (and probably always will) cause inconsistency between metadata and data itself. This problem can be solved using file system checks, but

[4]Logical Volume Manager

it can result in a lost of some files if there is not any information about them on metadata server. Synchronization between MDS ans OSSs is not possible.

Life Cycle Management

From a perspective of life cycle management, one main problem arises. It is impossible to retire a storage server (OST) without any disruption of the service. The OST must be marked as read-only, all files on the OST must be copied to the new location (which triggers creation of new objects in the system) and then renamed to its original name. Finally, the OST is removed from the system. The clients with opened file descriptors will be impacted at some point of the procedure as the original files stop to exist.h

The monitoring of the system is also covered in the manual. There are several options to monitor a Lustre system: the built-in SNMP[5] module, `collectl`[6] and LMT2[20], which stands for Lustre Monitoring Tool.

Security

The Lustre supports distributed Access Control Lists (ACLs) and standard Linux user rights, but support for strong authentication using Kerberos is under development.

7.4 GPFS

GPFS is another file system used mainly in HPC environments. It has a slightly different architecture to Lustre, because it stores metadata and data on the same servers. It also generally has more features. Its main drawback is that it is not free - it is licensed by IBM, the company that developed the system. The licenses for these tests were kindly provided by Petr Plodik, an IBM employee.

GPFS v3.4.0.2 is described in this section.

7.4.1 Architecture

GPFS is often compared to Lustre as they both focus on the same segment. Their architecture is slightly different because GPFS stores meta-

[5]Simple Network Management Protocol, a protocol used mostly in network management systems to monitor network-attached devices for conditions that warrant administrative attention

data and data on the same servers.

A vast variety of configurations is possible using GPFS. It can stripe, replicate (max two copies) both data and metadata and distribute metadata over selected set of data servers. It also supports storage pools and policies (e.g. place files bigger than 8GB to storage pool SATA). GPFS deploys its own caching in GPFS daemon (not in-kernel).

7.4.2 Performance

The same testbed was used as with in the Lustre benchmark. Our system used 256K blocks with 2 metadata replicas and only 1 replica of data.

As opposed to Lustre, the GPFS runs proved to be much more stable. There is no unexpected behavior and the network usage seems to be stable throughout the tests. This could be probably explained by smaller block size and better caching prediction.

The comparison with other systems with further comments is provided in the section 7.7.

7.4.3 Maintenance

Following sections were written with help of the official manual [11].

Requirements

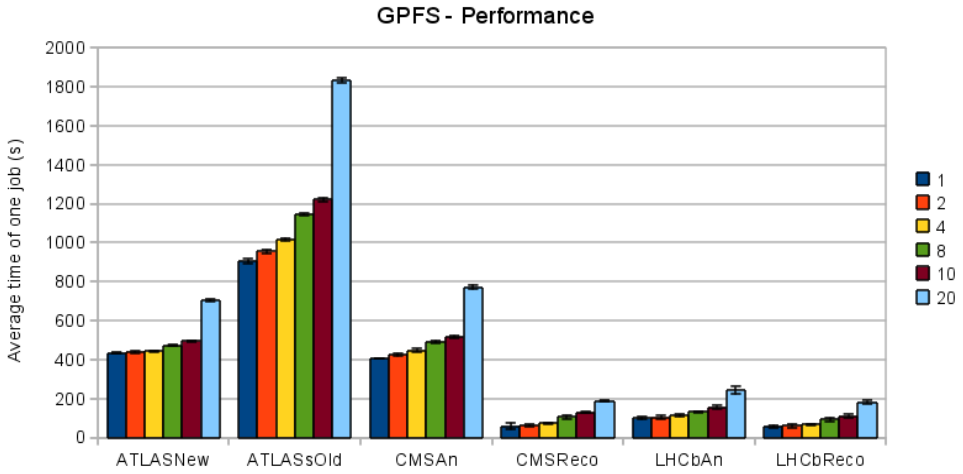
GPFS system can be run on almost every Linux systems that uses supported kernel as a kernel module must be compiled and installed. Officially, only SUSE and Red Hat are supported, but we managed to installed on Scientific Linux v5.4 as well, as it basically does not differ much. GPFS is also supported on AIX platforms.

Documentation

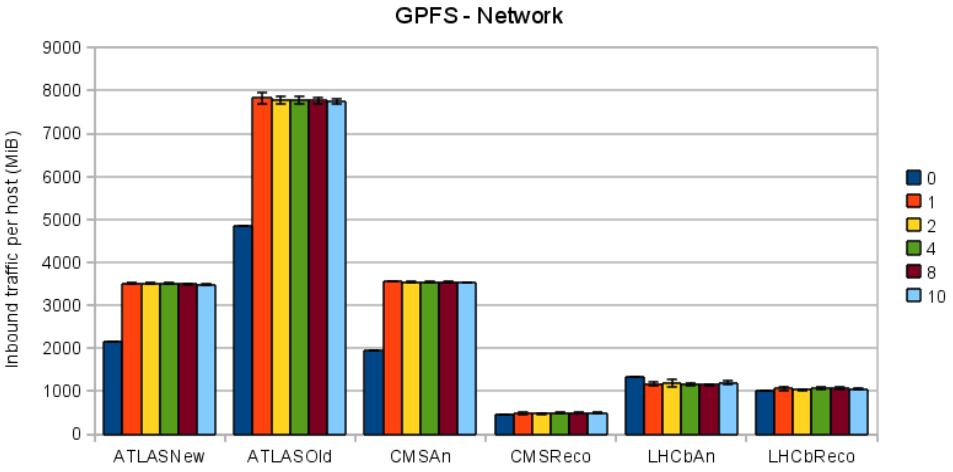
Documentation of GPFS is on very high level and is freely available at [10]. There are also support forums that seems to be active.

