TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Netzarchitekturen und Netzdienste

Decentralized Data Storage and Processing in the Context of the LHC Experiments at CERN

Jakob Johannes Blomer

Vollständiger Abdruck der von der Fakultät für Informatik
der Technischen Universität München zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Uwe Baumgarten

Prüfer der Dissertation:

1. Univ.-Prof. Dr.-Ing. Georg Carle
2. Univ.-Prof. Dr. Dieter A. Kranzlmüller, Ludwig-Maximilians-Universität München
3. TUM Junior Fellow Dr. Thomas Fuhrmann

Die Dissertation wurde am 19.12.2011 bei der Technischen Universität München
eingereicht und durch die Fakultät für Informatik
am 01.06.2012 angenommen.

Acknowledgment

This work would not have been possible without the countless discussions with my supervisors Predrag Buncic and Thomas Fuhrmann and many others. In particular, I would like to thank my collegues Artem Harutyunyan, Axel Naumann, and Carlos Aguado Sánchez for so many fruitful ideas and suggestions. Many thanks to Benedikt Hegner and John Harvey for carefully reviewing the manuscript and for all their helpful comments. I want to thank the people who are close to me, supported me, and who always lent an ear to me. During my stay at Cern, I enjoyed the company of wonderful friends.

Abstract

The computing facilities used to process data for the experiments at the Large Hadron Collider (LHC) at CERN are scattered around the world. The embarrassingly parallel workload allows for use of various computing resources, such as computer centers comprising the Worldwide LHC Computing Grid, commercial and institutional cloud resources, as well as individual home PCs in "volunteer clouds". Unlike data, the experiment software and its operating system dependencies cannot be easily split into small chunks. Deployment of experiment software on distributed grid sites is challenging since it consists of millions of small files and changes frequently.

This thesis develops a systematic approach to distribute a homogeneous runtime environment to a heterogeneous and geographically distributed computing infrastructure. A uniform bootstrap environment is provided by a minimal virtual machine tailored to LHC applications. Based on a study of the characteristics of LHC experiment software, the thesis argues for the use of content-addressable storage and decentralized caching in order to distribute the experiment software. In order to utilize the technology at the required scale, new methods of pre-processing data into content-addressable storage are developed. A co-operative, decentralized memory cache is designed that is optimized for the high peer churn expected in future virtualized computing clusters. This is achieved using a combination of consistent hashing with global knowledge about the worker nodes' state.

The methods have been implemented in the form of a file system for software and Conditions Data delivery. The file system has been widely adopted by the LHC community and the benefits of the presented methods have been demonstrated in practice.

Pre-Publications

Parts of the thesis have been pre-published:

- **CernVM-FS: Delivering Scientific Software to Globally Distributed Computing Resources**
  J. Blomer, P. Buncic, and T. Fuhrmann
  To appear in Proc. of the 1st Workshop on Network-Aware Data Management held in conjunction with the IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11), Seattle, 2011
- **A practical approach to virtualiztion in HEP**
  P. Buncic, C. Aguado Sánchez, J. Blomer, A. Harutyunyan, and M. Mudrinic
  The European Physical Journal Plus, **126**(1), 2011
- **Distributing LHC Application Software and Conditions Databases using the CernVM File System**
  J. Blomer, C. Aguado Sánchez, P. Buncic, and A. Harutyunyan
  To appear in Proc. of the 18th Int. Conf. on Computing in High Energy Physics (CHEP'10), Taipei, 2010.
- **Studying ROOT I/O Performance with PROOF-Lite**
  C. Aguado Sánchez, J. Blomer, P. Buncic, I. Charalampidis, G. Ganis, M. Nabozny, and F. Rademakers
  To appear in Proc. of the 18th Int. Conf. on Computing in High Energy Physics (CHEP'10), Taipei, 2010.
- **A Fully Decentralized File System Cache for the CernVM-FS**
  J. Blomer and T. Fuhrmann
  Proc. of the 10th Int. Conf. on Computer and Communications Networks (ICCCN'10), Zürich, 2010
- **LHC Cloud Computing with CernVM**
  B. Segal, P. Buncic, D. Garcia Quintas, C. Aguado Sánchez, J. Blomer, P. Mato, A. Harutyunyan, J. Rantala, D. J. Weir, and Y. Yao
  Proceedings of Science, **ACAT**(004), 2010
- **CernVM: A Virtual Appliance for LHC Applications.**
  P. Buncic, C. Aguado-Sánchez, J. Blomer, L. Franco, A. Harutyunyan, P. Mato, and Y. Yao
  Journal of Physics: Conference Series, **219**, 2010

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Contribution of the Thesis . . . 4
- 1.3 Structure of the Thesis . . . 5

2 Terms and Definitions **7**
- 2.1 The Large Hadron Collider . . . 7
  - 2.1.1 Particle Accelerator . . . 8
  - 2.1.2 LHC Experiments . . . 9
- 2.2 Computing Model . . . 11
  - 2.2.1 The High Energy Physics Event . . . 12
  - 2.2.2 Online Computing . . . 13
  - 2.2.3 Offline Computing . . . 15
  - 2.2.4 Types of Input Files . . . 16
- 2.3 Distributed Computing Services . . . 17
  - 2.3.1 Grid Computing . . . 18
  - 2.3.2 The Worldwide LHC Computing Grid . . . 19
  - 2.3.3 Distributed and Decentralized Computing . . . 21

3 Worker Node Virtualization **23**
- 3.1 Motivation . . . 23
- 3.2 Cloud Computing . . . 24
  - 3.2.1 Virtualization . . . 25
  - 3.2.2 Types of Cloud . . . 26
- 3.3 Prospects . . . 26
  - 3.3.1 Optimized Resource Utilization . . . 26
  - 3.3.2 Portable Analysis and Development Environment . . . 27
  - 3.3.3 Volunteer Clouds . . . 27
  - 3.3.4 Long-Term Data Preservation . . . 28
- 3.4 Challenges . . . 28
  - 3.4.1 Performance . . . 28
  - 3.4.2 Unmanaged Resources . . . 33

3.4.3 Image Distribution . . . 34
3.4.4 Image Proliferation . . . 34
3.5 Volatility . . . 35
3.6 Software Distribution . . . 35

4 Software Characteristics 37
4.1 Building Blocks . . . 40
4.2 Generic Properties of Software Files . . . 41
4.3 Related Quantitative File System Studies . . . 42
4.4 Cumulative Size Distribution . . . 43
4.5 Compression Rate and Speed . . . 44
4.6 Access Pattern . . . 50
4.7 Borderline to Event Data and Conditions Data . . . 51
4.8 Software Distribution in WLCG . . . 53
4.9 Design Criteria . . . 55

5 Software Distribution 59
5.1 Caching and Replication . . . 59
5.2 Content-Addressable Storage . . . 60
5.2.1 Block Level and File Level CAS . . . 61
5.2.2 Key Space . . . 62
5.2.3 File Catalogs . . . 63
5.3 Pre-Fetching . . . 65
5.4 CAS Transformation . . . 66
5.4.1 Incremental Synchronization . . . 68
5.5 Confidentiality . . . 72
5.5.1 Model . . . 73
5.5.2 Confidential CAS . . . 73

6 Decentralized Memory Cache 75
6.1 Requirements . . . 76
6.2 Distributed Hash Tables . . . 77
6.2.1 Key Space . . . 77
6.2.2 Consistent Hashing . . . 77
6.3 Self-Organizing DHT Algorithm . . . 78
6.3.1 Load Balancing . . . 81
6.3.2 Simulation . . . 81
6.4 State Dissemination . . . 88
6.4.1 Slot state dissemination . . . 89
6.4.2 Distributed Watchdogs . . . 91

7 Performance Measurement and Comparison **95**
7.1 Design and Implementation of the CernVM-FS 95
7.1.1 Caching 95
7.1.2 File Catalogs 97
7.1.3 Data Access 97
7.1.4 Data Distribution 98
7.2 Evaluation of the Decentralized Memory Cache 99
7.3 Software Distribution Comparison 101
7.3.1 Turn-Around Time 103
7.3.2 Network Load 106
7.3.3 Runtime Penalty 107
7.3.4 Summary 108

8 Conclusion **111**

1 Introduction

Distributed computing in high energy physics benefits from its embarrassingly parallel workload. Data collected at particle colliders such as the *Large Hadron Collider* (LHC) [EB08] naturally consist of billions of small chunks of the order of only few megabytes or less. These chunks are packaged into files of the order 10^2 MB to 10^4 MB in size that can be analyzed independently of each other. In order to perform simulation and data analysis on these data sets, the LHC experiments have built *grid* infrastructures, which combine a large number of world-wide distributed computers into one virtual supercomputer [FKT01,B+05]. The currently used infrastructure processes tens of petabytes each year using of the order of 10^5 processing cores, most of them in computer centers in Europe, North America, and Asia.

In order to successfully perform computing jobs on the LHC Grid infrastructure, each computing job requires access to specific data sets and to a specific version of the analysis software. The analysis software, in turn, requires certain libraries and services of the grid middleware and the operating system. In other words, each computing job has explicit and implicit dependencies to data sets and to a runtime environment. The problem of bringing the job and its dependencies physically together on a single computer is referred to as *co-location* (Figure 1.1). Currently, the dependencies are distributed beforehand in an opportunistic way. The information about the physical location of such distributed data sets and software environments is stored in central monitoring systems and used by the central job scheduler to select suitable physical worker nodes.

The key to achieving decentralized data processing is to break these dependencies, i. e. to decouple computing jobs from the location of data sets and the runtime environment. Distribution of large data sets, in such a way that physically distributed clients have fast access to data, has been studied in the context of various distributed and decentralized file systems [KBC+00, DKK+01, GGL03, DEFH05, Kut08]. Common techniques used in such file systems are replication, caching, and peer-to-peer transport. As data sets in high energy physics turn out to have unforeseeable popularity, caching is particularly useful in comparison to structured replication. Furano reviews the research on the application of such file systems to data sets in high energy physics [Fur11].

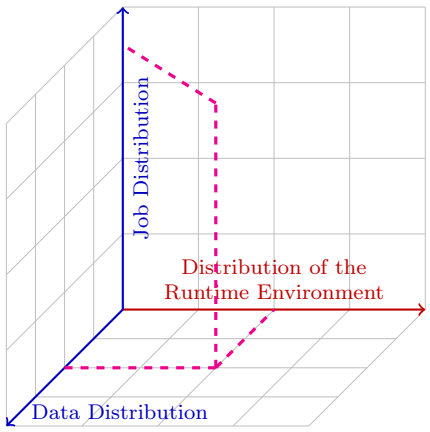


Figure 1.1: The 3 axes of co-location for LHC computing jobs. Towards decentralized computing, the distribution of computing jobs has to be independent from the distribution of data and the runtime environment. Here, we address the problem of distributing the runtime environment, thereby reducing the problem by one dimension.

Here, we will focus on the decoupling of the runtime environment from the distribution of jobs.

1.1 Motivation

The computing job description might be as simple as a couple of lines of code for invoking the analysis software with a set of parameters. The runtime environment required to successfully run such jobs, however, consists of the simulation and data analysis software, the grid middleware and — derived from these — a certain version of the operating system and its libraries. *Hardware virtualization* is a well-established technique used to break these dependencies; software together with the operating system and library dependencies is encapsulated in a virtual machine [SN05, Section 1.4]. Instead of the relatively static network of physical computing nodes, hardware virtualization facilitates volatility of computing resources. Computing jobs can be moved among physical computing nodes and they can be deployed on unmanaged computing resources such as provided by cloud infrastructures. This volatility brings benefits known from server consolidation, such as pausing virtual machines to temporarily free resources or moving virtual machines to less busy physical resources. Moreover, hardware virtualization provides a key building block for

the long-term preservation of experiment data and the ability to re-analyze them.

Unfortunately, due to its complexity, the problem of deployment and distribution of LHC experiment software is rather amplified by hardware virtualization. The LHC experiment software frameworks are jointly developed by thousands of physicists. They comprise of the order of 10^7 lines of code in 10^5 small files per release, typically organized in thousands of shared libraries that are searched and loaded at runtime. New physics understanding is extracted through the process of data analysis, which is in turn incorporated into the data processing applications. This workflow results in weekly or daily release cycles. New releases do not render previous ones obsolete, since published releases have to remain accessible for the sake of testing the reproducibility of physics results. Expanded to the expected life time of the LHC of at least 15 to 20 years, the overall number of files of the runtime environment sums up to the order of tens of terabytes and 10^9 files and directories. Moving and deploying such an environment in the form of a virtual machine hard disk image negates the potential benefits of hardware virtualization.

Instead of providing the runtime environment in relatively large images with a half-life of a few days, it is desirable to create a minimal bootstrap image and to store the rest of the runtime environment in the network [AFM05]. Approaches to solve the distribution of the software using general purpose distributed file systems, such as AFS, Coda, or Lustre, fail to scale in terms of number of files and number computing nodes and due to the volatility of resources. Distributed file systems for large-scale data sets are optimized for a ratio of data size to meta-data size that is many orders of magnitude larger than observed for software. Thus their performance is even worse than general purpose distributed file systems. In the context of grid computing, special purpose distribution systems for analysis software emerged [LHP+04, CGL+10]. However, such systems do not address the problem at the scale of LHC experiment software and neither do they address the volatility of virtualized resources.

Beck et al. propose the use of content-addressable storage as a key building block for globally scalable network storage [BMP02]. Meta-data can be stored in content-addressable files as well, having large directory trees partitioned into sub trees that are securely linked together using Merkle hashes [Mer06]. Based on a study of the characteristics of LHC experiment software, this thesis will introduce a new method to efficiently interface content-addressable storage with standard file systems. Furthermore, this thesis presents a new approach to co-operative caching tailored to the volatility of virtualized computing nodes.

1.2 Contribution of the Thesis

This thesis comprises the following results and new methods:

1. The thesis presents the first systematic study of the characteristics of LHC experiment software from the point of view of a file system. The study includes measurement results describing the cumulative size distribution, compression characteristics, growth rate, redundancy, and the access patterns. The results are compared to previous file system studies of typical UNIX and Windows workstation file systems. LHC data processing applications form a discrete, highly redundant, and meta-data intensive workload with an inhomogeneous access pattern (spike accesses) and strong demands for UNIX file system semantics. Based on the results, design criteria are defined for the efficient distribution of LHC experiment software.

2. In order to maintain a large number of files on file systems based on content-addressable storage, several engineering problems have been solved. The thesis proposes user-assisted cutting of the directory tree, which results in a better trade-off of meta-data file sizes and number of meta-data files as compared to currently used fixed partioning schemes. For a write pattern that publishes a set of changes to a file system as a new snapshot, the thesis proposes an incremental transformation of files into content-addressable storage. In contrast to existing approaches, the proposed transformation is based on a file system change set and thus avoids traversing the entire file system tree. The thesis proposes a new naming scheme for content-addressable storage that supports closed user groups through file encryption, whilst providing cachability and immediate revocation of read access.

3. The thesis proposes a new decentralized algorithm for co-operative caching on the cluster scale. In contrast to existing approaches using consistent hashing [KLL+97, GLS+04, ZH04] or "hint-based" peer selection [SH96, FCAB00], the thesis proposes an opportunistic dissemination of cache responsibilities in the complete cache space. The algorithm is designed as being inherently resilient to volatile computing resources since peer churn does not automatically result in rebalancing of cache contents. Instead, rebalancing of cache contents is the result of the computing nodes' sole decision to decrease their load by dropping cache content. Cache state and presence dissemination is performed by two customized low latency gossip protocols.

The algorithms and methods described in this thesis have been implemented and evaluated against real workload.

1.3 Structure of the Thesis

The thesis is structured as follows. Chapter 2 discusses the workflows and the terms and definitions of LHC computing tasks. We introduce the concept of grid computing and review centralized parts in the current computing model.

Chapter 3 discusses hardware virtualization as a key enabling technology for cloud computing. We review particular obstacles one has to overcome in order to harvest cloud computing resources and computing resources of volunteers for LHC computing. We report on performance studies of typical LHC workloads in virtualized environments.

Chapter 4 discusses various features of LHC experiment software. We analyze the static and dynamic characteristics of the LHC experiment software. We compare the characteristics of the software to related file system studies. We conclude by establishing design criteria for efficient software distribution.

Chapter 5 discusses the maintenance of file systems based on content-addressable storage. We identify several problems of existing technologies that prevent such file systems from being used at the scale of LHC experiment software. We discuss more efficient approaches to these problems.

Chapter 6 presents the design of a fully decentralized file system cache. We develop two low latency state dissemination protocols for the distribution of cache states and presence states of the worker nodes. We evaluate the algorithm using simulation based on file system traces.

Chapter 7 reviews implementation details of the proposed methods. We evaluate an implementation of the decentralized memory cache developed in Chapter 6. We define performance metrics for software distribution and compare the proposed methods with existing technologies.

Chapter 8 concludes the thesis and summarizes its results.

2 Terms and Definitions

This chapter provides an overview of computing in high energy physics with respect to the LHC experiments. The characteristics of computing are derived from its application as a critical part of the experiment designs and the data analysis. Section 2.1 briefly introduces the setup of the LHC experiments, their organizational structure, and the scale of the produced data. Sections 2.2 and 2.3 define commonly used terms and definitions in LHC computing and review the overall computing workflows as well as the distributed computing infrastructure.

2.1 The Large Hadron Collider

The Large Hadron Collider (LHC) and its attached particle detectors are large scientific machines used to study fundamental constituents of nature. The LHC collides particles in order to study their emerging products, the so-called *secondary particles*, that are registered by detectors. The large size of the LHC is due to the very high energy it has to give to particles before colliding them. With higher collision energies, more secondary particles are produced that must be detected and smaller structures can be probed.

When particles with very high energies collide, the temperature and density during the collision is comparable to the environment that existed during the very early state of the universe as described by the Big Bang theory. The LHC is designed to create an environment that corresponds to a temperature of 10^{19} K, which is comparable to 10^{-25} seconds after the Big Bang.

The LHC allows the predictions of the so-called Standard Model to be probed at very high energies. The Standard Model is a physics model that describes the fundamental particles and their interaction via the strong force, the weak force, and the electromagnetic force [Per00]. Its predictions match the results obtained by existing experiments very accurately. Still, the Standard Model does not combine all four fundamental forces (the three aforementioned plus gravity) into a single theory. Also, the Higgs particle, which is a necessary building block of the Standard Model in order to explain the origin of mass [Hig64], has not yet been discovered. At the macroscopic scale several phenomena remain

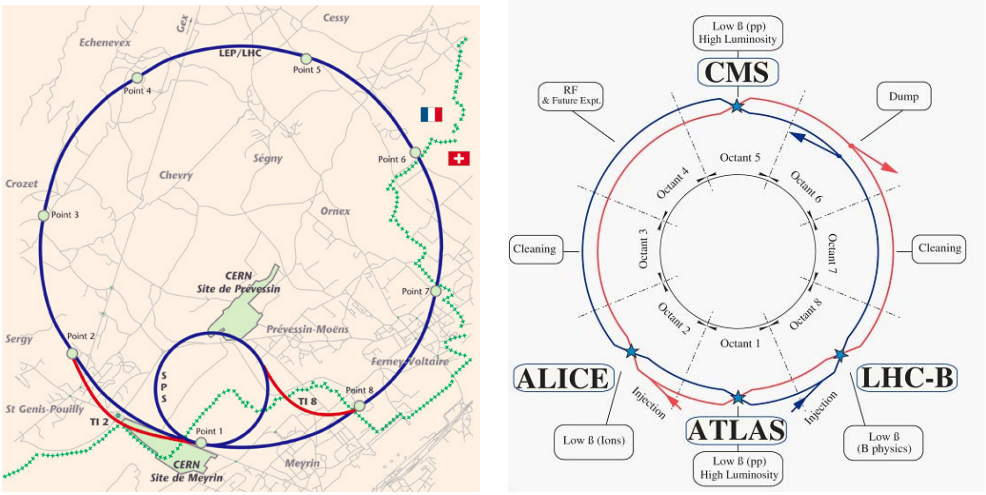


Figure 2.1: Left hand side: the perimeter of the Large Hadron Collider (LHC) and the transfer lines to the *SPS* pre-accelerator embedded in a map of the area (source: L. Guiraud, CERN public picture collection). Right hand side: a schematic image of the LHC as presented by Evans and Bryant [EB08].

unexplained, such as the nature of the so-called dark matter and dark energy, or the asymmetry between matter and antimatter. Beyond the Standard Model, the LHC experiments will probe, for instance, the supersymmetry theory [Per00].

2.1.1 Particle Accelerator

The LHC is a synchrotron, i. e. a circular particle accelerator, operated by the *European Laboratory for Particle Research* (CERN[1]) [EB08]. The LHC is reusing the ring tunnel of the former LEP accelerator. The tunnel has a circumference of 26.7 km (Figure 2.1). It is beneath the franco-swiss border near Geneva, at a depth underground ranging from 45 m (East) to 170 m (West). The planning started in 1984 and the LHC construction was approved in 1994.

The LHC accelerates two *beams* of either protons[2] or lead ions that counter-rotate in beam pipes next to each other through the ring. Each beam consists of up to 2808 *bunches*, packets of particles that are depending on their position

[1] The acronym CERN is derived from the laboratory's provisional name "Conseil Européen pour la Recherche Nucléaire"

[2] Protons belong to a class of particles called "hadrons". In contrast to "leptons" such as electrons, hadrons are heavy particles. A proton, for instance, has 2000 times the mass of an electron.

in the ring from a couple of centimeters to the order of micrometers in size. Each bunch consists of up to 10^{11} protons or up to $7 \cdot 10^7$ lead ions. The particle beams are accelerated, bent, and focused by strong magnetic fields. Starting with bunches injected from pre-accelerators, the LHC takes some 20 minutes to accelerate the beams to their final energy of up to 7 TeV (1.12 µJ) per proton. As the velocity v of a particle comes close to the speed of light c, its kinetic energy results in a substantial increase of its mass according to the formula $m(v) = m_0/\sqrt{1-(v/c)^2}$, with m_0 being the rest mass of the particle. At 7 TeV, a single proton is as heavy as a fly.

The LHC is at the moment the world's largest and most powerful particle accelerator. The LHC is designed to produce particle collisions with more than an order of magnitude higher energy than the previously most powerful particle accelerator, the Tevatron near Chicago [Edw85]. The operation of the LHC consumes about the same amount of power as the state of Geneva (≈120 MW). In comparison, the second largest energy consumer at CERN is the computing center with 5 MW. As CERN contributes around 20 % of the overall LHC computing power (cf. Section 2.3.2), we estimate the overall energy consumption for LHC computing to be ≈25 MW. That is a conservative estimation as it neglects the economy of scale of large computing centers.

At four *interaction points* around the LHC tunnel, the particle beams are brought into collision. The collision rate is up to 40 MHz, i. e. bunches cross each other every 25 nanoseconds. On average, some 20 particles from both crossing bunches collide at full design performance. Hence, beams might circulate up to 10 hours before new beams have to be injected. The collision energy is the sum of the respective beam energies. For protons, the collision energy is up to 14 TeV. For lead ions, the collision energy is up to 5.5 TeV per nucleon, i. e. a total collision energy of 1.15 PeV. During a collision, the particles' kinetic energy can be transformed into matter, according to the formula $E = mc^2$. In this process, a collision might create a variety of different secondary particles, with creation likelihoods depending on the particle. Such created particles might also decay into other particles. It is the purpose of the LHC experiments to capture, reconstruct, and analyze these collision products.

2.1.2 LHC Experiments

The LHC experiments are located at the four interaction points. Though strongly linked with CERN and the LHC, the experiments are not solely operated by CERN. The teams that build and operate the experiments are independent international collaborations, comprising hundreds of research institutes and thousands of physicists from all over the world.

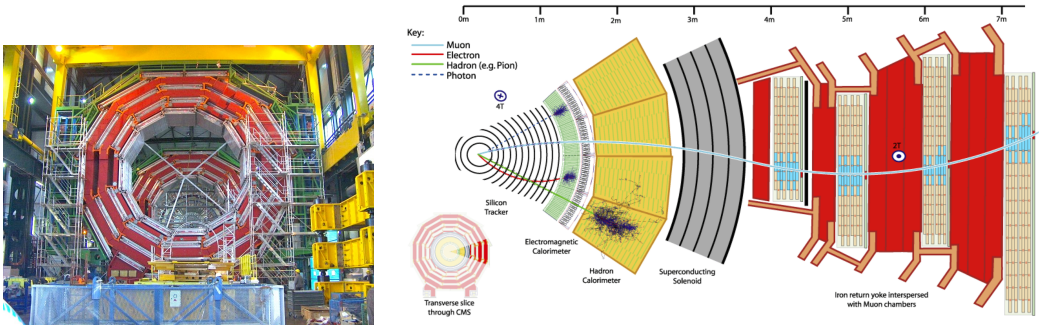


Figure 2.2: Left hand side: The profile of the CMS detector before final assembly (own photography). Right hand side: a schematic profile of the CMS detector (source: CMS public web site).

Each LHC experiment collaboration operates a *particle detector*. The particle detectors are installed in underground caverns at, or close to, the interaction points. They are intended to identify the secondary particles created in a particle collision and to measure their energy, direction of flight, charge, and momentum. They do so by combining signals from layered sub detectors. Such a sub detector might, for instance, expose particles to a magnetic field and measure their bending behavior in order to determine the particle momentum (Figure 2.2). Overall, a particle detector comprises up to a hundred million sensors. Extracting physics information from these sensor signals is called *reconstruction*.

Most particle detectors at the LHC are large constructions: the ATLAS detector, for instance, has a total weight of 7000 t and is half as big as the Notre Dame cathedral. The size of the detectors is required by the high energy of the secondary particles and the desired accuracy of the measurements; the larger the detector the stronger is the measurable influence of the detector parts (magnets, matter, etc.) on the secondary particles. In spite of their size, the detectors are delicate machines. The geometry of the construction as well as environment conditions such as temperature or gas pressure are subject to constant monitoring and calibration. The position of the sub detectors, for instance, requires calibration at the scale of micrometers. Still, the detectors are not entirely accurate. There is inevitably limited detector efficiency, blind spots, de-functional detector parts, and so on. A (considerable) part of the data analysis consists of *efficiency corrections*, i. e. estimating the real physics processes from the detector output.