Redundancy

GPFS can be configured as redundant solution. It is possible to have two copies (maximum) of data as well as metadata. Similarly to Lustre, it is also possible to define more storage servers that has access to one disk array. GPFS also uses a similar concept of rack-awareness as in Hadoop,



(a) Average job duration



(b) Data transferred per one job

Figure 7.5: Performance of the GPFS file system

which is here called failure groups. During data duplication, GPFS makes sure that copies are distributed to different failure groups.

Backup and Recovery

GPFS provides `mmbackup` tool to ease backups. It supports full as well as incremental backups. It takes advantage of snapshot function of the GPFS system, so the backup can be resumed. Standard backup software can also be used.

Life cycle management

The data servers in the GPFS systems can be retired without any disruption to the service using `mmdeldisk` command.

Monitoring of GPFS cluster can be done using SNMP[6] protocol. It supports periodic polling for performance metrics as well as triggering errors. GPFS also comes with integrated `mmpmon` command that can monitor node on which it is run as well as other nodes. Advanced information as read block size or latency histograms can be displayed.

Generic tools like Ganglia or Nagios can also be used.

Security

GPFS supports standard Linux user/group permission model as well as ACLs. Strong authentication using Kerberos is not supported.

7.5 HDFS

The HDFS system is a part of the Hadoop system that is being developed by Apache community [29]. Furthermore Hadoop is also constituted of another part called MapReduce which is a system for distributing (mapping) jobs to the nodes where data are located. However, the terms Hadoop and HDFS are often used interchangeably and we decided to not break this convention. When using word 'Hadoop' in this chapter, we only mean the HDFS part of it. The typical installation of Hadoop thus consists of hundrend of worker nodes, each one equipped with a couple of hard drives.

The system is freely available at [29].

Because of different philosophy of the system, different testbed has been used. Please refer to 7.5.2 for details.

The following section evaluates HDFS v0.21.

7.5.1 Architecture

Hadoop has a different philosophy in comparison with other file systems described in this thesis. The outage of a server in a Hadoop system is regarded as a part of normal operations and not as an exceptional state. The system uses file replication for achieving stability and redundancy

[6] Simple Network Management Protocol

which enables usage of commodity hardware servers without any RAID setups.

The Hadoop system consists of one and only one name server that is single point of failure of the system. To scale with possibly high number of data servers and clients, system's metadata are kept entirely in memory during operation.

Unlike the other distributed file systems presented that are more or less POSIX-like[7], HDFS only supports reading from existing files, creating new files and appending to the existing files. Changing files is not possible. This is to ease the complexity of keeping all replicas up to date. Fortunately, this is not an issue for HEP experiments, as the data files are never modified.

All files are split to 128 megabytes blocks by default. A block forms a basic unit of data in the system. Each block is than replicated to defined number of servers (can be defined per file, directory or whole system), there seems to be no limit in the number of replicas in the system. The HDFS is rack-aware which means that it ensures that replicas are scattered across different racks. It also uses the information when retrieving the closest replica of a requested block.

A heartbeat service for all servers with a default period of three seconds is put in place to enable the NameNode (metadata server) to spot a failed data node quickly. It then immediately triggers replication of under-replicated blocks.

The Hadoop is written completely in Java and can only be mounted as a normal file system through FUSE.

It was recently announced that Cloudera, a provider of Hadoop-based data management software, will invest 25 million dollars into development of Hadoop.

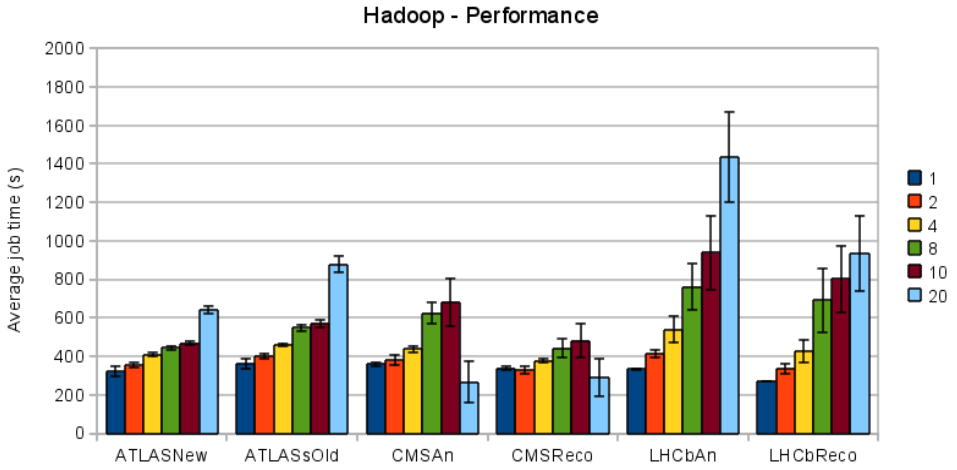
7.5.2 Performance

We used a slightly different architecture of the testbed when benchmarking Hadoop. One server acted as a metadata node, whereas all the remaining servers stored the data. The number of clients (ten), that we ran jobs on, remained unchanged.