There are seven experiments at the LHC sharing the four interaction points.

	Detector		Collaboration		
Experiment	Dimensions [m]	Sensors	Scientists	Labs	Countries
ATLAS	$46 \times 25 \times 25$	100 M	3000	173	38
CMS	$21 \times 15 \times 15$	100 M	3000	180	38
ALICE	$26 \times 16 \times 16$	18 M	1000	105	30
LHCb	$21 \times 10 \times 13$	1 M	770	55	15
TOTEM	$440 \times 5 \times 5$	230 K	65	9	7
LHCf	$2 \cdot 0.3 \times 0.1 \times 0.1$	< 10 K	21	10	4
MoEDAL	$\sqrt{25} \times \sqrt{25} \times 0.02$	passive	25	9	7

Table 2.1: Key figures of the LHC experiments. The experiment collaborations are distributed world-wide over many small groups and institutes.

Table 2.1 shows some of their key figures. The ATLAS [The08b] detector and the *Compact Muon Solenoid* (CMS) [The08c] are general purpose detectors. Having two detectors allows for cross-verification of discoveries. *A Large Ion Collider Experiment* (ALICE) [The08a] is a detector specialized to analyze lead-lead collisions. An important focus of ALICE's research is on the so-called quark-gluon plasma, a very hot and very dense state of matter that is assumed to have existed shortly after the Big Bang. *The Large Hadron Collider beauty* experiment (LHCb) [The08d] is a detector specialized to analyze the asymmetry between matter and antimatter. The aforementioned experiments are considered the four large LHC experiments.

The total elastic and diffractive cross section measurement (TOTEM) [The08f] experiment is a detector specialized for high-precision measurements of the in-depth structure of the proton. *The Large Hadron Collider forward experiment* (LHCf) [The08e] consists of two detectors specialized to compare particles produced in an controlled environment to particles from cosmic rays. *The Monopole and Exotics Detector At the LHC* (MoEDAL) [The09b] is a detector specialized to search for magnetic monopoles and large supersymmetric particles[3]. In the rest of the thesis, we will focus on the four large LHC experiments.

2.2 Computing Model

The following sections describe the terms and definitions used in computing for LHC experiments. The central entity in LHC computing is the *event*. The event

[3]At the time of writing, the MoEDAL experiment is approved but not yet in operation.

contains information about the secondary particles that have been created as a result of colliding primary particles from crossing beams.

The complexity of the LHC data analysis grows at least quadratically with time. The entire data set is reprocessed several times per year, in order to take into account improvements in the understanding of the detector performance (the so-called calibration and alignment of the detector) and to include improvements in the reconstruction and physics analysis algorithms.

2.2.1 The High Energy Physics Event

From the detector point of view, a particle collision results in a set of sensor signals. Signals typically result from responses of the detector elements (such as a silicon sensor) to the ionization produced by a particle traversing the sensitive element of the detector. The sensor signals corresponding to a single crossing of the two beams are digitized and recorded as a *raw event*. The size of a raw event is of the order of 10^3 kB to 10^4 kB. The *reconstruction* step that follows extracts the physics information from the set of digitized signals. In a first step, the fuzzy signals have to be transformed into global time and space coordinates, taking the detector geometry and alignment into account. Afterwards, information of many sub detectors is combined to reconstruct the trajectories of the secondary particles and to determine their properties, such as charge and momentum. The output of the reconstruction step is called *event summary data* (ESD). It has a size of the order of 10^2 kB, an order of magnitude smaller than the raw event. The structure of the physics objects can be represented as a tree, with the event as root node and the properties of the physics objects as leaf nodes. The ESDs might be further refined by stripping unnecessary information not required by an analysis. Such a stripped ESD is called *analysis object data* (AOD). There might be various AODs for a single ESD, each of them tailored to a particular type of data analysis. Figure 2.3 shows a visualization of a single simulated and reconstructed event in the CMS detector.

The analysis of events uses descriptive statistics on a large number of events to study certain physics objectives. The total size of the collected data set typically requires several 10^9 events. However, a single analysis from an individual scientist might involve only 10^7 events. Events can be analyzed independently from each other. LHC analysis tasks benefit from *event level parallelism*, they operate on an *embarrassingly parallel* workload. Hence, LHC analysis computing is not classical high performance computing but in fact it is *high throughput computing* [LRTB97]. The crucial performance figure is the

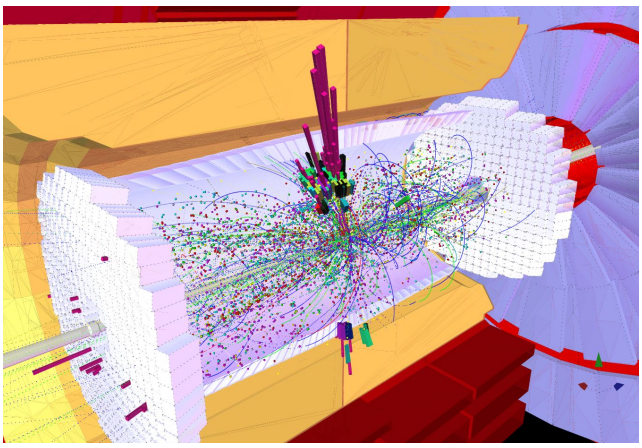


Figure 2.3: CMS event display of a simulated and reconstructed event. LHC applications iterate over a large number of such events. Whilst the secondary particles and their trajectories through the detector change with every event, the description of the detector stays constant. (Source: I. Osborne, public picture collection of the CMS collaboration)

number of processed events per second. Figure 2.4 visualizes the processing steps of events.

2.2.2 Online Computing

Online computing deals with the tasks that have to be performed in real time while the detector is capturing events. At full design performance, the LHC produces up to 600 million particle collisions per second (2808 bunches $\times$ 11245 turns per second $\times$ 20 collisions per crossing particle bunches). Assuming a raw event size of 1 MB this would result in a required data taking rate of 570 TB/s. The LHC experiments filter the raw events and discard the majority of "uninteresting" events according to simple criteria such as the energy of the secondary particles. This is done in real time by parallel, multi-level *trigger farms*. The trigger farms can reduce the number of events per second from 40 MHz to 100 Hz. The trigger farms are partly implemented using customized hardware. The triggers are tunable in order to reflect certain physics objectives.

The *data acquisition* stores the filtered raw event data to tapes at CERN. The data taking rate depends on the experiment, whether protons or lead ions collide, and on the capabilities of the storage system. At the end of 2010, the

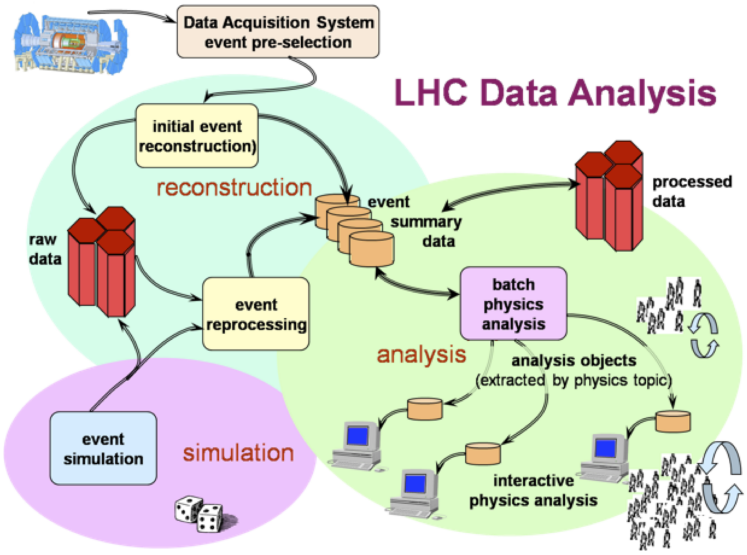


Figure 2.4: The processing steps of an event as presented by Harvey et al. [HMR09]. Simulation and reconstruction are steered by collaboration bodies but might be performed in the distributed computing environment of the Worldwide LHC Computing Grid. The reconstruction process is subject to re-processings that use up-to-date descriptions of the latest detector performance. Analysis is performed de-centrally by individual physicists and research groups.

four large LHC experiments combined recorded on average 2.5 GB/s with peaks at 7 GB/s [Bir10].

2.2.3 Offline Computing

Offline computing comprises the computing tasks not directly connected to the detector operation. The tasks of offline computing include the development of the software for reconstruction, analysis, simulation, and the management of the distributed computing. The reconstruction tasks and simulation tasks are organized by the respective collaboration bodies. The data analysis is driven by individual scientists and research groups. As such, analysis tasks are performed decentrally.

Reconstruction

Event reconstruction produces ESDs from raw events. It has to interpret the digitized detector signals and identify the secondary particles and their *tracks*, i. e. their trajectory through the detector. Furthermore, reconstruction determines the point of collision as well as several properties of the particles, such as charge and momentum. The reconstruction is not entirely accurate, its result is an approximation together with estimators for the statistical uncertainty.

Reconstruction is performed in several passes with a feed-back loop (Figure 2.4). With the output of a reconstruction pass, the detector experts tune the parameters of the reconstruction algorithms for better accuracy. For ATLAS, for instance, 10 % of data are immediately reconstructed, then further adjustments are done before all of the raw data are reconstructed. Usually, reconstructed data are available within 48 hours of the data having been taken. The feedback loop might involve changes and corrections to the reconstruction software and requires a fast distribution of the software to worker nodes.

A second feed-back loop involves data analysis, which provides better understanding of the detector. In turn, the improved knowledge of the detector helps to improve the reconstruction algorithms.

Event Simulation

Event simulation is an indispensable tool when dealing with machines as complex as the LHC or one of its detectors. During the design phase, simulations measure the detector's effectiveness and performance. Here, they are also used to produce large data samples that are used to prepare the (distributed) computing tools

before real data taking starts. During data taking and analysis, simulations of physics processes and the comparison with the detector output provide a means to determine the detector corrections that need to be applied to account for inefficiencies in the detector performance. Vice versa, the physics models used in the simulation toolkits are constantly improved by validating them using new data in new energy ranges and by incorporating newly discovered phenomena.

LHC computing uses discrete event simulators and Monte-Carlo algorithms that reflect the probability of the simulated physics processes. So-called *event generators* simulate the creation of secondary particles produced by a particle collision. The *transport simulators* simulate the particles traveling through the detector and their interactions with the detector materials. Obviously, transport simulators require an accurate geometric and physical description of the detectors, that includes details of cabling and support structures. As opposed to reconstruction and analysis, simulation is a CPU-bound task.

Analysis

During data analysis, a sample of "interesting" events is processed using methods of descriptive statistics. The signature that classifies an event as being interesting depends on the particular physics objectives of the analysis. For example, this might involve selecting all those events in which at least one muon has been identified. Data analyses might comprise multiple pipelined phases, in which event data output from one phase is temporarily stored and used as input for another phase. The final output of the data analysis is most often a histogram.

2.2.4 Types of Input Files

Except for event generation, the computing workflows operate on large amounts of input data. We classify the different types of input files that are required to execute LHC computing jobs.

Event Data The event data consists of collections of events. There are usually packaged in files of the size of 10^2 MB to 10^4 MB. The size of the event data files is of the order of 10^3 TB per year and experiment.

Conditions Data In order to extract the physics processes from event data, it has to be accompanied by the so-called *conditions data*. Conditions data contain the detector and environment conditions at the time of data taking,

such as detector position calibration, environment temperature, gas pressure, and so on. Conditions data might be enriched after reconstruction, for instance by data quality information. The size of the conditions data files is of the order of 10^3 GB per year.

Static Data Static data includes the description of the detector geometry and magnetic field together with quasi-constant information about physics processes, such as physics constants, probability distribution tables, and particle property databases such as decay probabilities. Static data are usually delivered together with software. The size of the static data files is of the order of 10^3 MB.

Software The software required to perform analysis includes the experiment specific analysis framework, common data analysis and simulation tools for HEP, and the user's analysis code. The external tools and libraries used by these frameworks are often shipped with the core software. Furthermore, the software comprises grid middleware and experiment specific tools for distributed computing and data access. Because of the feedback loops with physics results and because of the distributed development of the data analysis algorithms, the software is subject to perpetual changes that are reflected by daily or weekly release cycles. The size of the software files is of the order of 10^2 GB to 10^3 GB per year.

2.3 Distributed Computing Services

The computing infrastructure limits the rate at which LHC events can be analyzed. As mentioned before, LHC data processing benefits from the event level parallelism of the workload. Instead of elaborating parallel algorithms, sequential algorithms can be applied to a large number of events in parallel on a distributed computing infrastructure. In the past, the computing infrastructure changed two times fundamentally, from dedicated mainframes to PC clusters in the 1990's [BBC+91] and from clusters to the grid infrastructure starting from 2000. Currently, it is on the threshold to change again in order to benefit from *unmanaged* computing resources offered by various cloud infrastructures.

The shift to cluster computing was driven by a better price/performance ratio. The change to grid computing was caused by the fact that at the time of designing the computing services for the LHC experiments no single data center was powerful enough to process the data produced by the LHC. However, the members of the LHC experiment collaboration have their own smaller and larger computing premises. The LHC Grid is used to combine all these

resources into a uniform computing service. Ideally, the LHC Grid enables scientists to run analysis jobs on the globally distributed infrastructure the same way they do on their local cluster; the grid takes care of finding the physical location of the required data, finding free resources, running the job, and writing the output to a location accessible to the scientist.

2.3.1 Grid Computing

"The Grid" has been proposed by Foster et al. as a means to share resources "among dynamic collections of individuals, institutions, and resources—what we refer to as virtual organizations" [FKT01]. Foster later refined the definition of the Grid as a "system that coordinates resources that are not subject to centralized control using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of service" [Fos02]. This system effectively is a middleware layer that is supposed to shield applications from the heterogeneity and complexity of the underlying distributed resources. As such, the grid has been inspired by the idea of "utility computing", i. e. "computing" should by subject to the same standardization and ubiquitousness as for instance has happened to electrical power by power grids. The use of the grid was not meant to be bound to scientific collaborations, although LHC data processing was one of the grid drivers specifically mentioned by Foster.

Based on the definition of grid protocols and the overall system architecture, two major grid middleware implementations have been developed, Globus [FK97] and gLite [LHP+04]. The services provided by gLite are shown in Figure 2.5.

Access Services Provide the interface (API) to the grid users.

Security Services Used to grant or deny resource access to the users of a *virtual organization* (VO). Furthermore, the site proxy allows for controlling network traffic related to grid resources.

Information and Monitoring Services Used to supervise the status of resources and jobs and can in turn be used by other services to make decisions (such as: "which resources are free for the next job?").

Job Management Services Used for job scheduling, matching a task queue with the available resources and the user's quota.

Data Services Services for file level access to data and writable output storage. Files are addressed using logical file names in a universal namespace and

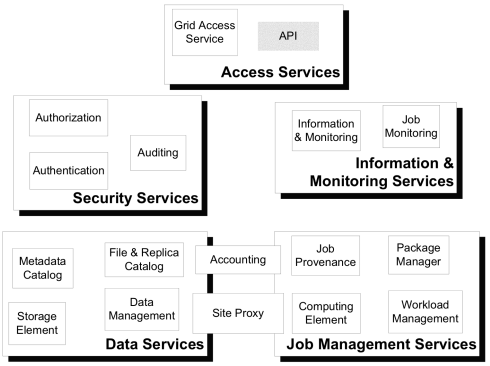


Figure 2.5: Web services provided by gLite as presented by Laure et al. [LHP+04]

can be replicated to multiple physical locations. The translation from logical to physical file names is done by file catalogs.

Obviously, the computing jobs do not only require data (provided by the data services), but also the experiment software. In gLite, the *package manager* is used to deliver software to the worker nodes. The gLite package manger steers standard package managers such as RPM in order to install software on the worker node or on a shared software area. Essentially, it tackles the problem of software installation using the existing grid services in the form of an *installation job* having a software release package as "data file".

2.3.2 The Worldwide LHC Computing Grid

The Worldwide LHC Computing Grid (WLCG) is the principal distributed computing infrastructure used for LHC data processing [B+05]. It is the largest grid infrastructure, comprising some 250 000 cores globally distributed over more than 140 registered computing sites and many more small institutional computing resources of individual research groups. The site size varies from just a couple of cores to many thousands of cores. By the end of 2010, WLCG executed some 1 million jobs per day [Bir10].

WLCG uses a fair share of resources provided by various regional, national, and international grid initiatives, most notably the infrastructures of the *Open Science Grid* (OSG)[4] in North America and of the European *Enabling Grids for E-SciencE* (EGEE)[5]. The share of resources given by WLCG to

[4] `http://www.opensciencegrid.org`
[5] `http://www.eu-egee.org`

the experiments is determined by memoranda of understanding signed by the participating computing centers, turning WLCG itself into an international collaboration (Figure 2.6).

In contrast to the somewhat computing-centric original grid architecture, one of the biggest challenges for LHC data processing is the data management of up to 15 PB new event data per year. As opposed to the loosely coupled Grid, WLCG defines a tightly coupled alliance of powerful computing centers collectively taking care of data management. Among these data centers, each data set is replicated three times resulting in a total volume of up to 45 PB per year. Based on the assumption that the standard Internet network links are too slow and fragile to cope with the event data, the LHC Optical Private Network (LHCOPN) [BMM05] was built. The LHCOPN connects CERN with 11 globally distributed computing centers via 10 Gbit/s links dedicated to event data transfer. The LHCOPN is embedded into the MONARC architecture [MON00] that defines a hierarchically layered structure of the WLCG computing centers. The computing centers are classified as Tier 0 to Tier 3, depending on their size and quality of service:

Tier 0 The CERN computing center. It stores the master copy of all data and provides some 15 % to 20 % of the overall computing capacity.

Tier 1 There are 11 Tier 1 centers, located in Europe, North America, and Taiwan. The Tier 0 and the Tier 1's form the LHCOPN. Tier 1's provide enough storage to contribute a significant fraction of long-term storage for event data. Furthermore, they provide fast network links, significant computing capacity and round-the-clock support. Tier 0 and the Tier 1's provide around half of the overall computing capacity.

Tier 2 There are some 120 Tier 2 centers in WLCG. Usually, they provide full grid services and a data cache and channel their data traffic through the nearest Tier 1. They do not provide archival storage for the event data. They do not guarantee 27/7 operations support.

Tier 3 Tier 3 centers are usually small computing facilities private to individual research institutes. They are used for analysis tasks. They can be as lightweight as a couple of cores in two low-end servers. They do not necessarily provide grid services nor are they necessarily part of WLCG monitoring. Nevertheless, they require *access* to data and analysis software.

Due to a lack of generally accepted grid standards, WLCG defines a set of standard services itself. Moreover, experiments developed their own, inde-

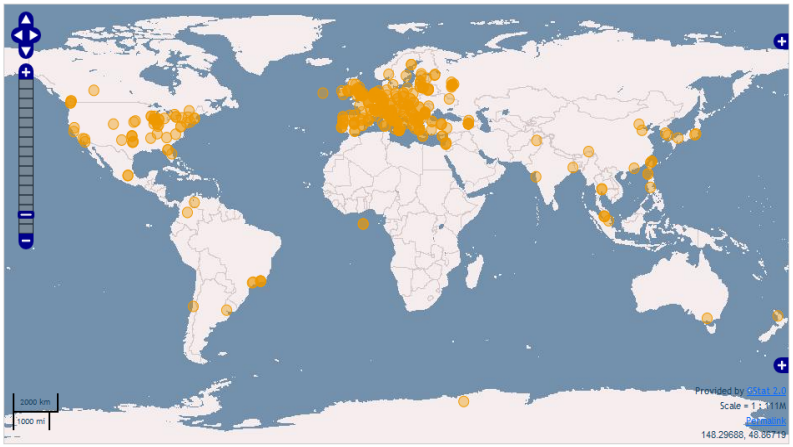


Figure 2.6: The location of the WLCG computing centers (source: `http://gstat-prod.cern.ch/gstat/geo/openlayers`).

pendent distributed computing frameworks [SAB+03, TBB+08, Mae08]. These independent frameworks have been included into the grid infrastructure by so called *pilot jobs*. Pilot jobs are essentially placeholder grid jobs that, when arriving on a worker node, inspect the environment and connect to the experiment's *task queue*. Thereby experiments implement their own pull-based job scheduling in contrast to the grid push model that is agnostic to the specific experiment job constraints.

2.3.3 Distributed and Decentralized Computing

While the infrastructure for LHC data processing is distributed, it is not necessarily fully decentralized. We will review centralized parts in computing for WLCG.

The grid installation jobs used for software deployment enforce a globally consistent set of shared software areas. In the following chapters, we will study means to decentralize software deployment.

Following the hierarchical Tier model, event data are staged to predefined locations according to a central planning process. The computing is data driven with the data spread around the LHC Grid first and computing jobs moved to the data later. As the scale of the distributed analysis grows, deterministic access patterns are replaced by unpredictable and random access patterns. From today's perspective, the need for a hierarchical data transfer path is fading away due to the rapidly increasing transfer rates of the normal Internet

links. However, because of the WLCG's geographic distribution (Figure 2.6), random site-to-site traffic has a high bandwidth-delay product, i. e. network connections having a high throughput and high latency at the same time. Due to organizational policies, many sites are behind firewalls and/or NAT layers that prevent incoming traffic and restrict outgoing traffic to use standard protocols. Each WLCG site allows high speed point-to-point connections between their worker nodes.

The operation of the LHC Grid depends on middleware services, which are provided by the sites. The operation of the middleware services has to be centrally coordinated and limits the execution of computing jobs to the managed resources provided by WLCG. The virtualization of worker nodes is required in order to harness opportunistic cloud resources and in order to simplify the grid site operation.

3 Worker Node Virtualization

In this chapter, we will discuss the prospects and challenges of running worker nodes in virtual machines. In contrast to the complex grid middleware interface, the interface used to control virtual machines is rather simple. Four commands are sufficient: start, terminate, list virtual machine images, and list running virtual machines. Virtual machines allow for separation of resource provision and service provision. They provide additional flexibility such as suspend-resume or migration of virtual worker nodes from a physical machine to another, possibly remote machine. Thus, virtual machines are often an unmanaged and volatile resource.

3.1 Motivation

In the current model of WLCG computing, the provision of the resources and the provision of the services for distributed computing are coupled. The site administrators and the experiment computing teams need to work closely together. Some motivating examples:

- The worker node needs to run the experiment software. The experiment software releases are installed on a shared software area on the grid sites. To that end, the site administrators maintain dedicated accounts for the experiment's software manager.
- The experiment's software stack requires a specific operating system. As a result, worker nodes are often dedicated to a specific experiment. For optimal resource utilization, however, grid worker nodes should support a variety of different experiments and virtual organizations.
- In order to run jobs, worker nodes require access to various grid services (monitoring, task queue, etc.). As the large experiments developed customized distributed computing services, these experiment specific services have to be available, too. Such services are installed and maintained by the WLCG site administrators.
- The sites need to install and maintain the experiment's toolkits for the access to event data and conditions data and the output storage.

- For small collaborations, it is challenging to use the grid infrastructure. Not only do they need to port their applications to grid middleware, they also need to convince site administrators to provide their specific services and requirements.

Worker node virtualization offers an opportunity to decouple infrastructure, operating system and experiment software life cycles as well as the responsibilities for maintaining these components. Virtualization allows for the different parts to evolve independently. The site operators have the responsibility to run virtual machines with minimal local contextualization, the experiment collaborators maintain the necessary libraries and tools, and the end user receives a uniform and portable environment for developing and running experiment analysis software on both single desktops and laptops as well as batch nodes in computer centers and computing clouds. Provided a basic interface to deploy and terminate virtual machines, experiments can extend their regular resources by opportunistic resources such as offered by commercial cloud providers.

3.2 Cloud Computing

Whilst originally part of the grid movement, the industry quickly diverged from the scientific community and propagated cloud computing as the preferred flavor of utility computing. The definition of the term "cloud computing" has been somewhat murky [VRMCL08] and only recently converged into a NIST standard [HLST11]. The NIST standard describes the defining properties of cloud computing by its resource utilization [SKH10]:

- Resources are used in an on-demand self-service way
- Resources are pooled by a cloud provider and shared amongst the cloud provider's users
- Resources are accessible by a broad collection of devices
- Resources are subject to rapid elasticity from the point of view of the users
- Resource utilization is measured, usually in order to allow for a pay-per-use model

In contrast to the grid, cloud computing is not necessarily about sharing of heterogenous, distributed resources but about the *provision* of *virtualized* resources. Internally, a cloud provider may or may not distribute its resources.

3.2.1 Virtualization

Virtualization is the key enabling technology behind cloud computing. Virtualization has been exploited on many different levels, such as multi-programming, virtual memory, the Java virtual machine, virtual function calls, or hardware virtualization. A virtualization layer provides a virtual view on the underlying resource that is tailored to the problem at hand. The virtualization layer translates between the virtual view and the "real" resource. This additional layer of indirection inevitably comes with a performance drawback, which might, or might not, be negligible.

Smith and Nair point out that virtualization is a concept very close to abstraction [SN05, p. 4]. They argue that virtualization does not necessarily hide the "implementation details". For instance, the virtual hardware provided by a virtual machine may or may not be more complex than the underlying physical hardware; in some cases, it can even be a pass-through, i. e. the virtualization layer provides the very same underlying physical hardware to its virtual machines. Similarly, the virtualized resource may be larger than the physical resource, in which case the virtualization layer *over-commits*.