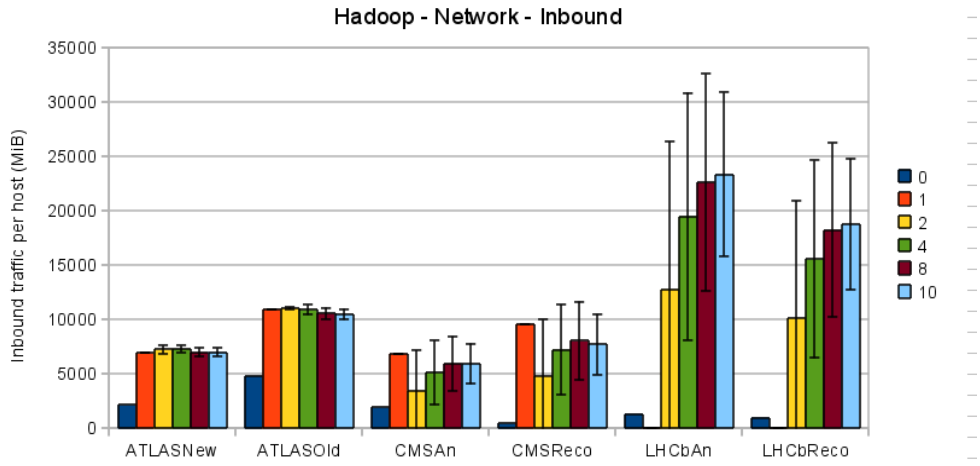
All the files in the system had replication factor of two.

It should be noted that we used FUSE in order to use the system as a regular Linux file system. Usage of FUSE is known to have quite

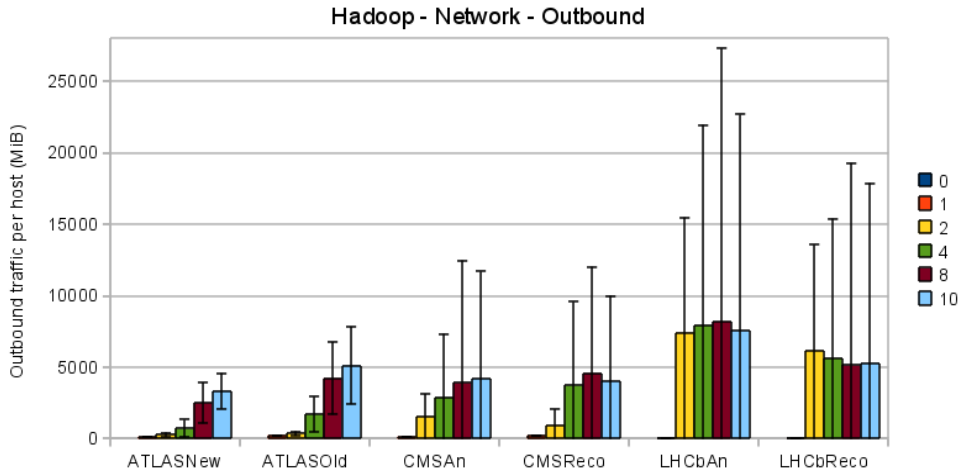
[7]There are very few distributed file systems that honors the POSIX file semantics completely. Only GPFS of the the presented systems is fully POSIX complaint. Exact details are beyond the scope of the thesis.



(a) Average job duration



(b) Inbound traffic per one job



(c) Outbound traffic per one job

Figure 7.6: Performance of the Hadoop file system

a big impact on the performance. We have seen very high (up to 80%) utilization of CPUs on client machines during the tests.

It is important to realize that the ROOT framework has a plugin for HDFS system, so usage of FUSE is not necessary in real life environments.

The question is how much the tests were influenced by the usage of the FUSE module. We believe that the tests that were run in `diff` time mode were not biased much as they still waited the right amount of time (if the system was fast enough). The `AFAP` mode was clearly influenced much more. From another point of view, the necessity of using FUSE can be regarded as an inseparable part of the Hadoop system when using it as a regular file system (i.e. mounting it).

The overall instability (large values of standard deviations) is caused be the randomness of placing the data - some jobs happened to have majority of their data locally on the client nodes whereas others had to transfer the data first.

Despite the above mentioned issues, the results seem to be reasonable in general.

Almost zero usage of inbound network for one instance of LHCb reconstruction and analysis jobs is due to co-location of the data files where the source data were placed on the client node that the job has been run on.

The detailed comparison with other file systems can found in section 7.7.

7.5.3 Maintenance

Requirements

Basically any operating system with a Java Virtual machine in at least required version (1.5.0 now) can be used for running HDFS.

Documentation

The HDFS does not come with an all-in-one manual, but the documentation is placed on the official web page of the project[29]. The author found the documentation clear and sufficient even though some of the procedures were a bit outdated. There are also plenty of blogs and mailing lists about Hadoop.

Redundancy

There are two types of redundancy to care of, data and metadata.

The HDFS file system is basically based on a redundancy of data. Every block (see above) is replicated to the predefined number of servers. The lost of a data node is registered within seconds and the automatic replication of under-replicated blocks is triggered within minutes. This makes a use of commodity hardware without any safety measures such as RAID setups possible.

The metadata, on the other hand, must be kept on a very stable hardware, as they reside on just one metadata server. Moreover, HDFS keeps all the information both on disk and in-memory for speed reasons. There are two files that store the Metadata. The FSImage that actually hold the data and Editlog that serves as a journal. It is possible to specify multiple locations to which the metadata are written (duplicated). It is therefore possible for example to save the metadata to a remote NFS server.

Backup and Recovery

Data itself are somehow backed-up for free as there are always at least two (or more) replicas of the file in the system. The true backups can however be done during normal operations with the advantage that the data can not be changed by design during the backup (only appends are possible).

Backing up metadata means to retrieve a FSImage and EditLog files from NameNode by either copying replica files or downloading it through web interface.

If the FSImage nad Editlog files are lost, it is not possible to recover the NameNode from Dataservers.

Life Cycle Management

Retiring a data node is just as easy as shutting it down. The data are automatically replicated somewhere else. In a case of decommissioning of many servers from the cluster, chance that all replicas of some files are on these servers increases. That is why Hadoop offers a command that can automate decommissioning of multiple nodes.

Change of the NameNode requires downtime of a whole cluster.