In a broader view, however, virtualization does reduce complexity. A virtualization layer provides *uniform* access to resources. Considering the example above, let n the number of operating systems and m the number of hardware platforms. Without virtualization, there are $n \cdot m$ combinations of operating systems and hardware platforms. Using virtualization, all operating systems have to run on the uniform virtualization layer whereas the virtualization layer has to run on all hardware platforms. Thus the complexity is reduced to $n + m$ combinations.

Hardware Virtualization

Hardware virtualization is defined as a virtualization layer working on the instruction set architecture (ISA) level. This kind of virtualization layer is also called *Virtual Machine Monitor* (VMM) or *hypervisor*. The virtual machine monitor provides the instruction set of a virtual CPU, including instructions to access memory and I/O instructions. In addition, the VMM provides virtual I/O devices, such as network interfaces, hard disks, or timers. The VMM provides an interface for an operating system to run on. Such a system running on a VMM is called a *guest* or *virtual machine*. The system on which the VMM runs is called a *host*. Often, the virtual hard disk of the virtual machine is stored on the host in the form of a single large file, the *hard disk image*.

3.2.2 Types of Cloud

Not all cloud resources are usable in the context of LHC computing. Depending on the provided interface, we distinguish three common types of cloud:

***Software as a Service* (SaaS)** The resources to run a specific application are hosted at the cloud provider. The cloud providers usually provide a web interface to the users. This idea was previously known as *Application Service Providing* (ASP). An example of SaaS is Google Docs.

***Platform as a Service* (PaaS)** Instead of providing a specific application, with PaaS the cloud provider offers a middleware to its resources. PaaS users can develop their own applications against such a middleware. An example of PaaS is Microsoft Azure.

***Infrastructure as a Service* (IaaS)** The resource interface of IaaS is basically a simple API to deploy, start, and stop the user's virtual machines. An example of IaaS is Amazon EC2.

Since the HEP community has developed very specific software required for LHC data processing, IaaS is the only type of cloud of further interest. From the view point of control and access, we distinguish between, on the one hand, *private* and *community clouds*, i. e. the resources are under the premises and for use of a single organization or a collaboration, and, on the other hand, *public clouds*, i. e. the resources are available for a fee to the general public (Figure 3.1). Furthermore, there is an emerging type of cloud called a *volunteer cloud*; here, a potentially large number of volunteers is willing to spend spare cycles of their home PCs for supporting scientific projects. The predominant infrastructure for volunteer computing is BOINC [And04].

3.3 Prospects

In the context of LHC computing, virtual machines offer notable benefits beyond simplified resource access. The following use cases can be envisaged:

3.3.1 Optimized Resource Utilization

For experiment collaborations, moving from one major version of the operating system to the next is a major and lengthy process. As a consequence, moving a part of the batch capacity to a new OS flavor often leads to wasted capacity as users are never fully ready to move to the new platform. By virtualizing the

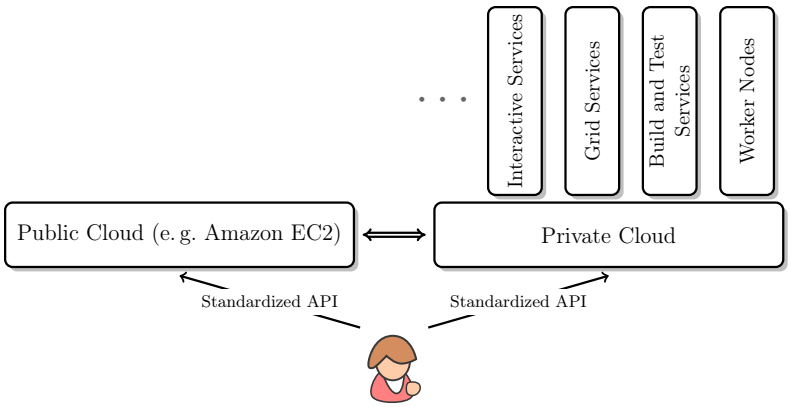


Figure 3.1: Various services required for LHC computing deployed as virtual machines in an IaaS data center. These services can be migrated to public clouds under peak load. This is an idealized picture; in practice, such a fully virtualized computing center is not yet present in WLCG.

batch environment one can dynamically provision standard batch nodes of a given OS flavor according to demand [CGM+10]. With such an approach, a more efficient utilization of computing resources can be achieved by dynamically consolidating idle virtual worker nodes on a single physical machine and by powering-off unused physical machines.

3.3.2 Portable Analysis and Development Environment

Given the variety of the operating systems that physicists run on their laptops and desktops (Windows, Mac OS X, Linux) and the fact that most computing capacity provided by the grid tends to be locked to a single version of the Linux operating system for considerable time, it quickly becomes impractical and often impossible to install, develop and debug locally the experiment software before eventually submitting jobs to the grid. This is evidently a use case where virtualization technology is beneficial. A Virtual Machine that runs the Linux operating system compatible with one available on the grid provides a build and test environment independent of underlying hardware or software platform.

3.3.3 Volunteer Clouds

Providing LHC applications in the framework of volunteer computing projects allows for an almost free extension of the existing computing resources. Fur-

thermore, volunteer computing projects have a public relations component attached to it and some outreach is gained for the scientific project. Volunteers run a variety of operating systems, including Windows, Mac OS X, and various flavors of Linux. By encapsulating the applications into virtual machines, the majority of desktop computers can be harnessed without the need to port millions of lines of code. A "VM Wrapper" that allows a BOINC job to be a virtual machine has been presented by Segal et al. [SBQ+10].

3.3.4 Long-Term Data Preservation

After the decommissioning of a detector, its collected data needs to be preserved [The09a]. Experiment data is typically unique (not superseded by other experiments) and can be used as a future cross-check for discoveries of other experiments. Novel analysis techniques and theoretic models can be probed against such data. However, the data alone has no meaning without any means to reprocess them. That requires the knowledge of how to do it as well as the entire software infrastructure to be available. Virtual machines offer an easy way of preserving such a software infrastructure including the operating system.

3.4 Challenges

In contrast to plain server consolidation, LHC data processing has some specific constraints and requirements that limit the straight-forward use of virtual machines. Instead of packaging a standard physical worker node as a virtual machine, many of the following problems can be avoided or mitigated by a carefully constructed virtual machine (Figure 3.2).

3.4.1 Performance

The additional virtualization layer will inevitably result in some performance loss. The problem is amplified by the internals of the x86 architecture and its 64 bit extension, the x84_64 architecture. Their instruction set architecture (in its original form) does not support the virtualization requirements defined by Popek and Goldberg [PG74, RI00].

A modern ISA has a CPU with at least two modes of operation. A privileged set of instructions is used by an operating system kernel, for instance for interrupt handling and virtual memory management. An unprivileged instruction set is used by the user's applications. Such a separation of modes is required in order to shield applications against each other and to shield system resources against abuse of applications. Popek and Goldberg define near-physical speed

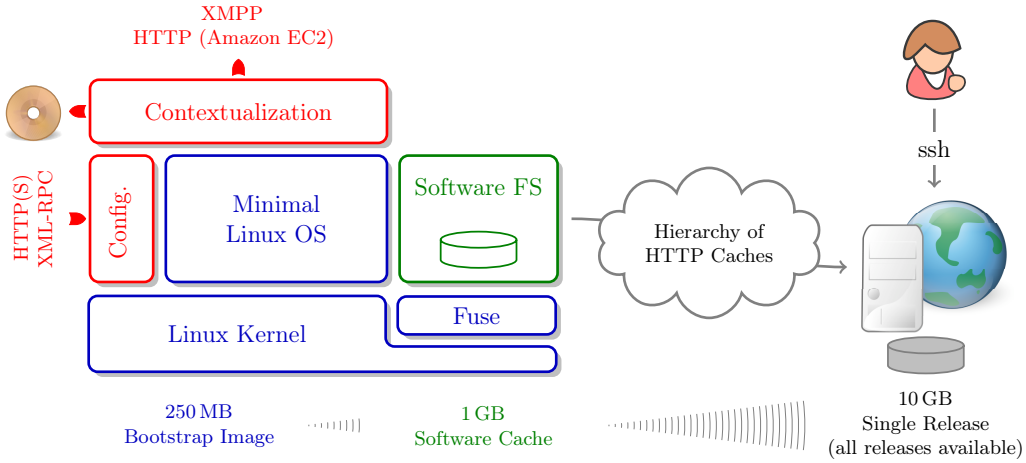


Figure 3.2: The CernVM appliance comprises three major components: 1. A minimal operating system derived from the experiment software dependencies. 2. A specially crafted software file system that downloads software from a remote web server on demand and caches data and meta-data. 3. Various contextualization and configuration interfaces to facilitate deployment in managed and unmanaged infrastructures.

as an essential property of a VMM in order to provide a virtualized application the illusion of running on physical hardware. That implies that the majority of instructions executed by the VM must not be emulated but executed directly by the physical CPU.

Popek and Goldberg annotate the instructions of the ISA as *privileged*, *sensitive*, or *conventional*. The latter are instructions that behave the same in either of the modes of operation of the CPU, such as addition of two integers. Privileged instructions are only executed in the privileged CPU mode, otherwise they raise a trap (e. g. the `HLT` instruction). Sensitive instructions behave differently depending on the mode of operation of the CPU. However, they not necessarily raise a trap when executed unprivileged; instead, they could for instance be ignored. An efficient VMM according to Goldberg and Popek passes the non-privileged instructions through; privileged instructions are emulated to make sure the state of the VM and the state of the host remain synchronized ("trap and emulate" virtualization). It is therefore necessary that the sensitive instructions are a subset of the privileged instructions. The x86 architecture, however, has 17 sensitive instructions which are not privileged [RI00]. For these "difficult" architectures, various VMM technologies have been developed, all of them with different performance characteristics. These are described in the following:

Binary Translation Binary translation combines emulation with just-in-time compiler technologies. It treats the stream of VM CPU instructions as source code and compiles it into an equivalent instruction stream for the host. Compilation takes place in blocks, whereas a block is defined as a maximum list of instructions without any contain control flow instructions (jumps) [Bru04, p. 30]. Compiled blocks stay in a code cache. For privileged and sensitive instructions, the VMM compiles a modified code path taking the host and the guest CPU state into account.

As the guest instructions are compiled rather than interpreted, Popek and Goldberg's efficiency requirement is effectively achieved. Moreover, the just-in-time compiler can even optimize the binary source code, such as avoiding unnecessary privileged instructions, or (pointer) arithmetic optimizations on common code paths [AA06]. Thus binary translation outperformed the early "proper" trap-and-emulate virtualization technology by hardware-assisted VMMs [AA06]. By so-called "guest additions", VMMs provide para-virtualized drivers for popular guest operating systems.

Examples of VMMs using binary translation are VirtualBox[1] as well as

[1] `http://www.virtualbox.org`

many commercial VMware products. Recent versions of these products choose dynamically between binary translation and hardware-assisted virtualization.

Para- and Pre-Virtualization Para-virtualization was introduced with the Xen hypervisor by Barham et al. [BDF+03]. It is a different approach to virtualization that requires the guest kernel to co-operate. Para-virtualization exploits the fact that the x86 architecture has not only two but four modes of operation, the *rings* 0–3. Out of those four rings, recent operating system use only ring 0 and ring 3. A para-virtualized operating system is modified in a way that it is able to run in ring 1. The guest system is called *DomU*, whilst the host system in ring 0 is called *Dom0*. As the DomU runs in ring 1, it is still shielded against its applications. As privileged instructions executed from ring 1 still trap, the Dom0 is shielded against its guests.

In contrast to binary translation, there is no overhead of the just-in-time compiler. The DomU operating system has direct (read) access to the physical CPU and the memory management unit (MMU). Hence it is able to perform informed decisions resulting in better performance. System calls can be forwarded to a guest kernel without the indirection via Dom0. I/O devices are not fully emulated, but rather multiplexed to the guests, using shared memory as efficient data transport. Barham et al. argue the effort of porting of the guest operating system is worthwhile due to the performance gain. They showed that the porting effort is rather low; para-virtualized kernels of Linux, Windows, and NetBSD can be created with less than 1.5 % of the source code modified.

LeVasseur et al. suggest a special compiler to do the porting of the guest operating system [LUC+05]. This is called *pre-virtualization*. In this approach, porting the guest becomes almost automatic and reduces the associated cost to a large extent. Depending on the quality of the compiler, pre-virtualization provides highly optimized versions of the modified code spots. Pre-virtualized Linux guests have been implemented for the x86 and Itanium architectures running on the Xen hypervisor and the L4Ka micro kernel.

Hardware-Assisted Virtualization The two major x86 CPU vendors developed ISA extensions in order to make the x86 architecture compliant to Popek and Goldberg's virtualization requirements (AMD-V [Adv05] and Intel VT-x [UNR+05]). Using these extensions, a conventional trap-and-emulate VMM can be build that runs unprivileged instructions natively, whilst emulating privileged instructions. Although developed independently, both extensions are very similar in concept. The virtualization extensions introduce another mode

of operation, a "VMM mode" beyond the conventional four privilege levels. Much like kernel traps, there are newly introduced VMM traps.

VMM traps are expensive operations. Also, the processor state as well as the virtual memory page tables have to be duplicated within the VMM for every guest. However, the hardware assisted performance improved. The effort for a VMM trap reduced from 1400 cycles in the Prescott processor to some 200 cycles on the Nehalem processor [AA06]. VMM traps due to page faults, another major source of performance penalties, have been addressed by AMD Rapid Virtualization Indexing [Adv08] and Intel Extended Page Tables [NSL+06]. These extensions arrange for the VM to manage its virtual memory without requiring to trap into the VMM.

With conventional hardware-assisted VMMs, I/O operations are still emulated. In practice, it is common to combine hardware-assisted virtualization with para-virtualized guest drivers. Usually, timers, network, and disk I/O are para-virtualized. Alternatively, the AMD-Vi [Adv09] and Intel VT-d [Int11] ISA extensions enable the memory and the interrupts of an I/O device to be directly channeled to a specific VM (*directed I/O*). Of course, that way I/O devices cannot be multiplexing to multiple VMs. With more VMs using directed I/O, more hardware devices have to be provisioned by the host.

The KVM kernel modules add a hardware-assisted VMM to Linux [KKL+07]. In this way, a virtual machine runs just as another host process. The KVM VMM benefits for most of its tasks from the elaborated operating system algorithms already implemented in Linux, such as for scheduling and memory management.

Container Virtualization Container virtualization is in fact not hardware virtualization. Instead, the system call interface is virtualized for groups of processes. These groups of processes get the illusion of working exclusively on the kernel. This is a lightweight virtualization which comes with very little performance overhead, usually only a few percent. Still, container virtualization provides isolation and flexibility features comparable to full hardware virtualization. Of course, it requires the "guest applications", i. e. the programs in the container, to be compatible with the host architecture and kernel. The Linux interface to the user mode has been very stable; container virtualization puts very little constraints on LHC experiment software.

There are a variety of Linux container virtualization products that come with a modified host kernel, such as OpenVZ[2] or VServer[3]. With Linux containers

[2] `http://www.openvz.org`
[3] `http://linux-vserver.org`

(LXC), there is also mainstream kernel support for container virtualization using in-kernel namespaces for isolation [Men07]. A very simple, yet not complete, container virtualization can be achieved by a "chroot jail". Together with virtual network devices, a chroot jail virtualizes the most common I/O subsystems disk and network.

Benchmarks and Mitigations

Our measurements with several hypervisors have shown that the performance penalty for a typical HEP application can vary between 5 % and 15 %, depending on the type of workload (CPU or I/O intensive), on the chosen hypervisor and on the CPU capabilities available to support the latest instructions aimed at improving virtualization performance [ASBB+11]. An optimal VMM technology has not yet been crystallized. Whereas binary translation, for instance, was significantly faster than hardware-assisted virtualization, with recent CPUs they have converged to a similar overhead.

There are some mitigations to the performance drawbacks. By providing virtual machine images for many hypervisors, the user can at least choose the hypervisor with the least performance overhead for a given physical machine. Secondly, not every workload is I/O intensive. Simulation is a CPU intensive task and faces almost no performance drawback from virtualization. Finally, performance-wise, the critical number is the overall number of processed events per second. The benefits of virtualization such as better resource utilization and less maintenance can outweigh small performance penalties.

3.4.2 Unmanaged Resources

Cloud resources should be considered *unmanaged*, i. e. without any guaranteed quality of service or computing environment. This is certainly the case for volunteer clouds. To a lesser extent, it is also the case for institutional clouds, as virtual machines might be migrated to another cloud infrastructure. Essentially, virtual machines have to be able to process data without relying on any supporting grid service. In particular, access to the experiment's job queue and data access must be handled from within the virtual machine.

The connection to the experiment's job queue has been tackled by *Co-Pilot* [Har10]. The Co-Pilot framework provides an adapter to connect virtual machines to task queues of multiple experiments. It exploits the fact that by using pilot jobs experiments anyway decoupled their job queues from standard grid queues.

Data access is not necessarily provided by local storage services. Instead, flexible remote data access means have to be available to the virtual machine. Several LHC experiments calculated the cost of moving event data required for data processing in and out of the scope of Amazon EC2. They concluded that data movement within WLCG is at least an order of magnitude cheaper. Nevertheless, the private cloud model can bring benefits to existing grid resource providers.

3.4.3 Image Distribution

Virtual machine images have to be distributed to physical hosts. A VM image is of the order of 10^2 MB to 10^4 MB. Distribution of such an image to worker nodes produces a considerable amount of traffic on a computing cluster. There are specific cluster file systems optimized for hosting virtual machine images such as VMware VMFS or OCFS2. Copy-on-write image files further simplify the VM image management, as a single read-only "golden image" can be used by all worker nodes with a small write cache local to the worker nodes. Typically only a very small fraction of the image file is changed at runtime. The *sheepdog*[4] distributed block device implements a decentralized data store for VM images. Wartel et al. presented a VM image distribution system for clusters based on BitTorrent [WCM+10]. Their system stages a 10 GB image in 20 minutes to some 500 hypervisors. The image distribution problem is best mitigated by small images.

3.4.4 Image Proliferation

Providing the base system for virtualized worker nodes is very different from yet another Linux distribution. Instead of providing a general purpose set of basic tools, it is rather a vehicle to run experiment software. As such, it is defined by the experiment software; it's core packages are precisely the dependencies of the experiment software. Naturally, this results in a slim operating system imposing minimal maintenance. While the maintenance issue appears as a side effect at first sight, it is not just a matter of convenience but in fact it is a necessity. Dealing with a proliferation of various slightly different base systems leads to the very same level of complexity one tries to avoid using virtualization.

Few, well-defined images are important from the view point of trust and security, too. Creating a trusted image may be a lengthy process as it requires several parties to audit and sign the image before it may be deployed on the

[4] `http://www.osrg.net/sheepdog`

grid [Cas10]. The process also requires the integrity of such an image to be guaranteed when it gets transferred across the network to different sites.

As the proximity of virtual worker nodes is unforeseeable upfront, it requires well-defined hooks for site operators allowing them to seamlessly adjust worker nodes to their environment. Failing to do so leaves site operators no other choice but to introduce their own modified images. Definition and use of that mechanism is called *contextualization*.

3.5 Volatility

In contrast to the static worker nodes of WLCG, cloud resources are volatile. The underlying hardware virtualization layer facilitates instant and massive addition and removal of worker nodes. The IaaS provider treats VMs as independent black boxes. Hence, LHC data processing in the cloud naturally has to avoid central points of decision making such as installation jobs and hierarchical data placement. Moreover, the services provided to virtualized worker nodes have to operate in a stateless manner. Chapter 5 presents methods to support a stateless software distribution service.

As far as interaction amongst VMs is concerned, the VMs require a means for zero-configuration node and service discovery and role negotiation. In the context of LHC data processing, the *PROOF on Demand* system, for instance, provides means to deploy an ad-hoc, zero-configuration data analysis cluster [MM10]. Chapter 6 presents a distributed cache algorithm for volatile computing environments.

3.6 Software Distribution

The distribution of experiment software to virtualized worker nodes is challenging. While grid sites provide software to worker nodes by a software service, such a service cannot be assumed by virtualized worker nodes in arbitrary environments. Packaging experiment software together with virtual machine images is unfeasible. Such large images represent a challenge to efficiently distribute to thousands of worker nodes. Volunteers are likely to be reluctant to download several gigabytes per week in order to contribute their CPU hours. Even if image distribution is solved from a technical viewpoint, the short half-life of the images represents a problem from the organizational viewpoint, as the images have to be endorsed. In the following chapters, we will study how to efficiently deliver software onto virtualized worker nodes.

4 Software Characteristics

The development process and the software organization of LHC experiment software differ from conventional software projects that are developed according to strict software engineering practices. Harvey et al. point out that "the development strategy for LHC experiment software follows an architecture-centric approach as a way of creating a resilient software framework that can withstand changes in requirements and technology over the expected lifetime of the experiment" [HMR09]. The framework is written by the experiment software team and includes core components, such as the description of detector geometry, magnetic field, and the event data structure as well as sophisticated algorithms used to provide the correct identification and measurement of all particles. These algorithms are typically written by physicists that are specialized in pattern recognition and analysis techniques but who are not necessarily proficient in writing software. Over time, the experiment applications become more and more enriched by new physics analysis algorithms that are using the framework. These analysis algorithms are often projects of one or only a few physicists. As several hundred physicists can typically get involved in software development activities, the total software base can rapidly become very large and complex.

On a typical day, there can be more than hundred developers that contribute to the code base of an experiment. In order to ensure both the possibility for part-time developers to quickly contribute new algorithms as well as a functioning software stack, experiments make heavy use of automation tools that support the software development process, such as versioning systems and tag collectors, dependency management systems, nightly build and test services, release validation tests and so on. The organization of the software becomes rather granular and reflects the independent software artifacts contributed by many physicists. Figure 4.1 shows that 10 % to 20 % of file system entries are directories or symbolic links.

Whereas software development is highly distributed, the release process is not. New releases are centrally published by the experiment's software group at CERN. Individual algorithms are not necessarily managed as part of the centrally maintained experiment software, but can be shipped with analysis jobs in the form of macros. These macros are compiled and linked "on the fly" against the framework. Therefore, the software sources have to be available not

only on a developer's machine, but also on a production worker node. Figure 4.1 shows that 10 % to 25 % of file system entries are C/C++ source files.

LHC experiment software makes heavy use of so-called *LCG Externals*. LCG Externals is a collection of software that is required by the experiment software, but not part of it. It comprises, among others, compilers, libraries (such as Boost), interpreters (such as Perl or Python), and a variety of Monte Carlo event generators (such as Pythia). The Monte Carlo event generators are accompanied by a significant amount of static data, e. g. function tables of numerical or measured functions, such as logarithm tables or *parton distribution functions* [Sop97]. Figure 4.1 shows that 50 % to 60 % of the volume of the LCG Externals are data files.

LHC experiment software uses C++ to a large extent. All C++ libraries that are dynamically linked at runtime have to be compiled by the very same compiler. This is due to a lack of standardization for name mangling when compiling high-level language features such as overloaded functions into C-compliant symbol names understood by the system's linker. Hence, a change of the compiler requires the full software stack to be re-published. Often, several versions of a release are published that differ only in the compiler version used to create the software.

LHC experiment analysis software is accompanied by grid middleware. Grid middleware is used to access experiment data files, to connect to the job queues and for authentication and authorization. In order to have a well-defined set of tools, the middleware is part of the experiment's software stack[1]; a specially "grid enabled" operating system such as XtreemOS [CJM+08] is not required.

Although some LHC experiment software components are portable and cross-platform, the common platform in WLCG is Linux. LHC experiment software is compiled and tested on Scientific Linux[2], which is a Red Hat Enterprise Linux clone.

The LHC and its experiments have an expected lifetime of at least 15–20 years. Therefore, it is risky to rely on a specific product from a commercial enterprise that might disappear from the market or change its support policy. LHC experiment software is free and open source software[3]. Hence from the license point of view, there are no restrictions for software distribution other than to grant access to the source code [Ope].

[1] Some grid sites, however, dynamically replace the supplied grid middleware by their local middleware installation.

[2] `http://www.scientificlinux.org`

[3] The particle simulation toolkit Fluka [FSFR05] makes an exception. Distribution of Fluka is cumbersome due to license restrictions. It is usually not used in grid jobs.

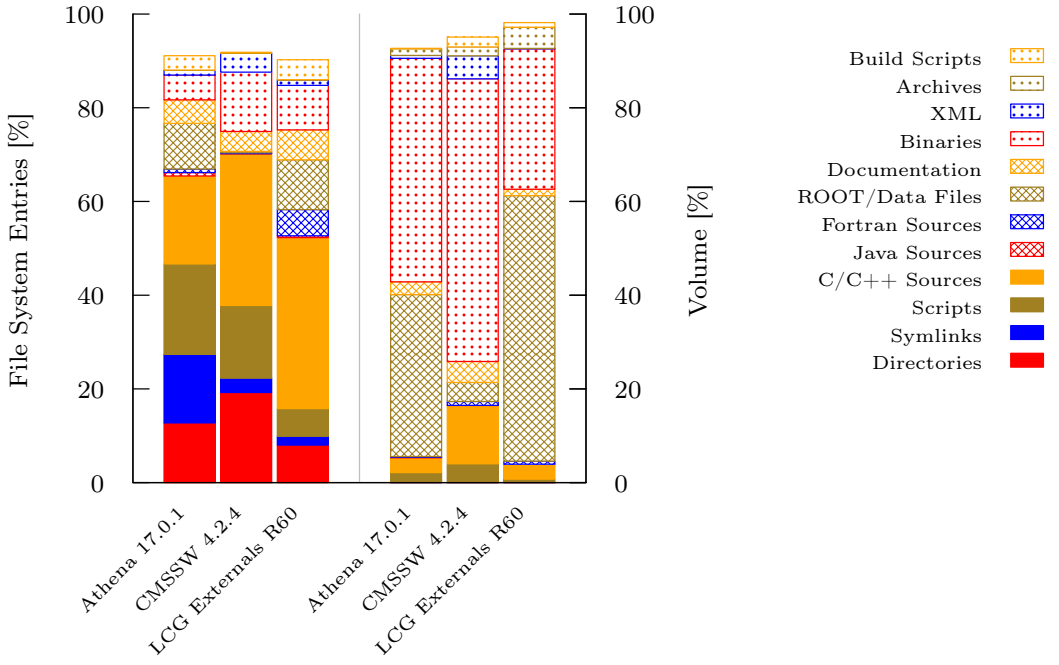


Figure 4.1: Contents of experiment analysis software releases. Classification is based on the file ending and the "execution bit" of the file mode. The gap to 100 % consists of unclassified files.