Monitoring of the system can be done through standard tools such as Nagios, Ganglia or Munin. There are specific plugins already available on the Internet. Hadoop installation also includes a web page on which one can see an overview of the system status.

Security

Security in Hadoop is, as of version 0.21, very weak. Hadoop supports Unix-like users, groups and permissions, but the problem is in how it determines current users - it executes `whoami` and `bash -c group` commands. One can write his own scripts and put it into PATH to fool the system. Moreover, DataNodes do not authenticate users at all, so one can read or write any block he wants provided he knows the right block ID.

This problem should be address in v0.22 release where Kerberos support is planned.

7.6 NFS4.1

NFS 4.1[23] is a recently standardized protocol for accessing data on a distributed system.

The minor version 1 of the NFS protocol brings supports for sessions (and thus minimizing traffic needed for individual requests) but most importantly it introduces the pNFS, parallel NFS protocol, the protocol to access files distributed on many disk servers. The pNFS cluster consists of a name server, data servers storing actual data and clients.

We were interested in in-kernel implementation of the protocol, as it will come for free with Linux kernel. It seems that the current model for deploying pNFS is to use in-kernel clients that are being ported to main kernel branch just now (October 2010). On the other hand, the server part will be initially developed by various storage vendors such BlueArc, Panasas or EMC. HEP community also started to implement server side of pNFS in DPM[47] and dCache[45].

Despite the ongoing rapid development, client as well as server part is still not considered stable.

We tried to evaluate the in-kernel server implementation (SPNFSD) because both DPM and dCache implementations were not ready at the time of writing. The same applies for implementations of storage vendors.

Unfortunately, we have not succeeded in evaluating of the NFS4.1 because of below mentioned reasons.

7.6.1 Installation

We first tried to install NFS4.1 kernel (2.6.34pnfs-1) and user-space tools to existing Scientific Linux v5.4 systems. Unfortunately, it turned out

that due to several file collisions between distinct packages and their certain versions, this would be a really hard task.

Then, we decided to install Fedora Core 13 Linux (kernel 2.6.34.7-58.pnfs35.2010.09.14.fc13), which already has the required user-space packages available. The installation was successful but the first tests showed very strange behavior of the pNFS cluster. It seemed that data are transfered through the Metadata server, which caused very poor performance. We have not managed to fix this issue as nobody answered on the pnfs mailing list. We consider this problem to be caused by the fact that the development is still on-going.

We have decided not to benchmark the solution.

7.7 Conclusion

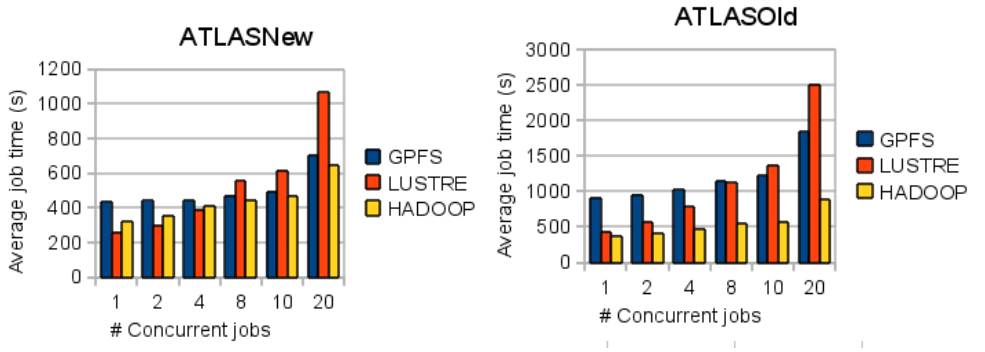
Three different storage solutions had been benchmarked using the IOreplay(see 6.2) tool and evaluated by other non-performance metrics. It is clear that choosing one system rather then another always depends on the given usage, workload, budget and also personal preferences. We tried to focus on the typical workload in the HEP community to help deciding among presented file systems. Unfortunately, we did not manage to get the in-kernel NFS 4.1 implementation working.

Summary plots that compare file systems performance (figure 7.7) and network usage (figure 7.8) in the different test cases follow.

Generally, it seems that GPFS is the most stable and fastest solution. The main exception is the `ATLASOld` test case, where Hadoop shows very high performance. We remark, that this test case performed extremely high number of useless seeks. It is believed that the dominance of Hadoop was caused by two reasons - a) due to a `diff` time mode used, so the FUSE overhead was not reflected and b) real seeking within few tens of megabytes, which hit the block size of the Hadoop system.

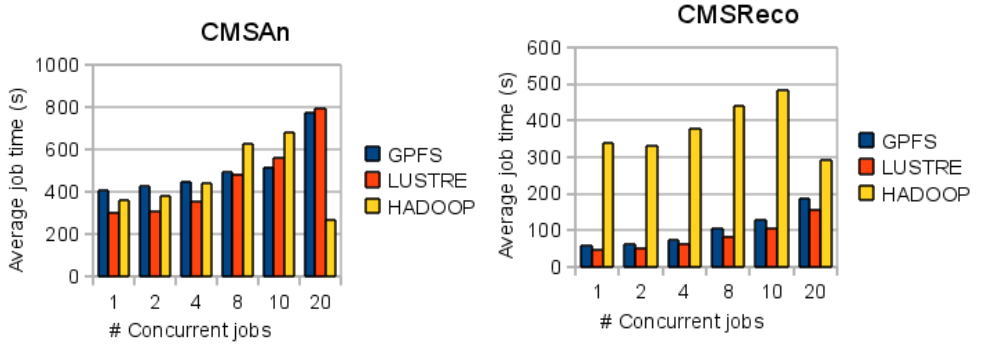
Not considering the `LHCbAn` and `LHCbReco` test cases, Lustre shown to be capable to perform very similarly to the GPFS system. It is possible that if changing the stripe size in the Lustre system, it could prevent the system from this misbehaving.