New releases do not render previous ones obsolete as analysis tasks are bound to specific software versions to preserve reproducibility. Reproducibility must be preserved at least for a couple of years, the typical creation time of a PhD thesis. In the light of long-term data preservation, experiment software should be available (and runnable) even tens of years after the decommissioning of an experiment. In this way, new discoveries can be cross-checked against data from previous experiments that originally did not search for a specific particle signature.

4.1 Building Blocks

This section provides a brief overview of some of the most important building blocks of LHC experiment analysis software.

ROOT ROOT [BR97, NB+09] is an object oriented large-scale data analysis framework written in C++. Not only does ROOT use C++ as its programming language, but C++ is also exposed to users as interpreted scripting and command and control language. Thus it facilitates rapid analysis algorithm prototyping while the very same code, in compiled form, can be used for high performance analysis of large data sets in production environments.

The C++ interpreter in ROOT is also the foundation for ROOT's C++ reflection system. The reflection system is required for the self-descriptive data storage of C++ objects. Collections of objects stored in so called *ROOT trees* are optimized for data analysis by column-wise storage. ROOT trees form the database for LHC event data and conditions data.

Furthermore, ROOT contains a variety of libraries. Amongst others, it contains a numerical engine, random number generators, descriptive statistic tools, a workload distribution system for clusters (PROOF), and tools for data visualization.

Event Generators There is a variety of event generators for simulation of the primary interactions, such as ALPGEN, HERWIG, JIMMY, Pythia, and others, many of them written in Fortran [KRZ08]. Besides the libraries and executables, these tools contain large tables describing measured probability functions of certain physics processes. The event generation provides input data for transport through the particle detectors, for simulation of the detector response, and for generation of the simulated raw event data.

Geant3 and Geant4 Geant3 [BBM+87] and Geant4 [AAA+03] are transport simulation toolkits. Geant3 is written in Fortran and among the LHC experiments only used by ALICE. Geant4 is written in C++ and used by ATLAS, CMS, and LHCb. Geant4 includes support for large geometry descriptions and a variety of physics models tailored to all sorts of particles and energy ranges.

GAUDI and Athena GAUDI is the data analysis framework of the LHCb experiment [BBB+01]. Athena — an enhanced version of GAUDI — is the data analysis framework of the ATLAS experiment [ATL05]. Athena is mostly written in C++. Despite using compiled languages for the core and compute intensive components, large parts of the analysis software run in Python, which is the framework's scripting language. Athena uses the POOL persistency framework [DÖ3] to access the experiment data. The POOL framework uses an RDBMS as meta-data storage for the plain ROOT data files. Additionally, Athena uses POOL to pre-produce plain ROOT n-tuples for further analysis.

AliRoot and CMSSW AliRoot is the data analysis framework of the ALICE experiment [BBC+03]. CMSSW is the data analysis framework of the CMS experiment [HLPW06]. These frameworks are closely coupled to ROOT. They follow a *software bus* architecture, where a core process iterates over event data. This core process can be configured to call plugins that implement the algorithms working on the data. Plug-ins are provided as shared libraries that are dynamically loaded.

Distributed Computing Frameworks The distributed computing frameworks comprise the generic grid toolkits Globus [FK97] and/or gLite [LHP+04]. The experiment specific grid toolkits are AliEn [SAB+03] for the ALICE experiment, PanDA [Mae08] for the ATLAS experiment, DIRAC [TBB+08] for the LHCb experiment, and CRAB [SLB+08] for the CMS experiment. Among others, they provide APIs for user authentication, accessing event data, and for the experiment's job queue.

4.2 Generic Properties of Software Files

Most of the following characteristics apply not only to LHC experiment software but to software in general. The requirement to be able to reproduce experiment results puts additional constraints on LHC experiment software.

1. Software is (almost) immutable. It is accessed in a write-once read-many (WORM) pattern. Immutability of a software release is induced by reproducibility. However, depending on the experiment policy, it is possible to sporadically replace (patch) files.

2. Software is versioned. A unique version string should identify an entire software stack. Ideally, such a version string identifies as well the underlying operating system version or a specific virtual machine image. Once published, a software release should remain accessible.

3. Software is read-only. This applies to executables. Configuration data might be subject to customization and localization. However, configuration data can be placed outside the software tree, such as the /etc directory on Unix platforms. We consider the publishing of new releases not as traditional "writing". Experiments release their software by a small group of release managers on a well-defined release environment. In contrast to "writing", publishing is an atomic step making a set of new and immutable files accessible.

4. Software has a simple access-rights model. This follows from the fact that it is read-only. Only the release managers are allowed to publish new and updated data. Usually, software is readable by anyone. In some cases, e. g. in case of license restrictions, software access has to be restricted to a closed user group.

5. Most software is not relocatable. This is usually due to the software installation tools that hard-code full paths. Once installed, the software always has to run from the very same directories.

6. Files are usually read as a whole. This holds for executables mapped into memory, source files read by a compiler, and configuration data. Static data is sometimes read in chunks.

4.3 Related Quantitative File System Studies

The quantitative characteristics of file system content have been studied before. Such studies help to optimize sizes for internal file system data structures such as blocks of buffers. For network file systems, the design is further influenced by the expected number of requests and the overhead of transferred bytes. Previous studies have some sort of averaged, typical file systems as data set. In our case, we are solely looking at LHC experiment software.

Tanenbaum et al. study the static file size distribution of university UNIX workstations [THB06]. Results are compared to the size distribution of a web server and to a similar study from 1984. The mean file size has doubled from 1984 to 2005, but nevertheless most files are still small (< 4 KiB). Evans and Kuenning study irregularities in the file size distribution [EK02]. They observe that file size distributions are subject to random spikes, making it hard to find a good fit function. We observe the very same phenomenon in LHC experiment software. Hence, we focus on the *cumulative* size distribution, i. e. the number of files smaller than a certain size. Leung et al. study network file system workloads in enterprise environments on Windows workstations using the CIFS protocol [LPGM08]. They follow up on a series of trace-driven network file system analyses, as well as a series of previous studies of Windows workstation contents. We will see that their observations almost entirely contradict ours. They observe client-dependent access frequency, a write-oriented workload involving large files that are rarely re-opened. Meyer and Bolosky study the file systems on Microsoft desktop computers [MB11]. They observe a trend to large files (megabytes to gigabytes) with a non-linear access pattern. The content is mainly dominated by software binaries, whereas LHC experiment software comprises a significant number of source files and static data.

LHC experiment software can be considered as a discrete file system workload. Besides size distribution, we will focus on compression characteristics, growth rate, redundant files, and the access pattern.

4.4 Cumulative Size Distribution

The software stack for an LHC experiment comprises several gigabytes and tens of millions of lines of code. ROOT and Geant4 alone contribute some 5 MLOC. As updates and new releases are published weekly or more frequently, the number of files is expected to grow to the order of 100 million files over 5 years. This is comparable, to within an order of magnitude, to the number of files of ALICE event data.

The size of the software stack is partly explainable by the size of the experiment collaborations. Collaboration members are software users but also software developers. For example, the amount of effort that was required for the development of the CMS experiment software, has been estimated to be $\approx$1000 man years [Law10].

The vast majority of files of LHC experiment software are small files, as shown in Table 4.1. In Figure 4.2 we see 50 % of all regular files are smaller than 4 KiB, and 80 % of all regular files are smaller than 16 KiB. Such small

files are stored in a single block or a few blocks on common local file systems. When compressed, they can be transferred in 1 or 2 Ethernet network packets (assuming a 1500 B maximum transfer unit). The LHCb software is similar to the web server content studied by Tanenbaum et al. [THB06]. Presumably this is due to the significant number of Python scripts used by LHCb. ATLAS, CMS, and ALICE, on the other hand are more dominated by compiled binaries and are close to the file system content of a UNIX workstation. However, larger files such as multi-media files dominate the tail of the cumulative distribution in the case of UNIX workstations.

		All Releases		Current Release	
Experiment	No. Releases	No. Files	Size [GB]	No. Files	Size [GB]
ATLAS	36	8 418 000	438	329 000	51
CMS	24	1 554 000	53	300 000	29

Table 4.1: Size of LHC experiment software for the 32 bit Scientific Linux 5 platform from June 2010 to June 2011. "All releases" includes the major software releases, as opposed to patch releases. "Current Release" referrs to the latest stable release of June 2011, including externals and grid middleware.

Between releases, as well as inside a release, there are a lot of duplicate files (see Figure 4.3). Over 40 releases of ATLAS software, for instance, show some 8 million path names referencing only 1.6 million distinct files. Duplicates occur, for instance, when the same sub-packages are copied from one release to another.

In the first couple of months of LHC data-taking we saw weekly releases of experiment software with a monthly data growth rate of about 10^6 files (10^5 unique files) and 10 GB to 50 GB per experiment (Figure 4.4).

4.5 Compression Rate and Speed

In this section, we analyze compression rates and speed for LHC experiment software. We compare commonly used lossless block compression libraries, which implement variants of the Ziv Lempel algorithm, Burrows-Wheeler transformation and Huffman coding. All libraries have been compiled with gcc 4.1.2 with full optimization. The compression and decompression speed measurements have been performed on a Xeon X7460 2.66 GHz having all data on a ram disk. On a normal hard disk, the compression and decompression

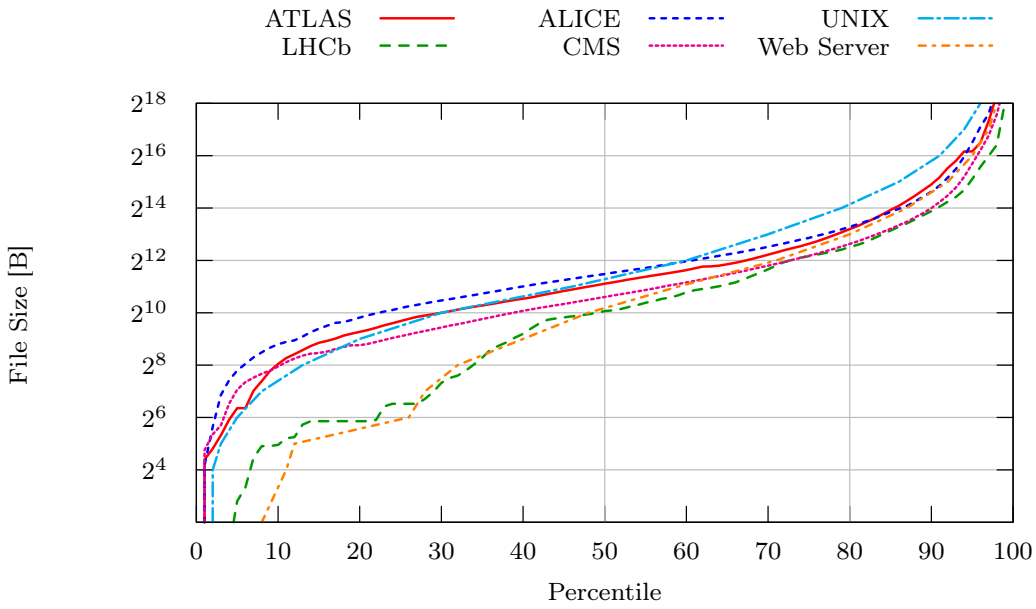


Figure 4.2: Cumulative size distribution of several LHC experiment software areas compared to the observations by Tanenbaum et al. [THB06]. "UNIX" refers to their UNIX workstation distribution and "Web Server" to the web server content they compared their results to.

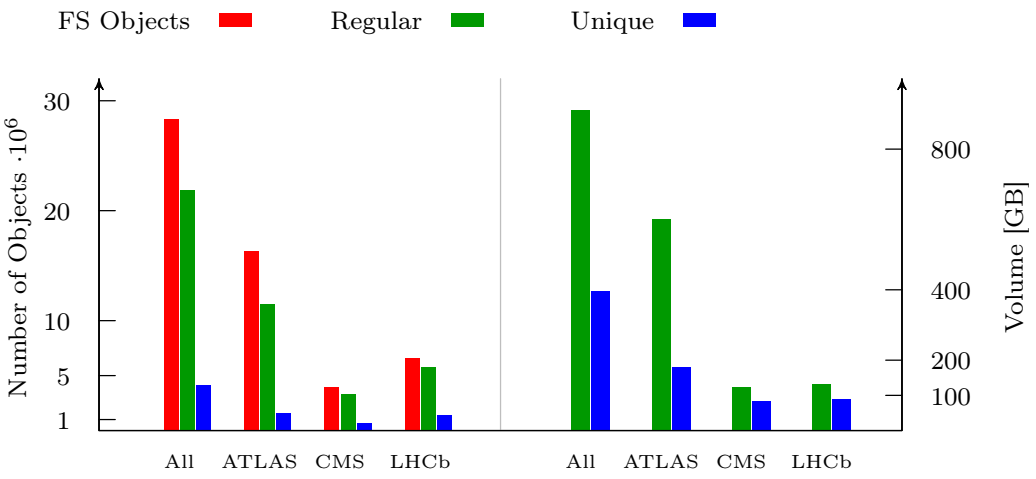


Figure 4.3: Size of ATLAS and CMS experiment software releases published during January 2010 to April 2011 in terms number of file system objects and volume. The number of unique files is a factor of 6 smaller than the overall number of files. By comparison with the right-hand side (volume), it turns out that the smaller files tend to be duplicated. This can be explained by the fact that larger files are usually binaries, not source files. Binaries change if *any* related source files change.

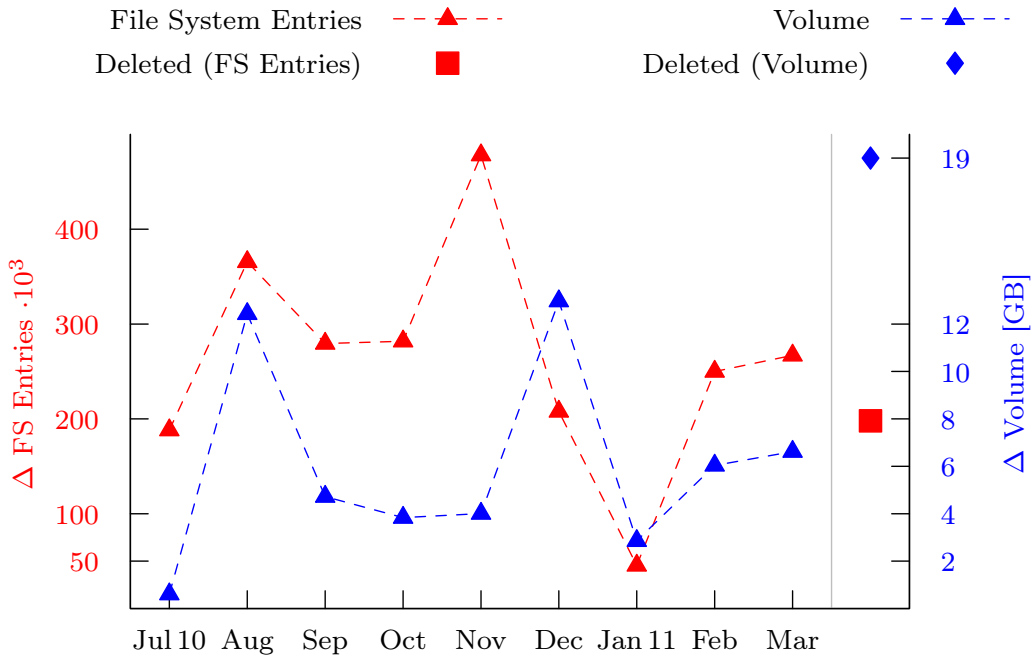


Figure 4.4: Monthly data growth rates of LHCb experiment software.

speed is also related to the compression rate, as smaller files can be read and written faster. The following libraries are used:

zlib 1.2.5 The zlib library implements the widely used DEFLATE algorithm, a combination of LZ77 and Huffman coding. The algorithm is subject of the RFC 1951.

snappy 1.0.3 The snappy algorithm is optimized for compression and decompression speed. It uses LZ77 without Huffman coding.

bzip2 1.0.3 The bzip2 utility is widely used on UNIX systems and optimized for good compression rates at the expense of low compression and decompression speed. It performs a Burrows-Wheeler transformation followed by Huffman coding.

lzo 2.02 LZO is supposed to be a real-time compression library and hence optimized for speed. It features very little memory consumption on compression and decompression. It uses variants of the LZ77 algorithm.

liblzf 3.6 LZF is optimized for speed and portability. It uses a variant of the LZ77 algorithm.

xz 4.999.9beta The xz utility is optimized for good compression rates at the expense of very low compression speed but competitive decompression speed. It implements the LZMA2 algorithm that uses a dictionary approach similar to LZ77

We benchmark an ATLAS software release and compare it to a set of files that represent the union of ATLAS software files used during 3 days by the CERN Tier 1 grid nodes. Table 4.2 shows the results. In general, a good compression algorithm for software yields a high compression rate, a fast decompression speed (as long as decompression is CPU bound), and a small increase of the compression rate for small files if software is distributed file-by-file. The compression speed does not matter that much because software has a WORM access pattern. The compression rate is better for the grid node set without changing the relative order of the compression libraries. In fact, the difference between the best rate and the worst rate gets larger. This can be explained by the non-compressible parts in the software release that are not accessed by jobs, for instance the compressed source packages. The order of compression and decompression speed, however, does change. This can be a hint on how big the impact of non-compressible data is to the compression algorithm.

	Athena 17.3.0				CERN Grid Node			
Library	$\div t_{\text{compr.}}$	$\div t_{\text{decompr.}}$	Rate [%]	Scale Factor	$\div t_{\text{compr.}}$	$\div t_{\text{decompr.}}$	Rate [%]	Scale Factor
zlib (default)	89.9	7.6	33.9	1.14	96.2	8.2	28.4	1.03
zlib (best speed)	36.3	8.3	37.1	1.12	34.6	9.0	31.8	1.02
zlib (best rate)	264.9	7.5	33.6	1.15	270.6	8.1	28.2	1.03
bzip2	346.6	64.1	30.1	1.19	395.6	58.5	25.2	1.04
snappy	9.2	3.6	47.8	1.08	9.8	2.9	43.8	1.02
lzo (default)	14.7	4.4	44.0	1.10	15.8	5.6	39.4	1.02
lzo (best rate)	428.5	4.1	37.4	1.13	403.8	5.8	31.9	1.03
lzf	10.7	4.1	46.5	1.09	11.3	3.8	42.9	1.02
xz (default)	1122.7	28.1	22.5	1.37	901.3	27.9	17.4	1.13
xz (best rate)	1764.1	25.8	20.4	1.47	1161.2	24.6	15.1	1.20

Table 4.2: Comparison of compression algorithms for LHC experiment software. "Athena 17.3.0" refers to the ATLAS Athena software release; "CERN Grid Node" refers to union of the ATLAS software files required by CERN grid worker nodes in three days. The rate represents the compressed size compared to the uncompressed size (smaller is better). The time columns for compression and decompression are specified as factors compared to a memory copy. Hence, compression and decompression factors have the same scale. The "Scale Factor" specifies the compression rate regression when compressing each file individually instead of the tarball of files as a whole. It specifies, how well the algorithm scales to small files, as we have in LHC experiment software.

	stat()			open()		read()		
Benchmark	all	uniq	Hits [%]	all	uniq	all	uniq	Hits [%]
Linux Kernel Compilation	438.8	4.2	98	426.9	2.4	426.2	2.4	99
ATLAS Examples Compilation	4 987.7	43.5	91	111.1	2.3	119.5	2.3	96
LHCb Setup Environment	75.6	11.0	81	5.8	1.2	12.3	1.2	41
ALICE simrec	1 607.7	1.5	98	2.0	0.4	0.4	0.4	0

Table 4.3: Overall number of system calls (in thousands), distinct path names (in thousands), and Linux file system buffer hit rates for several typical jobs.

4.6 Access Pattern

Unlike data, software requires a variety of features from the hosting file system. The hosting file system has to support symbolic links and hard links[4]. It has to support the `mmap()` system call to load executables into memory. It has to support POSIX file locking, which is a known issue of NFS shared software areas. At the scale of LHC experiment software, the file system has to support many concurrent open file descriptors. Athena, for instance, loads some 800 shared libraries on startup, which is close to the Scientific Linux 5 default limit of 1024. Moreover, as these libraries are opened by the system's linker almost concurrently and since jobs usually start at the same time, the access pattern results in large peak loads for shared software areas.

A single analysis job typically does not require all the experiment software but only a small fraction of the files of a particular release. However, those few path names are subject to a lot of system calls, in particular `stat()` calls (Table 4.3). Moreover we see lookups for non-existing files at about the same order of magnitude as successful lookups. Lookups for non-existing files occur, for instance, when the linker searches for libraries in a list of search paths. Hence for efficient execution, the hosting file system should provide a negative meta-data cache (caching "file not found" responses) in addition to a positive meta-data cache. Running analysis jobs on the worker nodes in a cluster might easily overload a shared software area [L+11]. This occurs even on parallel file systems, such as pNFS and Lustre, because the bottleneck is on meta-data operations.

Files in LHC experiment software are located in deep directory structures, as shown in Figure 4.5. Meyer and Bolosky observe that the majority of files is located in not more than five directory levels [MB11], whereas LHC experiment

[4] In LHC experiment software, however, hard links do only occur within the same directory, such as the `cc` binary to the `gcc` binary for the GCC compiler.

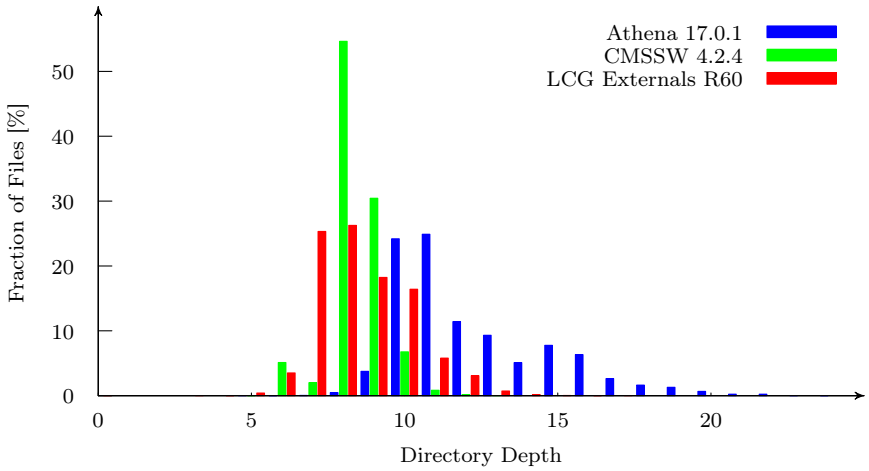


Figure 4.5: Fraction of files on located at a certain directory level. Counting starts with the highest directory under the responsibility of the collaboration, i. e. there are no artificial leading levels such as /usr/local.

software has the majority of files located in deeper than 5 levels of directories. This is caused by the fine grained directory structure within each release as well as the organization of different versions, platforms, and software parts. For instance, the release version, the architecture (32 bit or 64 bit), and the type of release (nightly build, production release) each account for another directory level. According to Conway's law, the long trail for ATLAS might reflect an organization of the collaboration with many subgroups. Naturally, deep directory structures cause the number of meta-data operations (such as `stat()` or `ls()`) to increase.

If distributed pre-installed, software directories have to be mounted at the same location on all worker nodes that was used to install it. In effect, this way of distribution requires a global namespace, such as AFS [MSC+86] is using.

4.7 Borderline to Event Data and Conditions Data

Many software characteristics also apply to experiment data and conditions data. Experiment data and conditions data are fully immutable, versioned, read-only, and with a simple access-rights model. Each particular job requires only a small fraction of all available data.

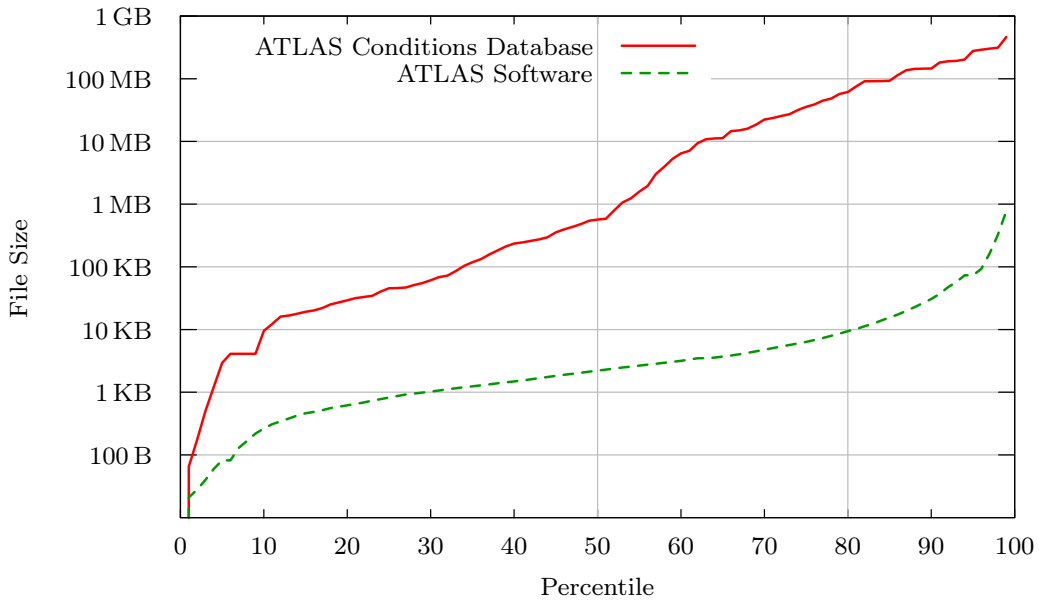


Figure 4.6: Size distribution of ATLAS conditions data compared to ATLAS software. The file sizes grow more rapidly. While 50 % of all files are smaller than 1 MB, almost 20 % of the files are larger than 100 MB.

In contrast to software, data is relocatable and often only certain blocks are required, for instance only a subset of n-tuples available in a ROOT file. Path names do not necessarily provide enough information to find a certain data set. Additional information about the data sets and their associated physical files is stored in relational databases in order to perform the filtering and searching. As Figure 4.6 shows, the size distribution of conditions data is much different from software with file sizes reaching the gigabyte range.

The amount of redundancy is low because every data set contains information of another measurement. However, this holds true only for raw data. Post-processing steps such as reconstruction and physics annotations produce new data from the raw data in an algorithmically precise way. In that sense, storing the reconstructed data can be seen as an optimization. The space cost is considered to be cheaper than the re-processing cost. In fact, the original version of the Virtual Data Toolkit [Roy09] produced all intermediate steps from raw data to analysis histograms only on demand.

4.8 Software Distribution in WLCG

Although there is a variety of distributed file systems for the cluster scale, there are only few POSIX compliant file systems for geographically distributed resources that could possibly be used to distribute software. Such file systems interface to peer-to-peer networks [KBC+00, MMGC02, Kut08]. Because of the organizational boundaries of WLCG sites, however, a peer-to-peer network on the worker node level is currently unfeasible. Furthermore, the underlying peer-to-peer networks have to be optimized for small latency, i. e. they have to serve thousands of requests for chunks per node and second. Unlike data distribution systems such as the Hadoop, the problem at hand requires efficient meta-data handling. Annapureddy et al. presented a file system for the distribution of a runtime environment based on the Coral content distribution network [AFM05]. Unfortunately, the source code was not released, neither was it available upon request. Furthermore, the scale of their benchmarks is one to two orders of magnitude smaller than required for LHC experiment software.

Hence, the currently used systems for software distribution in WLCG are mostly based on package managers and cluster file systems. In the following, we summarize the state of the art.

Andrew File System The *Andrew File System* (AFS) [MSC+86] is a distributed read-write file system optimized for home directories on globally distributed workstations. AFS provides a global name space (/afs) that is partitioned into *cells* and *volumes*. ATLAS, CMS, and LHCb host the authoritative copies of their experiment software on public volumes (cell /afs/cern.ch). Using AFS, software need only be installed once in order to be used by all clients. AFS clients cache opened files in a local, persistent cache. The meta-data reside in a local memory cache. *Coda* extends AFS with support for disconnected operations [Bra98], which implies a persistent meta-data cache. Coda is not considered as a production grade system by its developers.

AFS has a client/server ratio of about 200:1. This is mostly due to the fact that an AFS server has to keep the connection states of all its clients. AFS uses the Rx protocol on top of UDP, which provides similar services to TCP. In order to ensure cache consistency, AFS uses callbacks from the server to the clients. This causes connection failures for clients connecting from behind a NAT layer. Virtual machines are often behind a NAT layer provided by their hypervisor.

Grid Installation Jobs So-called grid installation jobs are special grid jobs used for software installation. They are supported by the gLite and AliEn package managers. Grid installation jobs pre-install software releases on the worker nodes. Hence, grid installation jobs for a certain release have to be executed successfully on all sites before jobs can be scheduled that use the release. As not every user is supposed to install software, grid site administrators maintain special accounts for the experiment's software managers. Software managers, in turn, have to manually supervise the successful installation of new releases on all sites.

In order to not download the software packages onto all worker nodes, grid sites use so-called *shared software area*s. A shared software area is essentially some space dedicated to experiment software on a cluster file system such as Lustre or NFS, which is mounted by the worker nodes. Thus software is installed only once per grid site. However, shared software areas introduce a central point of failure and a high chance to overload the servers by meta-data operations.

ALICE Software Installation System The ALICE software installation system can be considered as improved grid installation jobs. Software releases are still distributed as packages. On the grid sites, these packages are distributed using BitTorrent [Coh03] among the site's worker nodes. Even though there is a BitTorrent component in the system, the overall service is centralized with an authoritative tracker at CERN. There are the following differences as compared to normal grid installation jobs:

- Software is installed locally on each worker node. Hence, there is no shared software area that can overload.
- The distribution of packages inside a grid site is done via BitTorrent. Hence, the repository servers do not overload even though all worker nodes fetch packages.
- Software packages are installed on demand. Hence, there is no need to supervise pre-installation of software releases.
- ALICE software is relocatable. Hence, it does not require any fixed worker node configuration, but the installation system finds some scratch space in most cases.
- ALICE uses stripped software packages tailored to grid nodes, which comprise only some 300 MB. It is unclear if such a stripped version can be created for other experiments.

While BitTorrent prevents the repository server from overloading, still *all* files of a software release have to be transferred to *all* worker nodes. That not only results in increased network traffic, extracting and installing the packages is also time consuming and the probability of failures increases.

GROW-FS Given the problems with grid installation jobs, the CDF experiment used the special purpose file system GROW-FS for software installation [CGL+10]. GROW-FS pioneered the idea of a shared software area for installation in combination with persistent data and meta-data caches on worker nodes. Hence, the meta-data operations are performed decentrally on the worker nodes. As shared area for installation, GROW-FS uses a web server. The directory tree is pre-processed before distribution in order to create a file catalog containing meta-data and MD5 checksums of regular files.

GROW-FS is designed for local, trusted networks. GROW-FS uses a single file catalog for the entire directory tree, which is not digitally signed. Changes to the software area require complete reconstruction of the file catalog as well as cache invalidation and remounting on the worker nodes. On worker nodes, the entire file catalog is loaded into memory when the file system is mounted.

4.9 Design Criteria

In this section we summarize the desirable design criteria for a software distribution system. The criteria are based on the LHC experiment software characteristics and the experience gained using various software distribution methods in WLCG. The software distribution system has to scale to the order of 10^5 worker nodes and it has to work equally well from individual worker nodes to large grid sites. The most important design space dimensions for scalability and maintenance overhead are shown in Figure 4.7.

One-Time Installation Software should be centrally installed by the experiment's software release manager. As we assume non-relocatable software, we require a network file system mounted on all worker nodes that provides a global namespace.

Scalable Distribution A single point of installation must not become a single point of failure for distribution. Hence, data has to be replicated and cached along distribution to the worker nodes. Only data and meta-data actually required should be transferred. In a network of 10^5 worker nodes, failures are inevitable. The worker nodes need to automatically recover

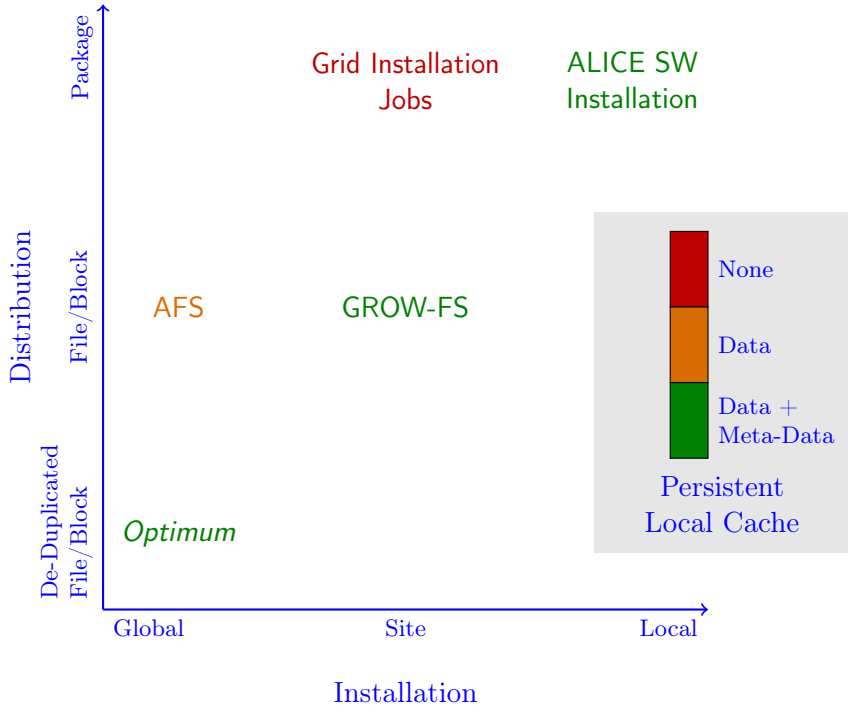


Figure 4.7: Classification of software distribution methods in a 3-dimensional design space.

from failures and they need to automatically fail-over to replicated data resources.

Decentral Execution The large number of meta-data operations can only be handled in a decentralized manner by the worker nodes. Worker nodes require a positive and negative meta-data cache in addition to a data cache.

Redundancy Elimination LHC experiment software is subject to a lot of redundancy. By file de-duplication and compression, the overall number of files and the volume can be deflated by factors of 5–10.

Location Transparency In order to support volatile computing nodes, software access should be location transparent. This means, the physical moving of worker nodes to different network addresses has to be supported.

Integrity and Authenticity Ensuring software integrity and authenticity is crucial. However, many data access systems do not ensure data integrity (e. g. cluster file systems, WebDAV). While corruption to data files may crash a worker node or in the worst case result in false physics results, the potential damage of unverified software is much higher. WLCG is an attractive victim for attackers because of the large computing power and the good network connection of grid sites. Unverified software can, for instance, be exploited to run distributed denial of service attacks or to factorize large integers.

Technical Restrictions The file system running on the worker nodes has to support symbolic links, hard links, and the `mmap()` system call. In order to allow for local customization of configuration files, the file system additionally has to support *variable links* that are evaluated at runtime. The file system has to work properly behind a NAT layer and firewalls. Thus, it must only use standard network protocols and the connection has to be initiated by the worker node.

5 Software Distribution

As pointed out, a file system for LHC experiment software distribution should support data caching and positive and negative meta-data caching. Caching and replication are the standard techniques used to reach scalability and resilience in distributed file systems [TvS07, Section 12.6]. This chapter reviews the problems that have to been solved when caching and replicating software files. The use of content-addressable storage is proposed as the principal format for distributing and caching data and meta-data. We address several engineering issues in order to tailor the use of content-addressable storage to the problem at hand.

5.1 Caching and Replication

When fetching files from nodes other than the master source (i. e. from cache nodes or replication nodes), one has to address the following issues:

Replacement Once a cache is filled, new items have to replace existing items. The algorithm that selects which files are to be replaced is determined by the replacement strategy. The predominant strategy is to replace the *least recently used* (LRU) item. Panagiotou and Souza show that LRU comes close to the optimum in practice [PS06]. Furthermore, all operations on the LRU data structure (insert, delete, access) can be implemented with complexity $\mathcal{O}(1)$ using a hash map and a linked list.

Expiry and Consistency Unlike event data and conditions data, software files and static data are not truly immutable but might be subject to patches. This means, the same path name might refer to different versions of a file. Software files are, however, not independent from each other. When, for instance, the dependent libraries A and B are patched, a cache node might still deliver the original library A, while B has expired and a patched version is reloaded from the master source.

Corruption and Poisoning Both, cache corruption and cache poisoning result in erroneous files being on the worker node. Cache corruption is an unintended phenomenon caused by faulty hardware or software bugs. In

fact, corruption can even occur when fetching files from the master source due to undetected transport errors. Such errors usually result in a crash of the software on the worker nodes. Cache "poisoning", in contrast, is a malicious replacement of files in a cache node. Exchanging software files can be used to execute programs different from the intended physics analysis on the worker nodes. Cache poisoning is a problem that arises due to insecure infrastructures.

5.2 Content-Addressable Storage

In order to solve the problems of consistency, corruption, and poisoning altogether, without spoiling scalability, the use of cryptographically signed content-addressable storage (CAS) has been proposed [DKK+01,BMP02]. With content addressable storage (CAS), files carry a file name that depends on their content rather than on their location in a directory tree or on a storage device. The content address is retrieved from a cryptographic hash (or at least a collision-free hash [BSNP95]) of the content. Content addressable storage has many advantages, in particular for LHC software repositories. They include the following:

- Data integrity is trivial to verify by re-hashing files.
- Maintaining cache consistency is trivial as files are immutable and never expire. Hence, caching becomes protocol-independent and a caching service can be stateless.
- Identical files in different locations are mapped to the same content addressable file. Hence, CAS provides content de-duplication.
- As file changes result in new file names, CAS facilitates the construction of versioning file systems.
- The hash key used as file name can be re-used for distributed hash tables and key-value stores.

Content-addressable storage has been used in various contexts: Quinlan and Dorward used CAS for de-duplication on archive storage [QD02]. Tolia et al. pioneered the idea of opportunistic CAS in distributed file systems [TKS+03]. They use CAS as an alternative channel for fetching files. Content-addressable storage is naturally used to store data in distributed hash tables. A number of peer-to-peer file systems have been developed on top of distributed hash

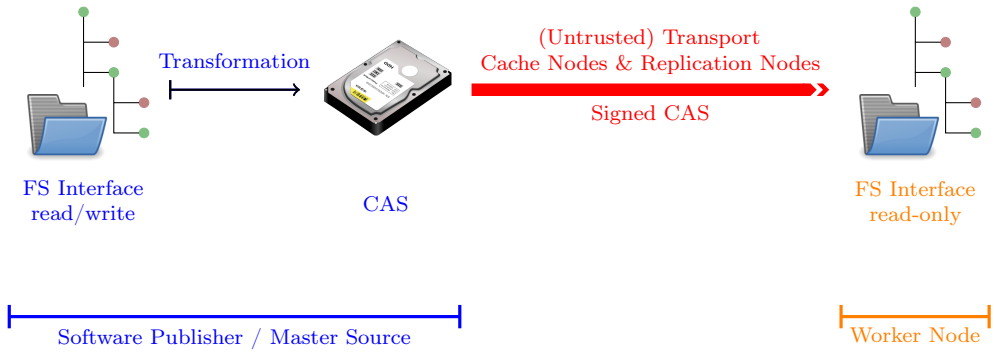


Figure 5.1: End-to-end software distribution using content-addressable storage. Both, the publisher's and the worker nodes use a file system interface. On the publisher's end, software updates are staged. On the worker nodes, the file system is read-only (but not immutable). For transport, caching, and replication content-addressable files are used.

tables [RD01, DKK+01, MMGC02]. The *git* versioning system[1] internally uses content-addressable storage in order to replicate entire repositories including version histories; it does not have a concept of caching and on-demand delivery. The *camlistore* system[2] uses an HTTP interface to content-addressable storage to store and synchronize personal data.

The concept of content-addressable storage is flexible and allows for context-specific adaptations. In the context of software distribution, CAS should be used "behind the scenes" (distribution, caching, replication, etc.) whilst both the worker nodes and the publisher's end require a regular file system interface in order to execute applications and to stage software updates (Figure 5.1). Towards such a design, we address the granularity of CAS (block level or file level), the encoding of the directory structure, the efficient staging of software updates, and data confidentiality.

5.2.1 Block Level and File Level CAS

Content-addressable storage can be applied to entire files or to sub-blocks of files. With respect to de-duplication, we have seen in Section 4.4 that file level de-duplication reaches a deflate factor of 5–6 for the number of software files and static data files. Furthermore, most files are read as a whole and are

[1] `http://git-scm.com`
[2] `http://camlistore.org`

smaller than a single network packet. In that case, block level CAS would increase the number of requests without improving performance.

For completeness, we will briefly discuss CAS for conditions data and event data. Such data contains recorded measurements and as such there is very low probability of having redundant data at all. From conditions data and event data files, often only certain blocks are read. An optimal block size depends on the individual data analysis job and its request pattern. Since we might want to cache and distribute data *in memory*, a single verifiable entity should anyway not be larger than a couple of megabytes.

For large files, one might also argue that in case of verification failures entire files have to be re-transmitted as opposed to just one failed block. That depends on the error probability which is influenced by many factors, such as transmission errors, wrong blocks read from a hard disk and so on. Let us assume a setting in favor of small block sizes with a high error probability of one error per 100 GB ($p_{\text{error}} := 1/(100 \cdot 2^{30})$), a large file size of 10 GB ($Z := 10 \cdot 2^{30}$) and a small request size of 10 B ($r := 10$). We are searching the optimal block size x that minimizes the average file transfer size. We have to transfer Z/x blocks. The probability for a successful block transfer is $(1 - p_{\text{error}})^x$ and the average number of block transfers is given by $1/(1 - p_{\text{error}})^x$. Hence we have an average transfer size of the file of

$$\frac{Z}{x}(r + x)\frac{1}{(1 - p)^x}.$$

Minimizing the formula results in an optimum of x of around 1 MB. However, even with a block size of 100 MB the average transfer size increases by less than 10 MB. So, clearly the block size is determined by the requirement to cache data in memory.

5.2.2 Key Space

The content hash function $h : \{0, 1\}^* \mapsto \{0, 1\}^n$ has to be chosen with respect to the expected overall number of hashes. Assuming an order of 10^5 files per release, a release rate of once per week, an expected lifetime of LHC experiments of 15–20 years, and a safety margin of one order of magnitude, we have an upper bound of 10^{10} files. Using cryptographic hashes, the hash keys are uniformly distributed in the key space. The probability of a hash collision with m files in an n-bit key space $p(m, 2^n)$ is given by the *birthday paradox* [Fel68]; it is approximately

$$p(m, 2^n) \approx 1 - \frac{1}{e^{m^2/2^{n+1}}}$$

and for a given threshold collision probability p, the minimum number of bits is

$$n \approx \log_2 \left(\frac{m^2}{\ln \frac{1}{1-p}} \right) - 1.$$

For a desired collision probability of less than 10^{-20}, which is much less than the unrecoverable error rate of typical hard disks, we need 132 bits. While common cryptographic hashes such as SHA-256 or RIPEMD-160 are well beyond that limit, the estimation can be useful in two cases:

- To choose the size of the hash space of cryptographic hash algorithms with variable output sizes such as *Skein*[3].
- A cache layer might tolerate occasional collisions at the benefit of reduced key sizes (cf. Chapter 6). Uniformly distributed keys with a reduced size can be constructed from any subset of bits of a cryptographic hash, e. g. a prefix. Here, we have obviously a cache-dependent value of m and p. For example, for 10 000 files and a desired collision probability of less than 3 % 44 bits suffice.

Cryptographic hashes might get broken, which has happened for MD5 for instance. In such a case, all files must be re-hashed using another algorithm in order to restore resilience against cache poisoning.

5.2.3 File Catalogs

A file system interface on top of content-addressable storage requires means to translate the directory location into CAS. That is done by file catalogs, special "translation files" that map the directory location to the hash key of a file (Tolia et al. refer to it as *recipes* [TKS+03]). Some file systems have a separate "translation unit" per file [TKS+03], some systems bundle translation entries per directory [Kut08], and some systems use one translation file per file system [CGL+10]. A single translation file per file system does not scale to large file systems. Fine-grained translation files, on the other hand, will result in many meta-data requests.

User-Assisted Partitioning

Here, we propose a user-assisted partitioning of meta-data based on the directory tree. Usually, we have strong meta-data locality based on directory subtrees,

[3] `http://www.skein-hash.info`

such as project directories, home directories, or software release directories. A software publisher has the following helpful knowledge about the directory tree:

- The parts of the directory tree that are used together. This is usually given by the software release root directories. Jobs require files only of a particular software release.
- The parts of the directory tree that are updated together. Here, it is important to know about volatile directories such as patch releases and certificate stores. The frequent updates of such directories should not affect other parts of the directory tree.

As the corresponding subtree roots are easy to spot for the publisher, he assists in marking a subtree's meta-data to be stored in a separate translation file (a *nested catalog*). Such "subtree markers" can be maintained by creating or removing magic files. The approach is similar to *volumes* in AFS but more flexible and lightweight, as only meta-data are affected by the cutting.

Nested file catalogs can be considered as a meta-data pre-cache. Consider, for instance, the ALICE simulation and reconstruction use case from Table 4.3 on page 50, which accesses 1500 distinct paths. We assume the data publisher created subtrees as shown in Figure 5.2. In that scenario only four catalogs have to be available in order to execute all meta-data operations. With a directory based cutting some 60 translation files are required. One might argue that the subtrees of the nested catalogs comprise some 18 300 entries in total, so we load ten times more meta-data than required. The volume of the meta-data, however, is very low, even for the small software files at some 0.25 % to 0.5 % of the overall volume. Hence we optimize for fewer number of files and requests. We note that any optimization for meta-data partitioning can be spoiled by recursive listings, for instance by the `find` utility.

File Catalog Layout Optimization

In order to support a file system interface, the catalog has to provide the operations path $\mapsto$ CAS Key and inode $\mapsto$ CAS Key. The path query can be supported by traversing the directory tree, which is time consuming. A direct lookup using the path as key, on the other hand, requires full paths to be stored for all entries. For LHC experiment software, the average path length is between 100 and 180 characters. Here, we propose to store a secure hash of the full path. In this way, the 100 B–180 B are compressed to some 16 B–20 B depending on the hash function. The cryptographic requirements of this hash function are relaxed to uniform distribution of keys, as the publisher needs to be trusted anyway.

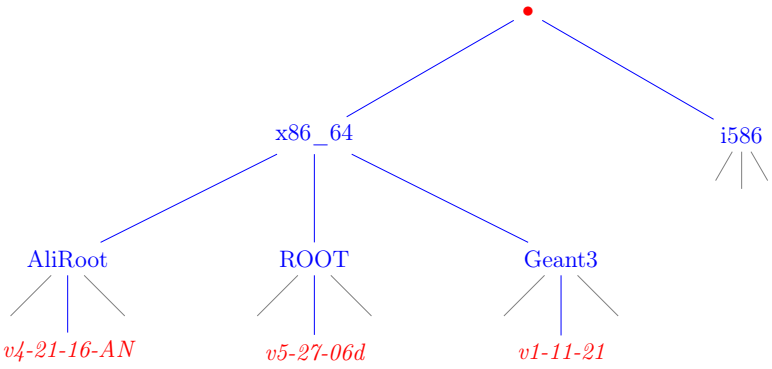


Figure 5.2: First levels of the ALICE software directory tree (simplified). Red nodes (italic font) indicate a user-assisted nested file catalog. Similarly to systems that store the translation file per directory, nested catalogs can be created recursively with the content hash of a sub file catalog stored in the parent catalog (*hash tree*). The content hash of the root file catalog fulfills two purposes: it carries a cryptographic signature in order to ensure data authenticity of the file system and it can be used to populate updates, either by using an expiry time stamp or by a publish-subscribe system.

Replication

Besides caching, we note that the file catalogs allow for a more efficient incremental replication than the standard *rsync* tool. Given that the updated file catalogs are replicated first, the difference set between master storage and replica storage can be constructed locally on the replica storage. Since CAS files are immutable, the difference set simply consists of all hash keys listed in the file catalogs that are not yet on replica storage.

5.3 Pre-Fetching

Pre-fetching is a widely used mechanism to reduce latency when opening and reading files [GA94, KL96, Cha01, SBM05, RP05]. For the workload of many small files the main task of pre-fetching is file access prediction (as opposed to *read ahead*, for instance, that is used for read requests on a single large file or streamed content). The prediction tries to determine files that are to be accessed soon, given the history of previously accessed files. Such a prediction is then used to request multiple files at once or to pre-build file packages.

For LHC experiment software, we can assume large cache sizes of the order of gigabytes on the worker nodes. This assumption is based on the amount of storage that is currently devoted to software files per worker node. Thus we can afford long-term predictions. Usually, file accesses occur in spikes of the order of up to thousands per second. As a consequence, the prediction needs to be determined beforehand and stored as meta-data. For more accuracy, the predicted information should be stored per path. For less storage consumption, the prediction information should be stored per content-addressable chunk, i. e. after de-duplication. As identical binaries are likely to access identical dependent files regardless of the path used to access them, we propose predictions per content-addressable chunk.

In order to predict upcoming file accesses, a number of mechanisms have been proposed, such as *informed prefetching*, speculative execution, or based on the analysis of previous accesses. In addition to these mechanisms, software files allow a simple yet effective *semantic* approach. Shared libraries and executables intrinsically store the dependent shared objects that are dynamically linked but statically loaded. Under Linux, for instance, such dependent shared objects can be extracted by the `ldd` tool. These dependent libraries are accessed when the executable is loaded, which is the vast majority of cases for an open request on executables. For LHC experiment software, there are on average between 10 and 50 dependent shared libraries per binary. This approach differs from the proposal of *file cules* [IDG06], which are distinct sets of files that are always accessed together; shared libraries, in contrast, are by definition in multiple such sets.

In addition to statically loaded shared libraries, many LHC frameworks use shared objects as dynamically loaded *plug-ins*. Figure 5.3 shows the monthly hot set of accessed plug-ins for ROOT. According to these figures, a long-term prediction of required shared libraries covers approximately half of the available plug-ins with very high accuracy. Hence, instead of pre-fetching we propose to *preload* a software file system cache based on statistics about library usage.