Taking a look in to the future, it is probably best to look at the `ATLASNew` and `CMSReco` use cases, where the new file format is used in the first case and not much could be improved in the latter case, which is strictly sequential (see figure 5.3). Caching was used in both the use cases. For the `ATLASNew`, all three solutions seems to perform similarly with exception of Lustre for twenty concurrent jobs.



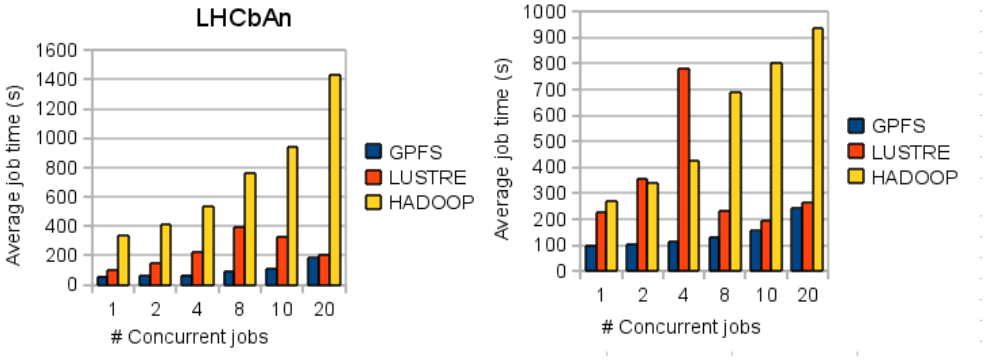
(a) ATLASNew

(b) ATLASOld



(c) CMSAn

(d) CMSReco

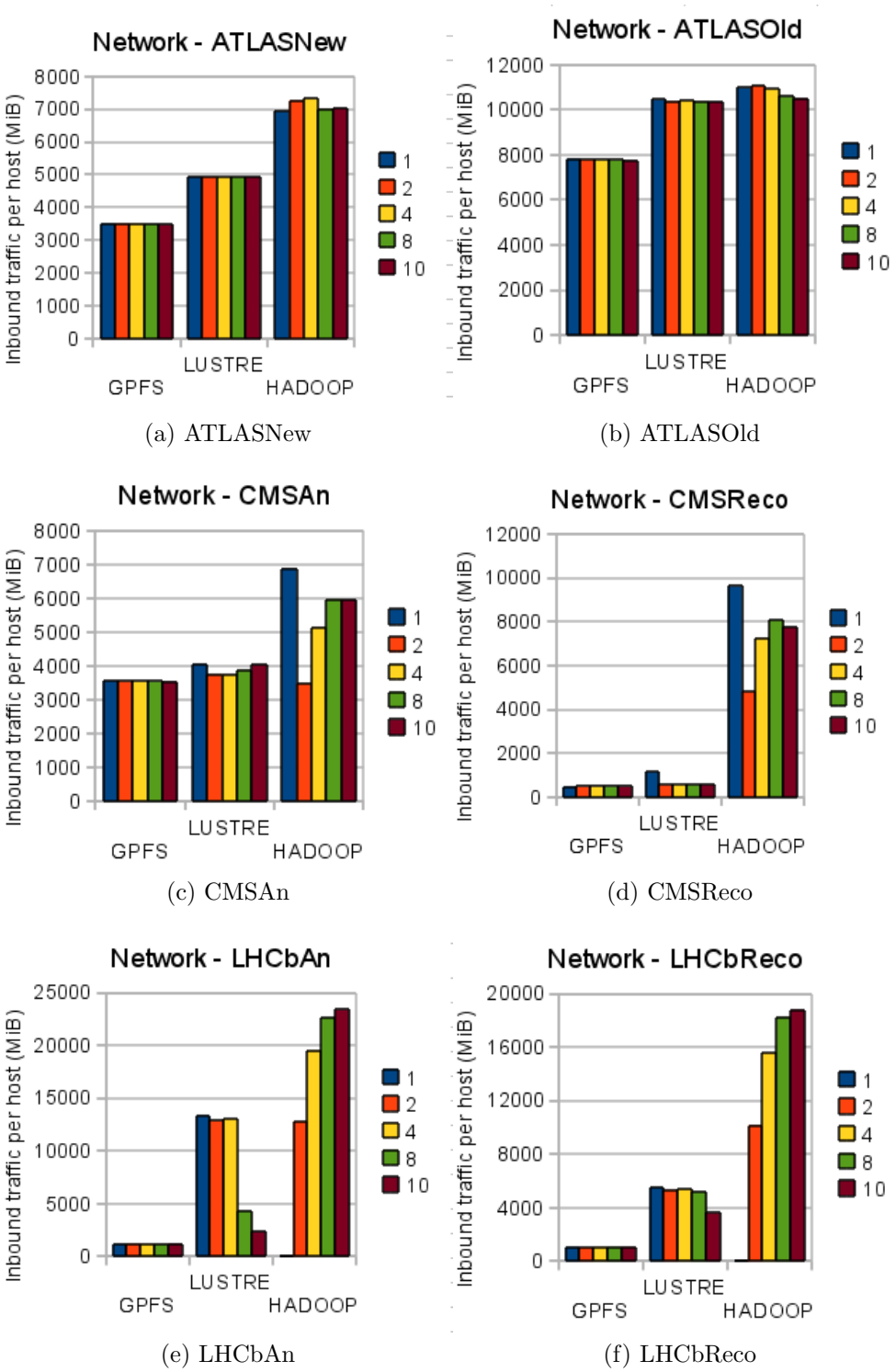


(e) LHCbAn

(f) LHCbReco

Figure 7.7: Performance comparison of various distributed file systems (lower is better)

On contrary, Hadoop has a problem with the `CMSReco` case, whereas the other two file systems seems to perform within 10% of difference.