5.4 CAS Transformation

Transforming directory trees into content-addressable storage, i. e. calculating the content hashes, is a costly process. Some systems perform the transformation in the backend of a specially crafted file system [QD02]. This approach has been used for archival storage and might be suitable for streams of event data and conditions data as well, where continuous streams of immutable files are written. Other systems, such as *git*, perform the transformation at certain

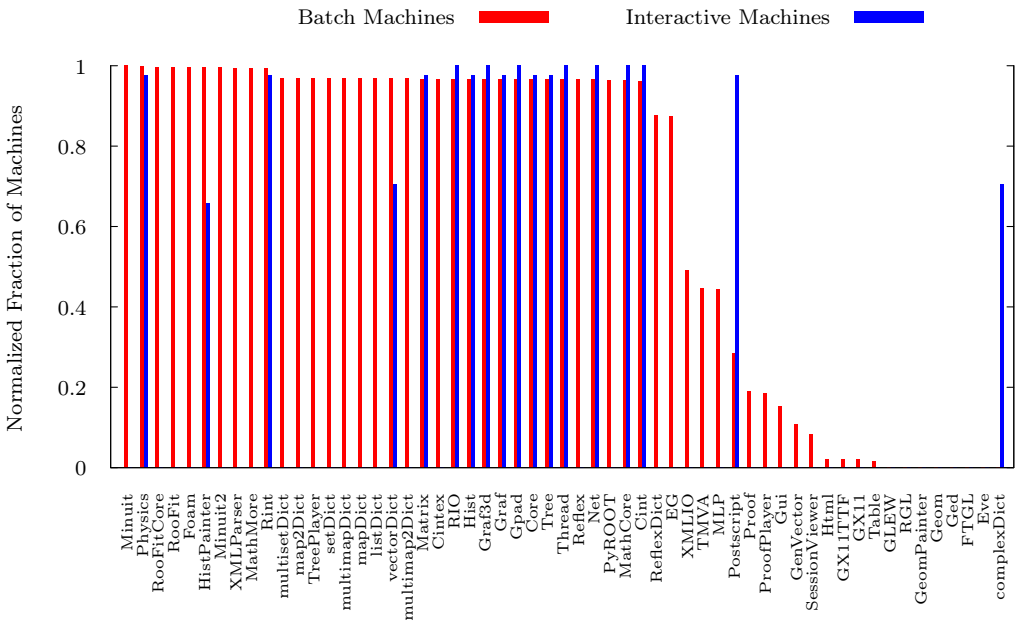


Figure 5.3: Monthly cache hot set for ROOT libraries over 1650 CERN Tier 1 worker nodes. "Batch" refers to the ROOT version of ATLAS Athena 17.0.0. "Interactive" refers to the ROOT version of the LCG externals that is mostly used for interactive data analysis. The numbers are normalized by the highest number of cache entries. There are 65 additional libraries available in ROOT, which have not been accessed at all.

Experiment	Writing System Calls [$\times 10^3$]	Modified Entries [$\times 10^3$]	Ratio [%]
ATLAS	6113	1829	29
CMS	4790	1391	29
LHCb	781	226	28

Table 5.1: Number of writing system calls compared to the number of modified files and directories during updates of LHC software directory trees. Statistics have been gathered during August 2011

checkpoints (`git commit`). Not only can the processing be parallelized, such systems also avoid to process files that are transient between checkpoints. Such transient files occur during the software installation process on the master storage, which comprises the following steps:

1. A new software release or patch release is added to the software directory tree. Typical transient files are temporary files produced during compiling the software or files that are renamed to their final location by an installation script.
2. The new software is locally tested by verification programs. At that point, the publisher has still the option to make further changes in case of verification failures.
3. The modified directory tree is published.

Table 5.1 shows the number of writing system calls for steps 1 and 2 compared to the number of actually modified files and directories. Existing checkpoint based systems, however, traverse the entire directory tree to determine changed files, which limits the scalability.

5.4.1 Incremental Synchronization

We will discuss two new approaches to perform incremental CAS transformation at the master storage. "Incremental" implies that only modified parts of the directory tree should be analyzed and processed, as opposed to a full reprocessing or a universal lookup for modified parts.

File System Change Log

Incremental transformation can be done providing a file system change log exists. Such a change log would track the changes made to the directory tree.

Compared to immediate transformation, a modified file or directory would just be added to a candidate queue. The entries of the candidate queue map operations from the set $\{\text{add}, \text{modify}, \text{delete}, \text{rename}\}$ to a pair of paths (the second member of the pair is empty for all operations but rename). Before processing, the change log has to be transformed into a change set, i. e. into three sets of paths P_a, P_m, P_d containing added, modified, and deleted paths, respectively, and a function $R : \text{path} \to \text{path}$ describing renamed paths. This can be done by Algorithm 1; its actual task is to keep the change set *consistent*, e. g. an added and subsequently removed file leaves no trace in the change set. For n entries in the change log out of which n_r entries are (directory) rename operations, the complexity of the algorithm is in $\mathcal{O}(n + nn_r)$. The worst case of $\mathcal{O}(n^2)$ is reached only if the change log contains $\Theta(n)$ rename operations *after* $\Theta(n)$ non-rename operations. Algorithm 1 is an online algorithm, i. e. it can process a change log stream. For final processing of the change set, R has to be applied before processing the path sets.

We will now discuss the following potential mechanisms to construct a file system change log.

Log-structured file system. A log-structured file system, such as the one developed for *Sprite* [RO91], does not organize the data in trees or inode lists. Instead, a stream of changes is stored. In order to harness the meta-data change log from the raw stream, one would need to parse the file system's internal format. If the written directory structure is not immutable, the file system has to perform a *log cleanup* at certain points in order to deflate the ever-growing stream to the actual volume of the data. Such cleanups might discard update information required for the transformation.

Intercepting system calls. User space interception of system calls, such as done by Parrot [TL05], operates on the wrong level, as changes to the software tree might be caused by operating system kernel tasks. In case of file system operations performed by a kernel task, the system call interception facility will not be notified.

A Fuse module. The "file system in user space" (Fuse) facility facilitates customized file systems by providing a minimal file system in kernel space that redirects all operations to a user space module [HS]. In contrast to intercepting system calls, the redirection here is performed by the kernel's file system layer. Channeling the file system calls through the user space inevitably comes with a performance penalty. For the workload of installing software, we see an overhead of approximately a factor of 2.

Algorithm 1: Construct a file system change set from a file system change log.

Input: Queue q with entries $e : \{\text{add}, \text{modify}, \text{delete}, \text{rename}\} \mapsto [\text{path}, \text{path}]$
Output: Path sets P_a, P_m, P_d, function $R : \text{path} \to \text{path}$
$P_a \leftarrow \varnothing$; $P_m \leftarrow \varnothing$; $P_d \leftarrow \varnothing$;
Empty queue of functions $R_q : \text{path} \to \text{path}$;
while q *is not empty* **do**
  $e \leftarrow q.\text{pop}()$;
  switch $e.\mathit{operation}$ **do**
    case *add*
      if $P_d \cap \{e.\mathit{path}[1]\} = \varnothing$ **then**
        $P_a \leftarrow P_a \cup \{e.\text{path}[1]\}$;
      else
        $P_m \leftarrow P_m \cup \{e.\text{path}[1]\}$;
        $P_d \leftarrow P_d \setminus e.\text{path}[1]$;
    case *modify*
      if $P_a \cap \{e.\mathit{path}[1]\} = \varnothing$ **then**
        $P_m \leftarrow P_m \cup \{e.\text{path}[1]\}$;
    case *delete*
      if $P_a \cap \{e.\mathit{path}[1]\} = \varnothing$ **then**
        $P_d \leftarrow P_d \cup \{e.\text{path}[1]\}$;
        $P_m \leftarrow P_m \setminus e.\text{path}[1]$;
      else
        $P_a \leftarrow P_a \setminus e.\text{path}[1]$;
    case *rename*
      $P_a \leftarrow P_a \setminus e.\text{path}[2]$;
      $P_m \leftarrow P_m \setminus e.\text{path}[2]$;
      $P_d \leftarrow P_d \setminus e.\text{path}[2]$;
      $\hat{R} \leftarrow p \mapsto$
      $\begin{cases} e.\text{path}[2]\text{"/"}p_\text{t} & \exists p_\text{t} \in \text{path} \cup \{\bot\} : e.\text{path}[1]\text{"/"}p_\text{t} = p\text{"/"} \\ \text{id}(p) & \text{else} \end{cases}$
      forall the $P \in \{P_a, P_m, P_d\}$ **do**
        forall the $p \in P$ **do**
          $P \leftarrow (P \setminus p) \cup \{\hat{R}(p)\}$;
      $R_q.\text{push}(\hat{R})$;
$R \leftarrow \text{id}$;
while R_q *is not empty* **do**
  $R \leftarrow \hat{R} \circ R_q.\text{pop}()$;
return P_a, P_m, P_d, R;

Kernel notifications. Kernel notification facilities, such as INOTIFY [Lov05] and SYSTEMTAP [EPC+05], can capture writing file system operations and expose them to user space. Such facilities work on events, i. e. monitored operations are tracked in a kernel space event buffer that is periodically flushed to the user space. Under heavy load, we observed that the event buffers overflow and events get lost.

Intercepting kernel file system calls. By intercepting not on the system call level but on the level of the file system layer in kernel, one avoids the loss of operations issued by kernel daemons. On recent UNIX flavors, the file system layer is object oriented and is called *Virtual File System Switch* (VFS). The virtual interface facilitates such interception, as it is not required to intercept individual file systems, but intercepting the abstract VFS suffices. A general purpose framework to do so was presented by Hrbata [Hrb05]. Figure 5.4 shows an approach to record file system changes based on that framework. For the workload at hand, we measure a performance penalty of less than 5 % caused by the creation of the change log.

A similar system can be achieved based on *stackable file systems* in kernel space, such as *Tracefs* [AWZ04]. This approach is comparable to the Fuse module, but without the need to pass operations through the user space. In contrast to intercepting VFS calls, the kernel data structures, such as inodes and dentries, have to be duplicated for the stacked file system.

Overlay File System

Overlay file systems combine several directories into one virtual file system that provides the view of merging these directories. These underlying directories are often called *branches*. Branches are ordered; in the case of operations on paths that exist in multiple branches, the branch selection is well-defined. By stacking a read-write branch on top of a read-only branch, overlay file systems can provide the illusion of a read-write file system for a read-only file system. All changes are in fact written to the read-write branch.

Preserving POSIX semantics of overlay file systems is non-trivial; the first fully functional implementation has been presented by Wright et al. [WDG+04]. By now, overlay file systems are well established for "Live CD" builders, which use a ram disk overlay on top of the read-only system partition in order to provide the illusion of a fully read-writable system.

In the same way, overlay file systems can be used to gather changes on the software directory tree. In this case, the read-only file system interface for the

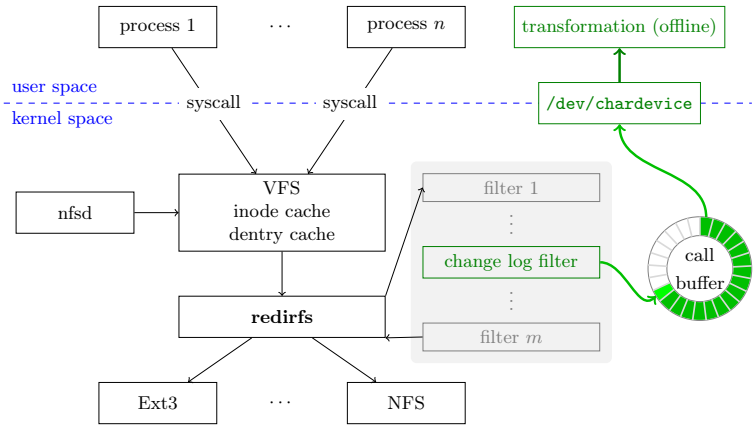


Figure 5.4: Tracking file system changes with *redirfs*. Changes to the software directory tree are exposed to a character device. During transformation into CAS or when the log buffer is full, writing VFS calls are blocked.

content-addressable storage is used in conjunction with a writable scratch area for changes. This option has the following advantages:

- CAS is the *only* storage format, i. e. we benefit from de-duplication on the master storage as well.
- The scratch area storing the changes is in fact already a change set.

However, this option also has the following disadvantages:

- The internal naming conventions of the overlay file system have to be recognized and parsed in order to process the change set.
- The rename operation on directories translates into a recursive remove and create, because internally the overlay file system crosses file system boundaries.

5.5 Confidentiality

Although at the moment virtually all distributed LHC experiment software is open source software, there might be use cases in the future for which software distribution is restricted by a license. Data confidentiality is also desirable for keeping event data, and thereby the derived physics results, confidential before publication.

5.5.1 Model

As security model, we assume a well-defined closed user group that is exclusively allowed to read from a file system. New members might join the group and existing members might leave the group, as is the case in a scientific collaboration. We fully trust the master storage and the worker nodes, i. e. protection is required only for the transport and the intermediate caches.

Encryption is widely used as a means to ensure data confidentiality in distributed file systems [KBC+00, RD01, Kut08]. Data are encrypted with a symmetric cipher. The key used for the cipher is encrypted with the public keys of the users of the closed user group. In the following, we will present an encryption scheme that fulfills the following objectives:

1. All files should carry an intrinsic name, i. e. as with content-addressable storage, files are immutable and file integrity should be verifiable by worker nodes using the file name.

2. Encryption should be optional in the sense that the master storage can turn it on and off, without the need to re-process all files.

5.5.2 Confidential CAS

We will first discuss two straight-forward schemes for encryption. We assume a file f, a symmetric cipher e, a secret key k, and a cryptographic hash function h. Option 1 is to set the file name to $h(e(k, f))$, i. e. we have content-addressable storage of encrypted files. Here, changing the encryption key involves full re-encryption of all files in order to retrieve the new content hashes. Option 2 inverses the order of hashing and encryption, i. e. we do not change the file name but encrypt files in a separate step. Obviously, changing keys does not modify the file name and the file is not immutable.

Instead, we propose to essentially keep the unencrypted file name but change it in a way that reflects the encryption key used. With "$\cdot$" being string concatenation, we propose the file name

$$h(f) \cdot h(k)$$

describing a file name with content hash $h(f)$ that has been encrypted using a key with the key hash $h(k)$. As h is a cryptographic hash, we will not reveal information about the content of the file nor about the secret key k. Furthermore, changing the key results in a new file name. We assume that changing keys is a rare operation required only when the membership of the

closed user group changes. Hence, during normal operation, we do not sacrifice cacheability of files. The master storage can be built in a way that encrypts a request for $h(f) \cdot h(k)$ on the fly (and does not deliver a request for $h(f)$, of course). Thus, we fulfill the objective of immediately turning encryption on and off.

Obviously, the worker nodes have to know the key k beforehand in order to request files. We can use $h(k)$ as a key identifier that is published in conjunction with the root hash of the file system. The master storage will deliver a request for $h(k)$ with the symmetric key encrypted by all public keys of the participants of the closed user group. Changing the closed group membership breaks down to publishing a new key identifier and encrypting the corresponding key with the public keys of the new members. Disregarding transient files on cache nodes, we thereby have immediate revocation of group membership.

6 Decentralized Memory Cache

We have seen that a local persistent data and meta-data cache decouples software execution from central servers. In a straight-forward form, each worker node contributes some scratch space of its hard disk for caching as well as the site providing a site-local cache. In this chapter, we will discuss how to replace the site-local cache server and the local disk caches altogether. Based on a distributed key value store, we will create a distributed and decentralized file system cache among the n worker nodes of the site. The decentralized cache automatically feeds itself from the master source. When the first worker node has fetched a file, it can share it among all other worker nodes in the cluster.

If data are stored in the *memory* of the worker nodes, we extend the caching to diskless server farms, as well as virtualized environments, where disk I/O has larger virtualization costs than network I/O [MA10]. Besides this, we reduce the overall space required to cache a volume of size s in a cluster of n worker nodes from $(n+1)s$ to cs, where the parameter c specifies the ability of the cache to share information (ideally $c = 1$). In order to do so, we have to make sure that *all* data are present in small chunks of not more than a few megabytes. To this end, we can assume chunking of larger files and fine-grained file catalogs down to the directory level (i. e. we sacrifice the meta-data pre-cache discussed in Section 5.2.3 to some extent).

Co-operative caching on the cluster scale has been studied before in cluster file systems. Some systems, for example SPRITE, use centralized file servers to maintain a consistent global view of the distributed cache [NWO88]. We refrain from such an approach because it is not fully decentralized and thus has all the scalability and reliability issues of centralized systems; SPRITE only reduces the load at the central server. Other systems, for example xFS, replicate the cache state globally [ADN+96]. Unlike our approach, xFS uses an additional indirection layer, the so-called *manager map*, which is distributed via network broadcasts. Such an approach is impractical in large, highly volatile environments because of its very high number of update messages. There are also approaches based on the dissemination of "hints" about each others cache content, either with individual hints per file [SH96], or using a cache summary that is only exchanged periodically [FCAB00]. Our approach, in

contrast, is opportunistic in the sense that it rapidly disseminates *areas of cache responsibility* of the complete file space.

6.1 Requirements

The feasibility of the approach relies on the efficiency of the distributed file system cache.

1. The distributed cache has to be decentralized, i. e. without any distinct service that needs to be provisioned.
2. The distributed cache has to spread both the number of requests and the size of the cached chunks equally among the worker nodes.
3. The distributed cache has to ensure proper sharing of files, i. e. once staged, files have to be indeed reused by other worker nodes.
4. As we assume volatile worker nodes, the distributed cache should not immediately re-balance its contents when worker nodes join or depart.

The distributed cache may assume that all worker nodes have an equal, very low communication latency, as is the case in local (sub-)networks. Nevertheless, the algorithm should work in open environments (e. g. Amazon EC2), i. e. there should be protection against spoofing. Likewise, the cache has to deal with corrupted data, i. e. any kind of failure of other worker nodes must not affect the principal functionality, but rather trigger the fetch of a chunk from the master source. As a result, we *do not need* timeliness and synchronization in the mutual updates.

When looking for chunks in a distributed cache, there are five kinds of *penalty* that a worker node potentially encounters:

1. If the chunk happens to be in the worker node's local memory cache, there is no penalty at all.
2. If the worker node asks another worker node that is able to serve the chunk from its memory cache, there is a latency in time of one round trip (RTT).
3. If the worker node has to ask $m > 1$ worker nodes before getting a positive response, and if these worker nodes forward the request, we have a latency of $(m + 1)\mathrm{RTT}/2$. If the asked worker nodes instead reply with a hint whom to ask next, we have $m \cdot \mathrm{RTT}$ latency. This is similar to the case of recursive and non-recursive DNS queries.

4. If the worker node gets a negative response, it has to fetch the chunk via the outside channel. This case can occur in combination with case 2 and case 3.

5. If a worker node silently disappears, its peer worker nodes can run into a network timeout. This case can occur in combination with case 2 and case 3.

The distributed cache should serve requests as in cases 1 and 2 with high probability.

6.2 Distributed Hash Tables

The distributed cache exposes a simple 2 function interface: `get(key)` $\rightarrow$ $\{$`data`$, \perp\}$ and `store(key, data)`. A system providing such an interface is known as a *key-value store*. In order to distribute the data over multiple machines, the key-value store uses internally a distributed hash table.

6.2.1 Key Space

Naturally, distributed hash tables use keys provided by content-addressable storage. However, a file system request consists of a path and a byte range, which first has to be translated into the corresponding CAS key. Hence, we have an alternative option of using the path and byte range as key. In order to avoid collisions, we have to mangle a repository ID and a file revision into the path. This option has the advantage that we can look up the CAS key while retrieving the file (we need the CAS key anyway in order to verify data integrity). The drawback is that we do not benefit from de-duplication, i. e. we store identical files multiple times. This spoils the cache efficiency. We waste storage and are unable to serve a request when we cache a file under a different file name or repository path. In the following, we choose to optimize for high hit rate and low memory consumption and assume the keys are derived from the chunk's content address.

6.2.2 Consistent Hashing

Current *distributed hash table* (DHT) implementations use consistent hashing [KLL+97] in order to partition keys (and data) over multiple nodes. Consistent hashing distributes data keys and node keys in the same key space. Each node's responsibility is bounded by the node IDs of its neighbors. While simple

and effective, the following problems arise when using consistent hashing in volatile environments.

It is unlikely that data keys are distributed equally over the nodes, especially when we consider hot spots in the requests; instead there is a high probability that a node is responsible for $\mathcal{O}(\log n/n)$ of the address space. There are a number of approaches that try to make all the nodes responsible for $1/n$ of the address space [GLS+04, BCM03, BKM05], for instance by assigning each node a couple of virtual servers.

Assuming full information is available about which are the participating nodes, uniform address space partition is trivial. However, a simple uniform distribution does not take into account that worker nodes may have different capabilities in terms of spare memory space and maximum request load. This might also change dynamically, i. e. instances that happen to be temporarily idle are able to take more requests than instances that have to process local file requests themselves.

Further studies take heterogeneous nodes and non-uniformly distributed data keys into account [GLS+04, ZH04]. Godfrey et al. introduce a certain number of *directories* and let the nodes report their load to them [GLS+04]. These directories then solve the load-balancing problem centrally. Another common approach for load-balancing is replication, in particular for keys with a high request rate.

When using consistent hashing, each join of a new node and each failure or leave of an existing node produces implicit or explicit load, since the areas of responsibility change. This effect is even increased with virtual servers. Högqvist et al. argue not to use consistent hashing, but instead distribute the keys based on global knowledge about each node's load [HHK+08].

Commonly used key-value stores usually aim at high availability, more complex requests, and persistent storage rather than being designed as a pure cache layer. Many of them, such as Dynamo [DHJ+07] and Cassandra [LM09], use consistent hashing. BigTable [CDG+06] relies on the Google File System. Scalaris [SSR08] uses a manually tuned distribution of keys to nodes, where the keys are distributed in lexicographical order around the DHT ring.

6.3 Self-Organizing DHT Algorithm

We are seeking for a DHT where a worker node chooses its key space responsibility itself, according to how much load it is able to withstand. Instead of solving the load-balancing problem centrally, we want the worker nodes to organize themselves. Furthermore, in cases where a large number of worker

nodes joins or leaves simultaneously, we want the responsibilities to be rather stable. In order to do so, we discuss a combination of consistent hashing with global knowledge about the worker nodes' load.

We assume a set of worker nodes that require certain content-addressable data chunks over time. Since we assume an outside channel from which we are able to retrieve chunks, new chunks are stored in the cache layer as a side effect of a cache miss. In order to exchange requests and data, the worker nodes send messages to each other (the internal interface of the DHT). The `request` message requests a certain chunk from another worker node; it is answered by a `deliver` message that has the chunk as payload. Figure 6.2 shows the implementation of the `get()` function and the `request` message handling. Note that in case of a cache miss, the asked worker node is responsible for retrieving the chunk via the outside channel.

Each worker node maintains its own memory cache for data chunks, using a least recently used replacement strategy. The worker nodes may choose the size of their caches independently of each other. Furthermore, each worker node maintains k *slots*, where for instance $k = 8$ or $k = 16$. A slot declares an area of responsibility within the key space. This is done by a prefix of arbitrary length, e. g. a slot r with prefix 0x123 is responsible for chunks with keys starting with 0x123. Other worker nodes will ask the worker node of slot r for data chunks with keys starting with 0x123 (providing there is no slot having a longer matching prefix for the chunk at hand). If there are multiple such worker nodes, they may ask any.

A slot can also be interpreted as a node in the complete key space tree (Figure 6.1). In this representation, a slot is responsible for data chunks with keys in the subtree that are rooted at the slot's node. In addition to its own slots, each worker node keeps the information about the other node's current slots.

In an ideal configuration with n uniform worker nodes, the sum of requests for each worker node's slots is the total number of requests divided by n. The system seeks to come close to an ideal configuration. To do so, each worker node may perform two basic operations on its slots:

Merge Two slots are merged in order to create a free slot. A worker node always merges the two slots that have the smallest key space distance. The prefix of the resulting slot is decreased to the common prefix of the original slots.

Split The slot's prefix length is increased by one bit, i. e. the worker node drops the right, or the left, subtree of the slot's key space. A worker node decides which subtree to drop according to the resulting coverage of the

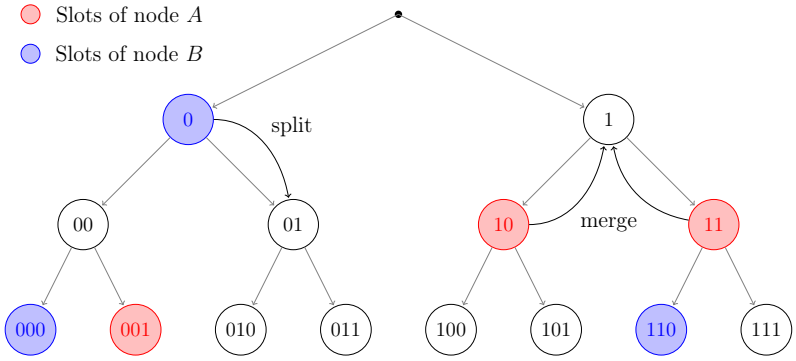


Figure 6.1: Example of a slot distribution over the key space 2^3. There are 2 worker nodes — A and B — having 3 slots each.

key space tree. It looks for the largest subtree that is covered by another slot and drops that one.

New data chunks having a key that matches none of the slot prefixes occupy a new slot with the chunk's key as prefix. A worker node merges two slots when all its slots are already occupied.

Even though the slots with the smallest key distance are merged, a merge might produce rather big areas of responsibility. Consider, for example, the first merge of a worker node with k slots: two slots having complete keys as prefixes are merged. If the keys are equally distributed, the resulting slot moves up from a leaf in the hash tree to a node at level $\log_2(k)$.

In order to restore the balance of the system, a worker node seeks to split such an oversized slot. We use the number of cache misses as an indicator for an oversized slot. Simulation experiments suggest that a small constant parameter turns out to give good results; we currently choose to split after 3 cache misses.

Even without cache misses, a slot might turn out to be *overloaded* in the sense that it has to serve more requests than other slots. This happens, for instance, for all the available slots at the moment when new nodes join. Let $r(s)$ be the number of requests for slot s. Since a split drops half of the key space in responsibility, we split s when

$$r(s) > 2\frac{\sum_{\text{all slots } i} r(s_i)}{nk}$$

Conversely, a similar approach could avoid *underloaded* slots by decreasing their prefixes. But according to simulations, such an approach decreases the quality of the algorithm. It produces a lot more churn in the slot responsibilities, without changing the overall hit-rate significantly.