(a) ATLASNew

(b) ATLASOld

(c) CMSAn

(d) CMSReco

(e) LHCbAn

(f) LHCbReco

Figure 7.8: Network usage of distributed file systems (lower is better)

From the point of network usage, GPFS seems to be the less demanding solution, whereas Hadoop requires in some cases order of magnitude more of the network usage to perform the job.

Comparing non-performance metrics is a bit tricky as the systems use different philosophy and are typically used for different purposes. Moreover, absence of one feature can be a show-stopper for somebody and negligible thing for somebody else.

It seems that the GPFS file system is the most mature and featurefull one from the list. It supports a vast number of configuration option. For example, one can decide whether replicate data/metadata or not, which is not possible in the Lustre. The main reason why not to choose GPFS is its price. The list price for 10 storage servers and 250 clients, all 8 cores machines, is around 90 000€.

Hadoop, on the other hand, does not support striping as opposed to the other two systems. It is also true that Hadoop will generally requires more hard drives to store the same amount of data, which could have negative impact on TCO[8] when also counting power consumption. On the other hand, it can be used on cheap commodity hardware so this effect is probably negligible.

If one had to choose between Hadoop and Lustre, the decision would probably be also based on a type of hardware he uses. In case of reliable storage hardware, one can go for Lustre system. In case of non-reliable, cheap storage and hard drives located in worker nodes, Hadoop could be an option.

Even though it is always hard to generalize the results into some "rules of thumbs", we try to provide general observations:

- Hadoop can be a very efficient solution if the workload uses a lot of seeks within the distance of Hadoop block size (`AtlasOld` case). On the other hand, it is very inefficient when there are seeks out of the distance of Hadoop block size (`CMSReco` and `LHCbAn`).
- Hadoop seems to use much more network bandwidth than other solutions.
- Even though that GPFS and Lustre performance seems to be similar, GPFS is slower than Lustre when using just few jobs, but scales

[8]Total Cost of Ownership

better than Lustre. However, this could be caused by different configured block sizes of the two systems.

- None of the solutions seems to be affected by useless seeking that is not followed by read or write requests.

Chapter 8

Related Work

8.1 IO profiling

Interestingly, it seems there are not many tools for user-space IO profiling of applications. We have to mention StraceAnalyzerNg[26] that use the same approach as IOprofiler but lacks several features. IBM has a tool for this task as well called HPCT I/O[14] but it is a commercial product.

8.2 IO replaying

The idea of of trace and replay mechanism is not new at all. There are many tools to accommodate this task in general. Unfortunately, there was always a good reason to not use is as it was recording/replaying on a different system layer than required or not freely available. The following list is certainly not exhaustive.

- Buttress[35], a toolkit for flexible and high fidelity IO benchmarking. It is a commercially licensed product.
- Tracefs and Replayfs[44], tracing and replaying at Linux VFS level. While very accurate, it would require code revision because of kernel changes since the paper was published (five years ago). Moreover, it requires kernel changes at the host where traces are recorded.
- blktrace, btrecord and btreplay, block-layer IO tracing and replaying tools at Linux. Given the block level nature of the tools, recording of traces is is always per whole node, and replaying traces is generally destructive (it writes directly to the block device, bypassing the file system layer, so it very likely rewrites some existing data and/or file system structures).

8.3 Benchmarking in HEP environment

There are quite a lot of benchmarks of various storage solutions done in HEP community. The common problem is the scope of the tests - they are usually run with just one type of job and the impact of access pattern is more guessed than understood.

Among others, several presentations of the HEPIX Storage Group[13] should be cited. The group uses ATLAS analysis and recently also CMS analysis jobs to benchmark distributed systems such as Lustre, GPFS, xRootd protocol or AFS with extensions that are installed on a testbed consisting of ten of up-to-date worker nodes and three storage servers. It seems that some effort is now being put in understanding and standardization of the workload, so it really represents jobs used nowadays.

There are also some tests performed at CERN: testing and evaluating Lustre[52], [46], and also xRootd protocol [34].

Chapter 9

Conclusions and Future work

9.1 Conclusions

The aim of the thesis was to explore current approaches and solutions in the field of data access in the WLCG. Using this knowledge and further analysis, the goal was to find inefficiencies in how applications access their data and to evaluate and benchmark new solutions of storing and accessing the data in a form of distributed file systems.

The overview of data flows has been provided in chapter 1. Chapter 2 deals with the data framework common to all four main LHC experiments. We have described how data are stored and what changes have been made to the framework recently, including their influence on performance.

To fully understand what type of workload HEP applications apply on a storage system, we decided to develop an IO profiling application. Several possible profiling approaches with their advantages and drawbacks have been described in chapter 3.

Based on requirements and information in chapter 3, we have developed an IO profiling application, the IOprofiler, described in chapter 4. The majority of author's own work begins in this chapter and continues through chapters 6 and 7 to conclusion.

We have then used IOprofiler to analyze access pattern of ATLAS, CMS and LHCb jobs. Although it has been developed specifically for this task, its usage is not limited to LHC experiments by design. Representative jobs of the three experiments have been profiled and described in chapter 5. Access pattern is very different across these jobs and sometimes shows quite inefficient behavior.

Because benchmarking using real LHC applications was hard to setup

and we wanted to present much easier way to reliably benchmark the systems, we have presented a tool for benchmarking using trace and replay mechanism, the IOreplay, in chapter 6. We have shown that it is capable of valid replaying with error within a few percent of the real application, provided certain requirements are met. Similarly to the IOprofiler, it can be used for virtually any application.