6.3.1 Load Balancing

There is no particular load balancing mechanism included in our algorithm. Instead, load balancing is a natural side effect of the algorithm. We reduce the general load balancing problem to an adequate distribution of the keys. Each worker node can then control its load by choosing the size of its memory cache freely and independently from all other worker nodes. Splitting and merging keeps the requests per slot equally distributed. Adjusting the slot size according to the number of cache misses controls the worker node's load.

6.3.2 Simulation

In this section we evaluate the performance of the distributed cache given ideal information dissemination. The simulation does not take into account the latency amongst the worker nodes, nor the computation overhead. This is negligible because the computation is marginal and we assume high-throughput, low-latency communication between worker nodes. However, we do take into account the communication via the outside channel on a cache miss by a latency of 10 ms. We simulate up to 256 worker nodes representing a single local subnet. We will later see that in order to keep the number of state dissemination messages at a reasonable level, sites with more than 128 worker nodes should be partitioned. Such large sites can simply be split into multiple independent multicast groups, for instance based on the physical racks. This also reflects the assumption of low-latency, point-to-point traffic.

We inspect the traces of compiling the example collection of the ATLAS experiment software. The traces reflect 110 000 requests of 1100 distinct files, i. e. on average each file is opened 100 times. The requests are not equally distributed in time, but there are certain hot spots having the majority of requests. The traces reflect a running time of typically a couple of hours.

For the synthetically generated benchmark we chose the same ratio of requests and distinct files, but the requests are equally distributed over time. Furthermore, all the requests choose their files randomly and independently, i. e. in the synthetic trace there is no locality in the sequence of requests. Hence, this can be considered a worst case scenario for a cache layer. Since typically

Algorithm 2: get()

```
Input: Key k
Output: Data chunk
if there is a matching slot then
    s ← slot with a longest common prefix with k;
    w ← worker node of slot s;
    return w.request(k, s);
else
    c ← retrieve chunk from outside channel;
    Store c in worker node's cache;
    if worker node has a free slot then
        s ← free slot;
    else
        merge two of worker node's slots;
        s ← resulting free slot;
    prefix(s) ← k;
    return c;
```

Algorithm 3: request

```
Input: Key k, slot s
Output: Data chunk
if worker node has k in its cache then
    c ← corresponding data chunk
else
    c ← retrieve chunk from outside channel;
    Store c in worker node's cache;
    if number of cache misses > 3 then
        split(s);
if number of requests for s > 2 times slot request average then
    split(s);
sender.deliver(c)
```

Figure 6.2: Implementation of `get()` and `request`

the worker nodes in a cluster compile and execute the very same software during an analysis run, all the worker nodes are fed with the same traces. This holds for both the ATLAS traces and the synthetic traces.

Synchronization

When LHC computing jobs are executed on a cluster, the job scheduler usually starts worker nodes in a synchronized manner. Later the worker nodes tend to automatically stay synchronized because the first node to access a file must wait for the off-site source, and meanwhile, the slower nodes can catch up. In an implementation, high synchronization results in a large number of cache misses caused by the inevitable latency of the state dissemination. In order to handle such high synchronization, upon a cache miss an implementation should determine a worker node that is responsible for staging the file based on the file hash. This worker node acts as a proxy for the original worker node; it is able to collapse concurrent requests into a single outgoing request. This does not turn our DHT algorithm into traditional consistent hashing; it is used only to determine a proxy peer for staging. For the simulation of the ATLAS traces, we synchronize the worker nodes up to a random jitter in the range of 0 ms to 100 ms.

Efficiency

We define efficiency as the number of cache hits divided by the number of requests. Figures 6.3 and 6.4 show results from runs that assess the efficiency of our algorithm. The combined cache size is half of the size of the requested file set. We compare to two theoretical models, a large common LRU cache for all participating worker nodes (*idealistic case*) and small independent LRU caches for every worker node (*naive case*). The combined cache size of all cache models is the same. The efficiency gap of our algorithm compared to the idealistic case is below 10 % for the ATLAS traces. For the synthetic traces, the efficiency of our algorithm tends to the idealistic case with increasing number of slots. Naturally, with constant combined cache size and increasing number of worker nodes the efficiency of the naive case drops.

Redundancy

Figure 6.5 reflects the amount of redundancy in the cache layer. We measure the *false cache miss rate*, i. e. the percentage of cache misses that result from asking a wrong worker node, while there would have been another worker node

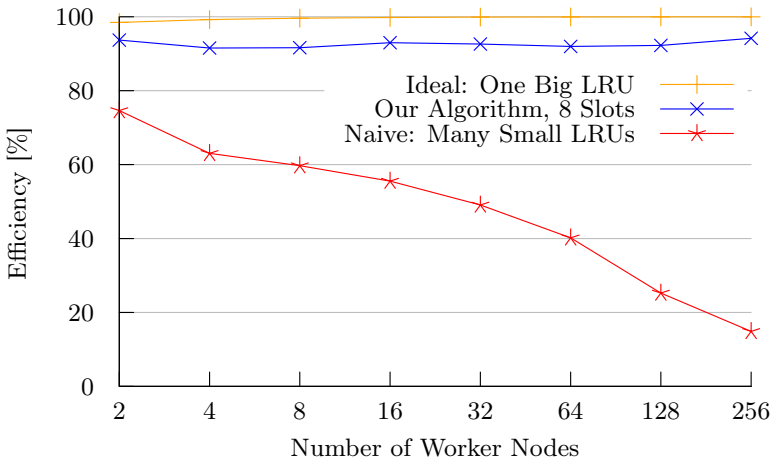


Figure 6.3: Efficiency comparison of requests for the ATLAS compile traces.

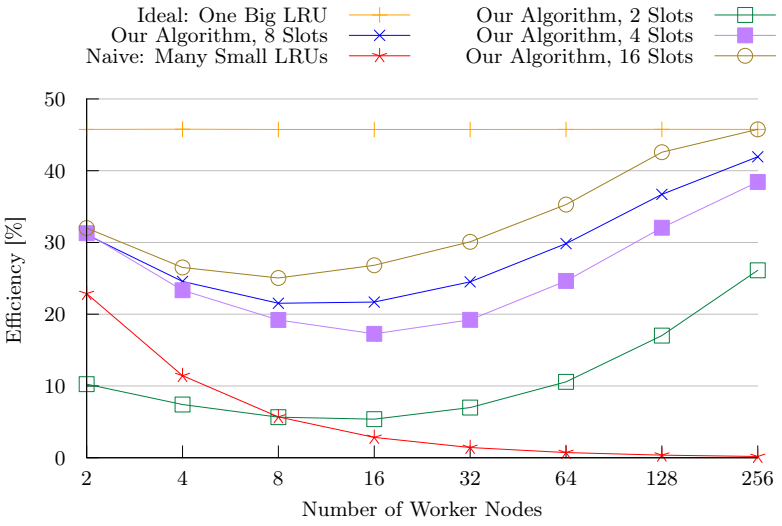


Figure 6.4: Efficiency comparison for synthetic traces and several slot configurations.

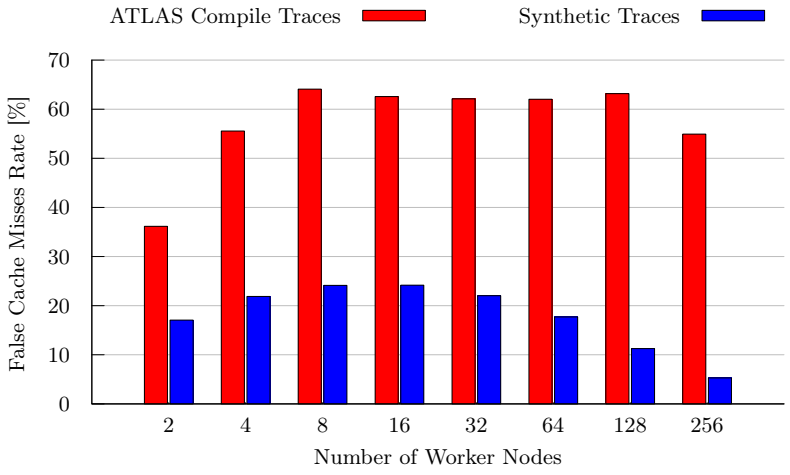


Figure 6.5: Fraction of false cache misses (i. e. a chunk is not found in the cache even though it is available) amongst all cache misses. The difference between ATLAS traces and the synthetic traces shows the impact of high worker node synchronization.

that contained the data. Imagine, in Figure 6.1 on page 80, worker node A with slots on 10 and 11 and worker node B with a slot on 1. A has data to the key 111 in its cache, whilst B has not. A merges its two slots, afterwards B splits to 11. From that point on, B is asked for the key 111 and produces false cache misses. Although Figure 6.5 suggests potential for improvement, note that at least for the ATLAS traces the overall number of cache misses is already less than 10 % of the number of requests. Furthermore, the comparison to the synthetic traces shows the impact of high worker node synchronization. This impact can be mitigated by the proposed implementation that collapses concurrent requests for the same file.

Load balancing

In this section, we examine the distribution of load onto the worker nodes. Since the requested chunks are small and often fit into one Ethernet frame, we measure load in terms of number of requests only.

Figure 6.6 shows the load distribution for 32 (64) homogeneous worker nodes with 4 (8) slots. For the ATLAS traces, half of the worker nodes show a deviation of around 10 % from the mean in most cases, with the minimum and

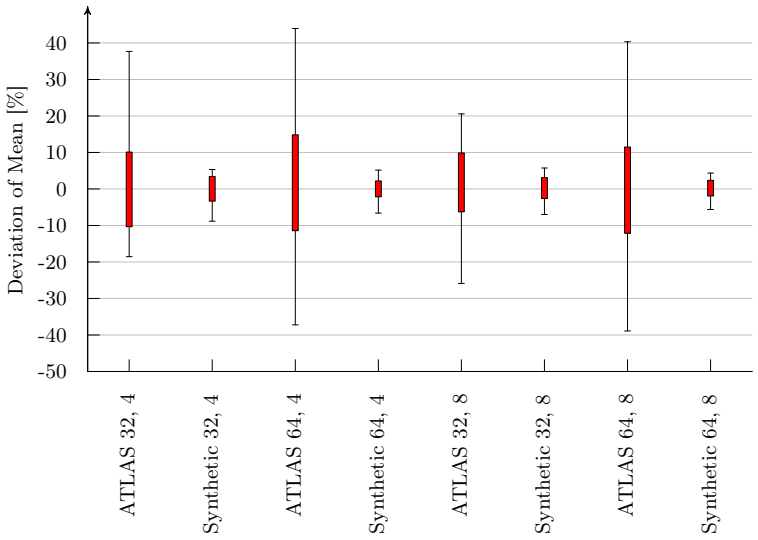


Figure 6.6: Distribution of requests for several configurations for homogeneous worker nodes as deviation from the average number of requests. The boxes show the 25 and 75 percentiles resp., while the bars show overall minimum and maximum.

maximum being as far as 40 % from the mean. Note that worker nodes still have the option to decrease their load by decreasing the number of slots.

Figure 6.7 shows the load distribution for 32 (64) heterogeneous worker nodes with 4 (8) slots. The worker nodes are divided into "strong nodes" and "weak nodes". Weak nodes have half the number of slots and half the size for their LRU cache as compared to strong nodes. We see that a worker node is able to reduce its load by decreasing these parameters. However, the figure does not suggest a sharp relationship between the amount of resources and the number of requests. So, in practice each worker node has to adjust its parameters dynamically. The efficiency of the cache was not significantly reduced by the simulated setting.

Figure 6.8 illustrates how the proposed algorithm behaves in case the number of available worker nodes changes suddenly. To this end, we let half of the worker nodes disappear in the middle of the simulation run for the "node loss" scenario. For the "node arrival" scenario, the number of already active worker nodes is doubled in the middle of the simulation run. As we see, all scenarios have about 90 % efficiency. Moreover, we see that losing worker nodes has a bigger impact on the efficiency than the sudden arrival of new worker nodes.

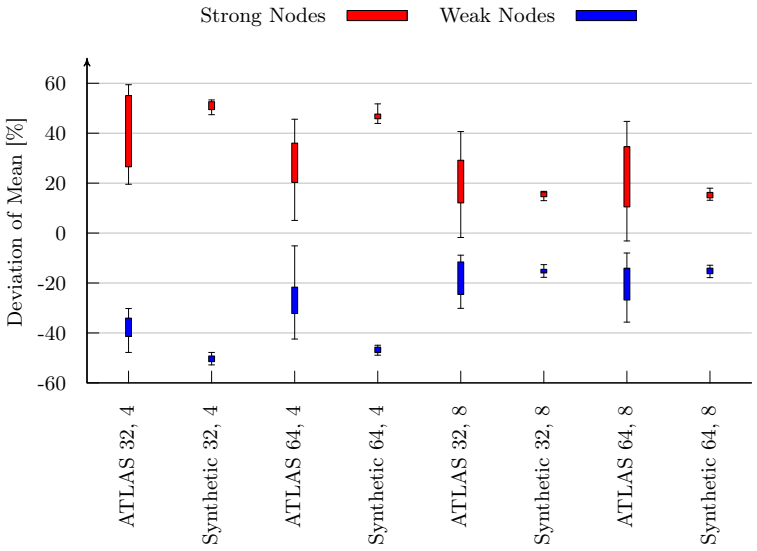


Figure 6.7: Distribution of requests for several configurations for heterogeneous worker nodes as deviation from the average number of requests. The configurations on the x-axis refer to the strong nodes. The boxes show the 25 and 75 percentiles resp., while the bars show overall minimum and maximum. Strong nodes have 2/3 of the resources. So each strong node should take 1/6 more load than the average.

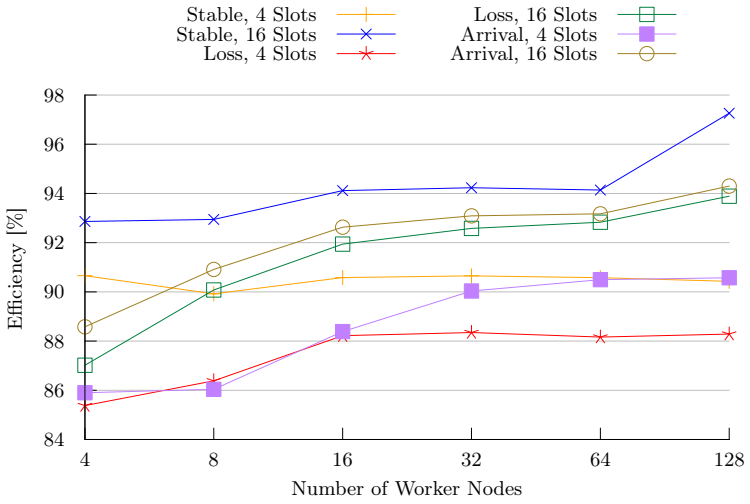


Figure 6.8: Efficiency in case of large and sudden changes in the number of available worker nodes. The configurations refer to the total number of worker nodes during a simulation run, i. e. before the loss of nodes and after the arrival of nodes.

6.4 State Dissemination

The system requires all worker nodes to share information about their respective cache content. The more timely that information, the better the cache efficiency. Link-layer multicast is an option, but the huge number of messages can drown the network, even though the messages themselves are tiny (4 bytes per cache slot). Also, many switches cannot handle multicast efficiently so that we often experience a high latency and packet loss rate for multicast traffic in clusters.

We distinguish two types of state information with different dissemination requirements:

- **Slot information** that describes the worker nodes' cache content by its slot prefixes and their number of requests, and
- **Presence information** that indicates which worker nodes are available.

The slot state of a worker node can change with every `open()` call. File system traces show that a typical analysis workflow issues about 100 calls per second. Hence, in a 1000 node cluster these calls result in 10^5 messages per second. We develop a gossip-based broadcast algorithm to handle this

avalanche of messages (see Section 6.4.1). It exploits the following properties of the slot state information:

- Inconsistencies in the slot information are tolerable.
- The per slot information is much smaller than the network's packet size.
- New slot state overwrites older state.

Maintaining the presence information requires our system to continuously monitor the worker nodes. The presence information itself can safely be exchanged via multicast, because changes in the set of available worker nodes are rare events, even in a large cluster. Even the simultaneous loss of a large number of worker nodes would not cause problems besides a brief, transient overload, which is tolerable in comparison to the loss of computing resources. This means that each time a new worker node arrives in the cluster, or when a worker node is about to leave the cluster, or when a worker node has been found "dead", the (detecting) worker node sends an according multicast notification message.

To avoid additional network traffic, we can piggy-back this protocol onto the slot information dissemination protocol (see Section 6.4.2). Nevertheless, the presence information dissemination can be used independently and in other contexts as well.

6.4.1 Slot state dissemination

The distributed memory cache needs to quickly spread the information about the slot state. Gossip protocols are a well established mechanism to do so [HHL88, BHO+99, EGHK03, EGKM04]. They also can combine messages from several nodes and thereby keep the number of messages low. But pure gossip protocols are inherently slow [Bir07], which contradicts our low-latency requirement. Therefore, we design a new gossip protocol, which exploits the fact that we have full presence information. Thus, we can disseminate the slot information according to a directed graph overlay (*dissemination graph*). Here, every worker node acts as source for a new slot state and as aggregator for the other worker nodes' slot state.

The dissemination graph needs to be a strongly connected graph, so the information sent by any worker node will reach all of the other worker nodes. To improve the robustness of the dissemination graph, we can even demand the dissemination graph to be k-connected with a large k, i. e. up to $k-1$ nodes may fail without disrupting the flow of information. The dissemination graph

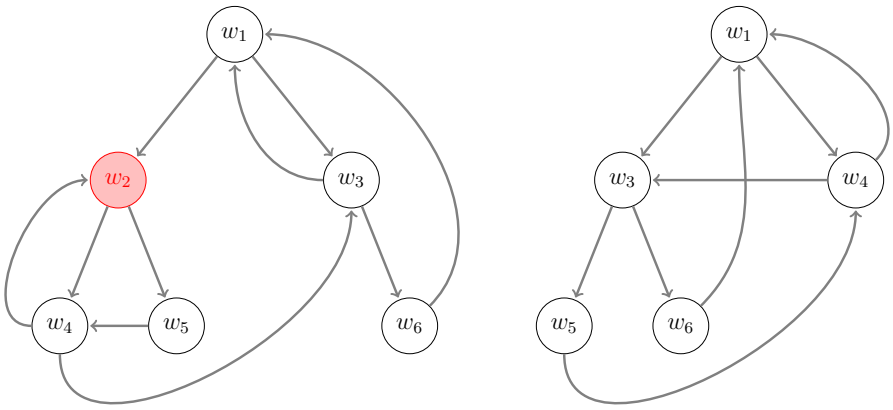


Figure 6.9: *Left hand side:* A 6 node LDI graph with fan-out 2. If worker node w_2 fails in the middle of state dissemination, worker nodes w_4 and w_5 become unreachable from worker nodes w_1, w_3, and w_6. *Right hand side:* As soon as worker node w_2 is detected as dead, the worker nodes re-organize the dissemination graph to an LDI graph with 5 nodes.

should have a small diameter, so that information *quickly* reaches all other nodes. At the same time, the number of messages each node needs to send (the *fan-out*) should be small[1]. Both requirements contradict each other. Given that we have a reliable presence information and can thus quickly reorganize the dissemination graph when a node fails, we chose to optimize for a small fan-out.

We use a so-called *low diameter interconnection* (LDI) graph [Mel07], which is constructed by connecting worker node r to the worker nodes $\{fr+k \mod n \mid 0 \leq k < f\}$. For $n = 2^m$ this yields a DeBruijn graph. The diameter of the LDI graph is logarithmic in the number of worker nodes. Figure 6.9 shows the $f = 2$ and $n = 6$ case.

When a worker node fails, the LDI graph changes (almost) entirely. We consider this an advantage because this property facilitates dead node detection (cf. Section 6.4.2). If, in another setting, more stability is required, the LDI could be constructed with virtual nodes. We do not use broadcast graphs such as hypercubes and tori [AGHK96, KK09] because of their additional overhead. In the following we consider a plain LDI with a fan-out of 2.

[1] A full mesh would correspond to link-layer broadcast. For n worker nodes the graph is $(n-1)$-connected and each worker node has fan-out $n-1$, but the nodes cannot aggregate the messages.

Unlike Melhem [Mel07], we disseminate slot information along all outgoing edges. Each worker node receives slot information from (up to) two worker nodes and enters it into its own global cache view. Then, it forwards (a part of) the combined information to its (up to) two downstream worker nodes. Each piece of the slot information is tagged with a version number, which is created by the worker node that holds the respective cache entry. The worker nodes keep track of the versions that they most recently forwarded to their downstream worker nodes. So they know which worker node needs which information.

In general, a worker node cannot immediately forward all the new slot information to both its downstream nodes because the snowball effect would double the message size in each step. Therefore, each worker node randomly picks some information and keeps the remainder for its next message. It preferably forwards different information to its two downstream worker nodes because this improves the overall freshness of the cache entries across the system. Eventually, some of the queued information is outdated by newly arriving information and can thus be discarded.

Compared to naive broadcast, we have two advantages: we combine many small messages into one large message that exploits the network's maximum transfer unit (MTU), and we automatically adapt to the overall system load because we aggregate more subsequent updates when many worker nodes quickly turn around their caches. We see a particularly large impact of the latter optimization, since when a worker node loads a new application it accesses many files within a short period of time. Aggregating the resulting updates differently along the different dissemination paths helps to balance the load. At the expense of more messages, a flow-control mechanism can be added to the dissemination, such as proposed by van Renesse et al. [vRDGT08].

6.4.2 Distributed Watchdogs

Presence information is exchanged via multicast. When a new worker node arrives in the cluster or when a worker node is about to leave the cluster it sends a corresponding notification message.

In a cloud system, worker nodes may leave ungracefully, for example, when their virtual machine happens to be suspended. Such worker nodes could cause large delays because other worker nodes might run into network timeouts when they try to retrieve files from a dead cache. In order to quickly detect dead worker nodes we build a watchdog algorithm, which exploits the fact that worker nodes regularly receive messages as part of the slot state dissemination protocol. If such a message fails to appear, the respective worker node can

be announced dead (via multicast), and the worker nodes can re-organize the cache.

The watchdog algorithm works as follows: If a worker node has no slot information to disseminate and has been idle for T_{max}, it must send an empty message. T_{max} determines the *heartbeat* of an idle cache. If a worker node did not receive a message within the time interval $T_{max} + T_{grace}$, it can conclude that its upstream worker node is dead and send an according announcement. The grace period T_{grace} allows for message delays. If the worker node was erroneously announced dead, it immediately responds with another multicast message. Otherwise, i. e. after another grace period T_{grace}, all worker nodes adapt their worker node table, which also implicitly creates a new LDI graph. A lost "I am alive" announcement could lead to an inconsistent graph; but a worker node that receives messages from a wrong upstream worker node can detect this inconsistency and trigger a retransmission of the missing announcement.

Watchdog Recovery Performance

We now study how quickly the worker nodes reflect a sudden loss of a potentially large number of arbitrary worker nodes. We measure the *recovery speed* in the number of heartbeats. To simplify the analysis, we assume a randomized setting in which the heartbeat messages are independent Bernoulli experiments. Furthermore, we generalize the analysis to a fan-out of k per worker node, i. e. each worker node randomly pings k other worker nodes. For piggy-backing the presence information dissemination on the slot state dissemination, we have $k = 2$.

Let n be the number of worker nodes, i. e. at the beginning each worker node has n entries in its table. At a certain point of time rn worker nodes fail, $0 < r < 1$. Now we have $(1 - r)n$ living worker nodes, each of them still having n records in their worker node table. Within the next couple of heartbeats, living worker nodes that happen to have a failed worker node as upstream worker node will announce its loss. Thereby, the number of undetected failed worker nodes decreases strictly monotonically. We give an estimation for the waiting time until all failed nodes are detected. To do so, we use an analysis similar to the *coupon collector's* problem [Fel68]. Our setting differs from the coupon collector's problem in the fact that there is not a single entity probing the peers, but all the living worker nodes probe concurrently. With more failing worker nodes, the number of probing worker nodes gets smaller. But the chance for each individual living worker node to randomly probe and detect a failed node increases.

We calculate the expected total number of probes $\mathbf{E}(P)$, where a probe corresponds to a received or missed dissemination message. Knowing that a heartbeat consists of $k(1-r)n$ probes, the expected number of heartbeats H is

$$\mathbf{E}(H) = \frac{1}{k(1-r)n}\,\mathbf{E}(P).$$

We calculate the probability for detecting the ith failed worker node d_i, provided that $i-1$ failed worker nodes are already detected:

$$d_i = \frac{rn-i+1}{n-i+1}.$$

The expected number of probes for a success w_i (the *waiting time*) has a geometric distribution, so $\mathbf{E}(w_i) = 1/d_i$. Hence we have

$$\begin{aligned}
\mathbf{E}(H) &= \frac{1}{k(1-r)n}(\mathbf{E}(w_1) + \cdots + \mathbf{E}(w_{rn})) \\
&= \frac{1}{k(1-r)n} \sum_{i=0}^{rn-1} \frac{n-i}{rn-i} \\
&\leq \frac{1}{k(1-r)} H_{rn} \approx \frac{\ln(rn)}{k(1-r)}
\end{aligned}$$

The same method estimates the variance using $\sigma^2(w_i) = (1-d_i)/d_i^2$:

$$\begin{aligned}
\sigma^2(H) &= \frac{1}{(k(1-r)n)^2} \left(\sigma^2(w_1) + \cdots + \sigma^2(w_{rn})\right) \\
&= \frac{1}{(k(1-r)n)^2} \sum_{i=0}^{rn-1} \frac{(1-r)n(n-i)}{(rn-i)^2} \\
&\leq \frac{1}{k^2(1-r)} \sum_{i=0}^{rn-1} \frac{1}{(rn-i)^2} \\
&\leq \frac{1}{k^2(1-r)} \frac{\pi^2}{6}
\end{aligned}$$

Overall we expect a very fast decrease of undetected failed worker nodes.