We have then benchmarked and described three distributed file systems: Lustre, GPFS and Hadoop, using the recorded workload of the LHC experiments in chapter 7.

It would be naive to expect this thesis to come with a "silver bullet", solution that would fit all the needs of the HEP community for data analysis. The contribution of the thesis lies in detailed understanding of the workload of different LHC experiments, discovery of bugs in the ROOT framework and in a fair comparison of various distributed file systems under different, well understood workloads. We have summarized and generalize, where possible, the results in 7.7. This can be a very valuable guideline when choosing the file system to use.

The IOprofiler and IOreplay tools we have developed to accomplish these tasks are also significant result of the thesis. IOprofiler can help any administrator or user to easily understand IO access pattern of their applications, while IOreplay can be used as a standalone benchmark in a situation when running the real application is not possible or practical. These tools are freely available[42].

Part of the thesis has been presented at International Conference on Computing in High Energy and Nuclear Physics 2010[41] with positive feedbacks.

9.2 Future work

IOreplay and IOprofile applications could benefit from new features. The first thing that should be added is support for not complete strace outputs, i.e. outputs from strace that was dynamically attached to a process. Calls like `dup` and `open` are missing in such traces, causing both tools to fail. There are two approaches to follow a) just to add a generic file name such as "FILE1" to those unknown files or b) use information from `/proc/PID/fd` system to discover the current *fd* mapping. Both of these tools could also benefit from the support of more system calls such as `pread`, `pwrite`, `rename` or `fsync`. Fortunately, these were not needed for analyzing and replaying the selected jobs.

More statistics could also be shown in the IOprofile application based

on the information it already collected. One such thing should be a timeline of files accessed during the execution.

A personal wish of the author is to start using the IOreplay as the benchmark for evaluation of tenders for new hardware at FZU, the current workplace of the author, where IOzone is currently used.

Focusing on the benchmarking part, there is always at least one parameter or option to tune when benchmarking such complex systems as storage systems are. One of the first things to try would be to change Lustre block size to a smaller value. Some of the emerging NFS4.1 implementations should be tried again in the near future as well.

List of Figures

List of Tables

Bibliography

[1] *ATLAS computing: Technical Design Report*. Technical Design Report ATLAS. CERN, Geneva, 2005.

[2] ALICE, A Large Ion Collider Experiment. `http://aliceinfo.cern.ch/Collaboration/index.html`, August 2010.

[3] ATLAS, A Toroidal LHC ApparatuS experiment. `http://atlas.web.cern.ch/Atlas/Collaboration/`, August 2010.

[4] BeStMan: Berkley Storage Manager. `https://twiki.grid.iu.edu/bin/view/ReleaseDocumentation/Bestman`, April 2010.

[5] CASTOR: CERN Advanced Storage Manager. `http://castor.web.cern.ch/castor/`, April 2010.

[6] Collectl. `http://collectl.sourceforge.net/`, September 2010.

[7] Compact Muon Spectrometer. `http://cms.cern.ch/`, August 2010.

[8] dCache. `http://www.dcache.org/`, July 2010.

[9] Disk pool manager. `https://svnweb.cern.ch/trac/lcgdm/wiki/Dpm`, July 2010.

[10] GPFS, The IBM General Parallel File System. `http://www-03.ibm.com/systems/software/gpfs/`, July 2010.

[11] GPFS v3.4 Information and Manuals. `http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/topic/com.ibm.cluster.gpfs.doc/gpfsbooks.html`, October 2010.

[12] GridFTP, an extension of the standard File Transfer Protocol (FTP) for use with Grid computing. `http://dev.globus.org/wiki/GridFTP/`, July 2010.

[13] HEPiX Storage Working Group. http://hepix.caspur.it/storage/index.php, October 2010.

[14] HPCT I/O. http://domino.research.ibm.com/comm/research_projects.nsf/pages/hpct.mio.html, October 2010.

[15] IOzone filesystem benchmark. http://www.iozone.org/, August 2010.

[16] Large Hadron Collider. http://lhc.web.cern.ch/lhc/, July 2010.

[17] LHCb, Large Hadron Collider beauty experiment. http://lhcb.web.cern.ch/, July 2010.

[18] Lustre 1.8 Operations Manual. http://wiki.lustre.org/images/3/3c/821-0035-12.pdf, July 2010.

[19] Lustre, a Network Clustering FS. http://www.lustre.org, July 2010.

[20] Lustre Monitoring Tool. http://lmt.sourceforge.net/, September 2010.

[21] matplotlib, a python 2D plotting library. http://matplotlib.sourceforge.net/, August 2010.

[22] MySQL: The world's most popular open source database. http://www.mysql.com/, May 2010.

[23] NFS4.1, Network File System (NFS) Version 4 Minor Version 1 Protocol. http://tools.ietf.org/html/rfc5661, July 2010.

[24] Plastic File System. http://plasticfs.sourceforge.net/, July 2010.

[25] StoRM: Storage Resource Manager. http://storm.forge.cnaf.infn.it/home/, June 2010.

[26] Strace analyzer, IO Profiling of Applications. http://clusterbuffer.wetpaint.com/page/Strace+Analyzer+-+Next+Generation, July 2010.

[27] Strace, system call trace tool. http://sourceforge.net/projects/strace/, July 2010.

[28] Systemtap: tracing/probing tool for complex tasks. http://sourceware.org/systemtap/, July 2010.

[29] The Apache Hadoop project. http://hadoop.apache.org/, October 2010.

[30] The Institute of Physics of Academy of Science of CR. http://www.fzu.cz, July 2010.

[31] The Scalla Software Suite: xrootd/cmsd. http://xrootd.slac.stanford.edu/, September 2010.

[32] Torque, an open source resource manager. http://www.clusterresources.com/products/torque-resource-manager.php, July 2010.

[33] Worldwide LHC Computing Grid. http://lcg.web.cern.ch/lcg/Default.htm/, July 2010.

[34] L. Abadie. Xrootd Scalability Testing. https://twiki.cern.ch/twiki/bin/view/LCG/Scalability_XrootD, October 2008.

[35] E. Anderson, M. Kallahalla, M. Uysal, and R. Swaminathan. Buttress: A Toolkit for Flexible and High Fidelity I/O Benchmarking. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 45–58, Berkeley, CA, USA, 2004. USENIX Association.

[36] M. Bencivenni, A. Carbone, A. Chierici, A. D'Apice, D. D. Girolamo, L. dell'Agnello, M. Donatelli, G. Donvito, A. Fella, A. Forti, F. Furano, D. Galli, A. Ghiselli, A. Italiano, E. Lanciotti, G. L. Re, L. Magnoni, U. Marconi, B. Martelli, M. Mazzucato, P. P. Ricci, F. Rosso, D. Salomoni, R. Santinelli, V. Sapunenko, V. Vagnoni, R. Veraldi, D. Vitlacil, S. Zani, and R. Zappi. Storage management solutions and performance tests at the infn tier-1. *J. Phys.: Conf. Ser.*, 119:052003, 2008.

[37] D. Bonacorsi. The CMS Computing Model. *Nuclear Physics B - Proceedings Supplements*, 172:53 – 56, 2007. Proceedings of the 10th Topical Seminar on Innovative Particle and Radiation Detectors, Proceedings of the 10th Topical Seminar on Innovative Particle and Radiation Detectors.

[38] N. Brook. LHCb Computing Model. Technical Report LHCb-2004-119. CERN-LHCb-2004-119, CERN, Geneva, Dec 2004.

[39] R. Brun, F. Rademakers, and S. Panacek. ROOT, an object oriented data analysis framework. 2000.

[40] P. Cortese, F. Carminati, C. W. Fabjan, L. Riccati, and H. de Groot. *ALICE computing: Technical Design Report*. Technical Design Report ALICE. CERN, Geneva, 2005.

[41] J. Horky. From Detailed Analysis of IO Pattern of the HEP Applications to Benchmark of New Storage Solutions. `http://117.103.105.177/MaKaC/contributionDisplay.py?contribId=314&sessionId=118&confId=3`, October 2010.

[42] J. Horky. IOapps toolkit - IOprofiler and IOreplay tools. `http://code.google.com/p/ioapps/`, October 2010.

[43] R. W. L. Jones and D. Barberis. The evolution of the ATLAS computing model. *J. Phys.: Conf. Ser.*, 219(7), 2010.

[44] N. Joukov, T. Wong, and E. Zadok. Accurate and efficient replaying of file system traces. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.

[45] Y. Kemp. dCache, LHC Data Analysis Using NFSv4.1 (pNFS): A Detailed Evaluation. `http://117.103.105.177/MaKaC/getFile.py/access?contribId=217&sessionId=60&resId=0&materialId=slides&confId=3`, October 2010.

[46] A. Peters. Lustre Scalability Testing. `https://twiki.cern.ch/twiki/bin/view/LCG/Scalability_Lustre`, October 2008.

[47] R. Rocha. Standard Protocols in DPM. `http://117.103.105.177/MaKaC/materialDisplay.py?contribId=100&sessionId=33&materialId=slides&confId=3`, October 2010.

[48] ROOT bug tracker. Useless seeking in ROOT using TreeCache. `http://savannah.cern.ch/bugs/?69845`, July 2010.

[49] G. A. Stewart, D. Cameron, G. A. Cowan, and G. McCance. Storage and Data Management in EGEE. In P. Coddington and A. Wendelborn, editors, *Fifth Australasian Symposium on Grid Computing*

and e-Research (AusGrid 2007), volume 68 of *CRPIT*, pages 69–77, Ballarat, Australia, 2007. ACS.

[50] Strace devel mailing list. Strace bug with corrupted output. `http://www.mail-archive.com/strace-devel@lists.sourceforge.net/msg01600.html`, July 2010.

[51] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright. A nine year study of file system and storage benchmarking. *Trans. Storage*, 4(2):1–56, 2008.

[52] A. Wiebalck, O. Barring, T. Bell, A. Horvath, P. Kelemen, and M. Santos. An Evaluation of Lustre as a General Purpose File System for CERN. Technical report, CERN, Geneva, Dec 2010.

[53] T. William. IOProf. `http://www.benchit.org/wiki/index.php/IOprof`, October 2010.

Appendix A

DVD Content

```
.
|- benchmark     - scripts and traces used for benchmarking HDFS system
|
| |-- atlas      - traces, mapfiles and ignore files for IOreplay (ATLAS)
| |-- cms        - traces, mapfiles and ignore files for IOreplay (CMS)
| |-- lhcb       - traces, mapfiles and ignore files for IOreplay (LHCb)
| |
| |-- hdfs.sh    - test script used for launching benchmarks for HDFS system
| |-- monnet.sh  - script used for monitoring of transfered data on a server
| |-- test.sh    - script used to launch selected job on one worker node
| `-- README     - description of files + manual how to use these script and
|                  what to change in order to benchmark Lustre and GPFS
|- docu    - documentation of ioapps suite, snapshot of web pages
|- ioapps  - containts source code of both IOreplay and IOprofiler,
|            man pages included
`- thesis  - text of this thesis in PDF format
```

Figure A.1: Content of the included DVD

Appendix B

Documentation

Because we have made the ioapps suite (IOprofile and IOreplay) publicly available at `http://code.google.com/p/ioapps/`, including installation manual, usage manual and screenshots, we provide a snapshot of the web pages as the documentation at the included DVD.

It is also important to note, that the applications comes with the standard Linux manual pages.