Figure 6.10 shows a simulation with 4096 worker nodes using the slot state dissemination graph (LDI graph with fan-out 2). The magenta line relates to the approximate closed formula for the waiting time. The closed formula for

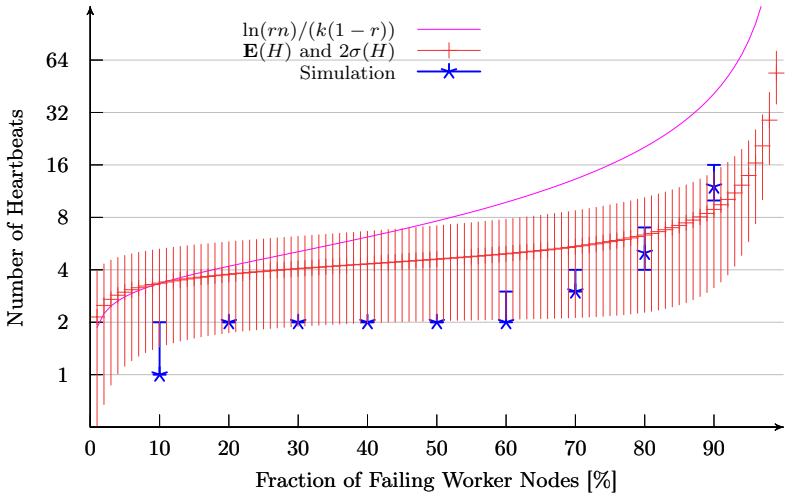


Figure 6.10: Dead worker node detection performance in number of heartbeats required to detect all failed worker nodes for 4096 nodes. For each failure rate, the failing worker nodes are chosen at random. Each simulation step is repeated 10^4 times.

the waiting time deviates from the exact formula with larger fractions of failing worker nodes and becomes an upper bound. The red line shows the recovery speed according to the theoretical model, here the exact value for the expected waiting time with error bars at 2σ. For lower fractions of failing worker nodes, the simulation is better than the theoretical model because the LDI graph ensures that each worker node is indeed probed by another worker node, as opposed to a truly random relationship. For a large fraction of failing worker nodes, the truly randomized probing is better because it ensures that there are no loops, i. e. every node is indeed probing *two* other nodes. Overall, even with 80 % of 4096 worker nodes failing at once, 4–5 heartbeats are sufficient to detect all dead nodes.

7 Performance Measurement and Comparison

I have implemented the methods described in my thesis as the *CernVM File System* (CernVM-FS). CernVM-FS is being used for delivery of software and conditions data by various LHC and other HEP collaborations on their distributed computing infrastructures. I have collected statistics from its deployment on more than 200 clusters representing approximately 40 000 worker nodes. At the time of writing, the hosted data comprises some 45 million files and directories together adding up to a some 3 TB.

7.1 Design and Implementation of the CernVM-FS

CernVM-FS is implemented as a *File System in User Space* (FUSE) [HS] module. Data and meta-data are accessed through the HTTP protocol [FGM+99] and are locally cached. Figure 7.1 shows how CernVM-FS interlocks with Fuse and a web server in order to deliver files. For transformation into content-addressable storage, dedicated installation machines at CERN are used. The transformation is implemented based on a file system change log created by a *redirfs* file system filter (cf. Section 5.4.1). The implementation uses multiple threads to calculate hashes and to compress files in parallel. File data and meta-data are stored DEFLATE compressed.

7.1.1 Caching

Two alternative facilities for local caching have been implemented. The first facility implements the distributed cache algorithm in conjunction with the dissemination protocol as presented in Chapter 6. It uses *memcached* [Fit04] as memory cache on each worker node. It is used for the evaluation of the decentralized memory cache (Section 7.2). The second facility uses LRU managed scratch space on a local hard disk. The vast majority of currently used LHC worker nodes have several gigabytes of unused scratch space. Furthermore, site-local web caches are already installed within WLCG for other purposes. Hence, this facility was preferred by the LHC computing community.

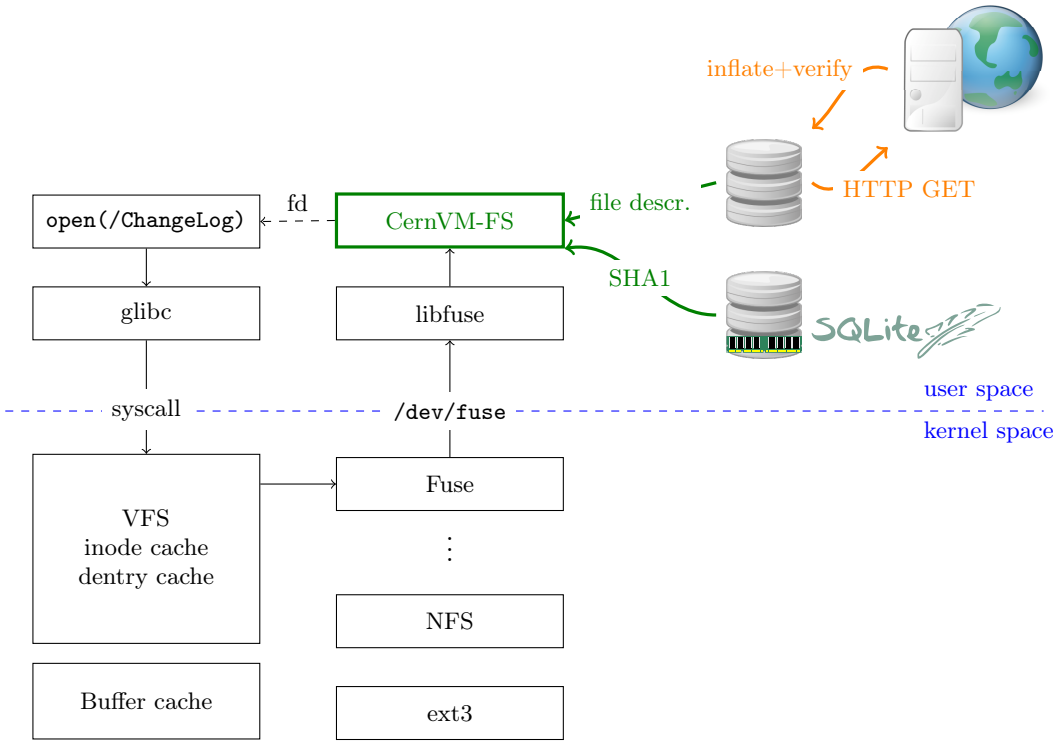


Figure 7.1: Process of opening a file. CernVM-FS resolves the name by means of an SQLite catalog, which is prepended by a memory cache. Downloaded files are verified against the cryptographic hash of the corresponding entry in the file catalog. The `read()` and the `stat()` system calls can be entirely served from the in-kernel file system buffers.

Special care has to be taken regarding the file system buffers of the operating system. When the contents of the file system change, the file system buffers of the operating system have to reflect the changes consistently. This means that either all files are served as of the old state or all files are served in the new state, but a mixing of both has to be prevented. CernVM-FS ensures such behavior by "draining out" the file system buffers. Before a new file system state is applied, buffering is temporarily disabled until all buffers have expired.

7.1.2 File Catalogs

The file meta-data are stored as *SQlite* databases [H+]. SQlite provides an embedded database engine. Meta-data requests can be easily expressed as SQL statements, as opposed to the use of hand-crafted formats. Furthermore, SQlite provides the means to restrict the memory usage simply by restricting the size of its page cache. This is in contrast to the GROW-FS implementation, for instance, that loads the entire meta-data into memory beforehand. Each SQlite database is stored as a single file. Like data, the file catalogs can be stored as content-addressable files.

Each database represents a distinct part of the directory subtree. The directory tree partitioning is done as described in Section 5.2.3, using magic files to indicate subtree roots. The partitioning can be changed at anytime by deleting or creating such magic files. Given this flexibility, release managers were quickly able to create a reasonable partitioning, and file catalogs very rarely exceed 50 MB.

The databases have a single table that represents the directory structure (Table 7.1). The layout of the tables is done as described in Section 5.2.3. As the distributed files are publicly readable, the ownership information can be neglected.

A file catalog contains a *time to live* (TTL) that advises the file system to check for a new version of the catalog when expired. Checking for a new catalog version takes place with the first file system operation on a CernVM-FS volume after the TTL has expired.

7.1.3 Data Access

In order to access non-cached data, CernVM-FS requires only outgoing HTTP connectivity to a web server and/or a web proxy server. The data access is stateless, which facilitates the migration of a worker node to a different network address. If available, however, CernVM-FS benefits from the HTTP/1.1 *keep-*

Field	Type
Path MD5 (Key)	128 bit Integer
Parent Path MD5	128 bit Integer
inode	64 bit Integer
SHA1 Content Hash	160 bit Integer
Size	64 bit Integer
Mode	32 bit Integer
Last Modified	Timestamp
Flags	8 bit Integer
Name	String
Symlink	String

Flags	Meaning
1	Directory
2	Transition point to a nested catalog
33	Root directory of a nested catalog
3	Regular file
4	Symbolic link

Table 7.1: Metadata information stored per directory entry. SQlite stores integers in a space-efficient manner by cropping leading zero bytes. On average, some 180 B per entry is stored. In comparison, the file headers of `tar` archives contain full paths and comprise 512 B each. The DEFLATE compression rate is around 65 %.

alive mode which keeps the TCP connection open. Parallel file requests are collapsed in order to prevent multiple downloads of the same file.

One might argue that the header overhead of the HTTP protocol represents a major performance drawback. Figure 7.2 shows that for the workload at hand there would be little impact of a smaller header. Assuming a maximum payload per IP packet of 1422 B, the majority of requested files is delivered in 1–2 IP packets. The fraction of files represented by the the intersection with the blue bars show the potential of transferring files in one packet instead of two, and in two packets instead of three, respectively, for an idealized empty header. This affects less than 10 % of all requested files.

7.1.4 Data Distribution

The WLCG distribution network consists of four public mirror web servers (*Stratum 1*) that are dedicated to CernVM-FS, as well as shared web caches at the sites. The network is shown in Figure 7.3. It is comparable to the Akamai architecture. The CERN Stratum 1 has currently 40–50 WLCG sites with a total number of 20 000–25 000 worker nodes connected to it. We see that about one out of 90 000 requests becomes corrupted in the web cache hierarchy, for instance by broken local site caches. These corrupted files are transparently refetched from Stratum 1 using the HTTP “no-cache” pragma.

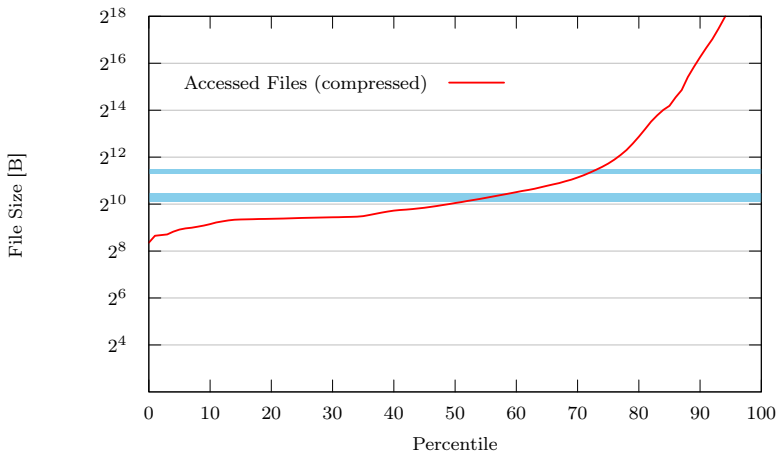


Figure 7.2: Cumulative size distribution of requested files. Files are DEFLATE compressed. The sample consists of operation traces over 1 month comprising 14 million requests. The blue bars show the range from 1092 B to 1422 B and from 2514 B to 2844 B, respectively. The height of the bars correspond to the observed average HTTP header size of 330 B.

The closest Stratum 1 server is determined by the file system through round trip time measurement. Fault-tolerance is obtained by horizontal scaling of the web caches in conjunction with fail-over logic built into the file system. Although not implemented, the number of Stratum 1 servers could be scaled by using peer-to-peer transport for the replication of files.

7.2 Evaluation of the Decentralized Memory Cache

In this section we evaluate the decentralized memory cache algorithm in conjunction with the state dissemination protocols, as described in Chapter 6. Timely state dissemination results in more accurate decisions of the cache algorithm; changes to a worker node's slot state result in state changes that have to be disseminated. Hence, the algorithm and the state dissemination protocols have to be evaluated in conjunction.

For evaluation, we use three benchmarks that are typical for LHC experiment software:

1. Software compilation is a typical first step in many WLCG jobs. We compile a Scientific Linux 5 kernel, which accesses 36 MB in 2400 files.

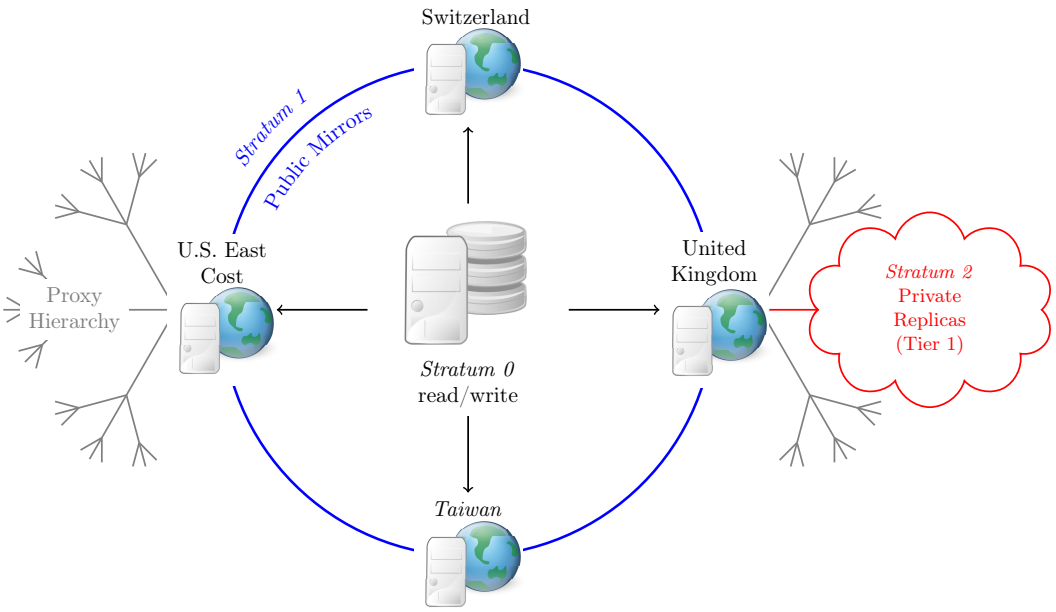


Figure 7.3: HTTP content delivery network: Replica servers are arranged in rings (Stratum 0 – Stratum 2). One protected r/w instance feeds several reliable, public, and globally distributed mirror servers. A distributed hierarchy of web caches fetches content from the closest public mirror server. The public mirror server can in turn be a master for private mirror servers, which might be demanded by large computing centers.

Often, files are opened multiple times, e. g. header files. On average each file is opened 200 times.

2. Setting up the runtime environment for an LHCb analysis job, i. e. selecting software package versions and resolving the library dependencies. (10 MB in 780 files, each file opened 12 times on average.)

3. An ALICE analysis job that compiles the job-specific physics code, links it against the ALICE experiment software framework, and runs the analysis. (90 MB in 350 files, each file opened twice on average.) This benchmark has a relatively low number of files. With 128 worker nodes not all of the 8 available worker node slots are used, even if each slot contains only a single file.

Benchmarks have been performed using 4–128 workers, which represents the maximum amount of resources CERN was able to spare for this evaluation.

Figure 7.4 shows how many times on average each file is requested from the master source. We see that from the Stratum 1 point of view, a cluster of 4–128 worker nodes behaves like 1–3 unclustered worker nodes. Also, we see that 128 nodes suffice to cache all the files that the workers request. If we further increase the number of worker nodes, the cache efficiency will not improve any more.

Figure 7.5 compares the cache maintenance traffic to a naive broadcast implementation. We assume that a naive broadcast will broadcast changes as soon as the slot information on a worker node changes. The system stabilizes at around 25–50 slot changes per worker node independently of the number of worker nodes. As all state changes have to be disseminated to all worker nodes, the number of messages is $\mathcal{O}(n^2)$. Large local networks can simply be split into multiple independent multicast groups, for instance based on the physical racks. This also reflects the assumption of low-latency point-to-point traffic. By combining messages in larger packets, we gain up to a factor of 4. The in-queue optimizations result in another 10 % – 20 % gain. Overall, the message dissemination traffic is below 5 % of the file transfer traffic.

7.3 Software Distribution Comparison

In the following benchmarks we use the ATLAS experiment software version 17.3.1, which comprises 9 GB, 240 000 files and 31 000 directories. Including the common dependencies of all releases, the software files comprise 21 GB, 370 000 files, and 41 000 directories. Additionally, we look at the distribution of a *patch release* that comprises 85 MB and 1400 files.

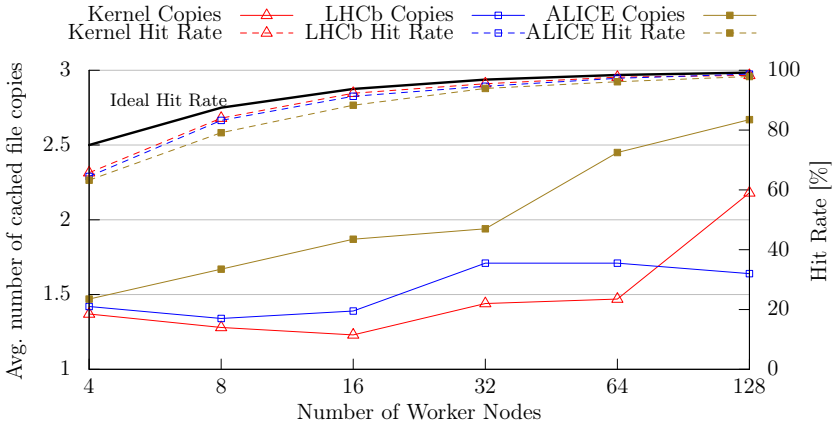


Figure 7.4: Redundancy factors of the distributed memory cache. The redundancy factor specifies how many times a file is requested from a Stratum 1 server. The redundancy factor in conjunction with the number of worker nodes specifies the cache hit rate. For n worker nodes the ideal cache hit rate is $(n-1)/n$.

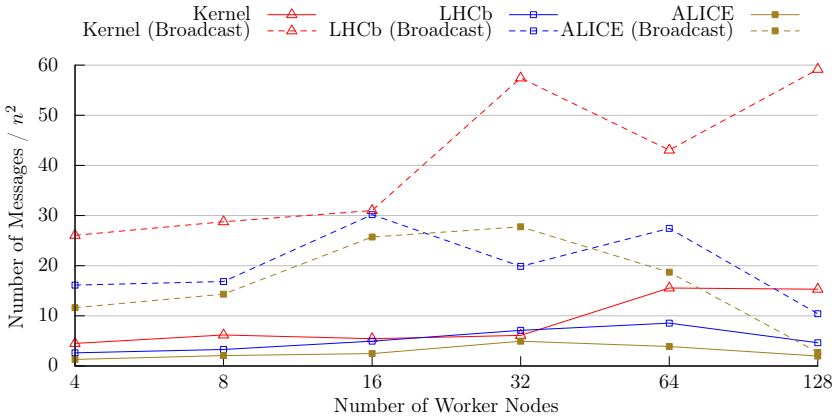


Figure 7.5: Number of messages divided by n^2 with n worker nodes. Our customized message dissemination requires several factors less network packets than a naive broadcast.

As typical workload for a software hosting file system, we use the bootstrap of the ATLAS event viewer because the event viewer depends on many different parts of the experiment software. This is a good example for the runtime behavior of the experiment software frameworks from a file system point of view. Overall, the benchmark comprises 4.6 million `stat()` calls and 4900 `open()` calls; it accesses 305 MB in 1480 files.

All benchmark nodes run Scientific Linux 5.7. They have two Xeon E5345 with 4 cores each at 2.3 GHz and 8 GB RAM. The hard disks perform a measured streaming rate of 61 MB/s for reading and 68 MB/s for writing. All nodes are connected to the same switch with Gigabit Ethernet. The measured round-trip latency between two nodes is 100 µs to 200 µs, the measured TCP throughput is 112 MB/s.

7.3.1 Turn-Around Time

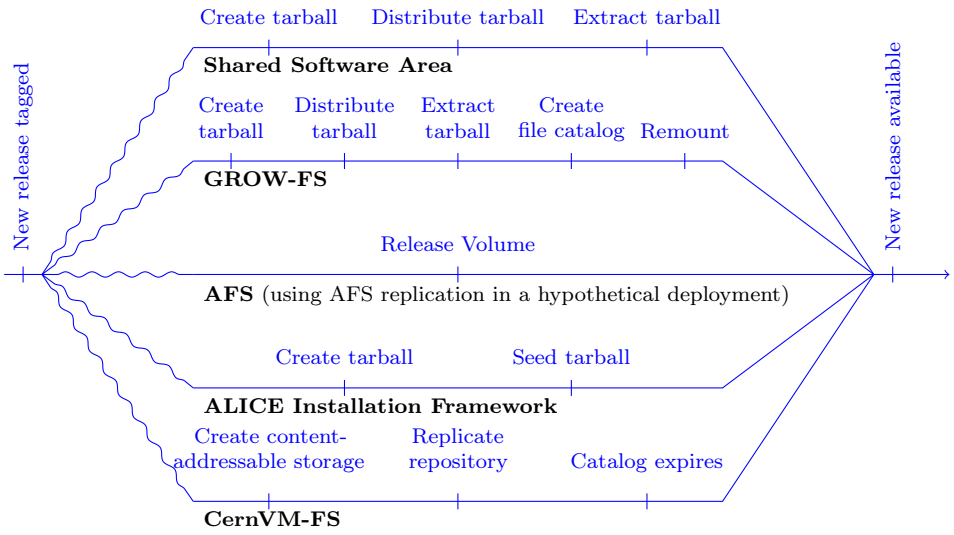
We define the *turn-around time* as the period of a new software release being built and tagged until it is available on all worker nodes. This period comprises the time it takes to publish a release and the time it takes to distribute a release, i. e.

$$t_{\text{turn-around}} = t_{\text{publish}} + t_{\text{distribute}}$$

A fast turn-around time speeds up the feed-back loop of verifying physics results after software modifications. It is important to have new releases available on *all* worker nodes. It also simplifies the job scheduler, as in this case jobs may safely land on any node of the distributed infrastructure. This does not necessarily mean that all newly published data have to be transferred to all worker nodes; data just have to be accessible in principle. Furthermore, we assume that the common dependencies of all software releases, such as system libraries and compilers, have already been installed. Figure 7.6 shows the necessary steps to make new releases available for several software distribution systems.

Shared Software Area

Using a shared software area on grid sites, $t_{\text{turn-around}}$ is the time to create a release tarball, distribute this tarball to all grid sites, and untar it to the shared software area (cf. Figure 7.6). The *gzip* compressed tarball of the release is 2.9 GB in size, the *gzip* compressed tarball of the patch release is 17 MB in size. Because the tarball contains many small files, the read and write throughput to the hard disks differs substantially as compared to streamed content. The throughput for the creation of the tarball was measured as 10 MB/s. The



ALICE	Duration [min]	
Step	Full Release	Patch
Create Taball	16	16
Seed Tarball	< 1	< 1
Sum	< 17	< 17

CernVM-FS	Duration [min]	
Step	Full Release	Patch
Create CAS	38	< 1
Replication	4	< 1
Catalog Expiry	15	15
Sum	57	< 16

Figure 7.6: Steps required to make new software releases available to worker nodes for several software distribution methods. For the ALICE installation framework and for CernVM-FS, the duration of the required steps in the case of publishing the full release and the patch release is shown. Similar calculations can be done for the remaining software distribution methods; in practice, however, these methods have a turn-around time of the order of days.

throughput for the extraction of the tarball was measured as 13 MB/s. For distribution, we assume optimistic values of 5 min delay for the grid scheduler to start the job on the sites and 100 Mbit/s throughput to all grid sites.

Overall, we have a minimum turn-around of 32 min = 15 min (create tarball) + 5 min (grid scheduler) + 4 min (distribute tarball) + 11 min (extract tarball) for the full software release and slightly more than 5 min for the patch release. In practice, the LHC experiments estimate a couple of hours to distribute a software release to the majority of grid sites. Due to installation failures on the remaining grid sites, however, the turn-around time is much higher. For ATLAS, failures occur typically at 10 % to 20 % of all grid sites. For CMS, failures occur typically at 3 % to 4 % of all grid sites. Failure reasons are, for instance, errors caused by the grid job scheduler, temporary network glitches on a grid site, or hard disk errors at the site. According to the experiments, virtually every time a software release is distributed manual failure recovery is required. Thus the real turn-around time can typically take a few days.

GROW-FS

The GROW-FS system is designed for local trusted clusters and has not been deployed on wide-area networks. Here, we will assume a setting comparable to shared software areas with GROW-FS installed on the grid sites instead of a cluster file system. In addition to the shared software areas, however, GROW-FS needs to create the file catalog of the new software release. The throughput for creating the GROW-FS catalog is 14 MB/s. So overall, we have a minimum turn-around of 42 min for the ATLAS software release.

In practice, however, this approach will be prone to the same failures as the normal grid installation jobs. In addition, GROW-FS lacks a possibility to change the contents of a mounted file system. Worker nodes have to drain out running jobs and remount in order to apply software updates. Thus the real turn-around time would be of the order of days.

AFS

AFS volumes can be replicated in order to scale the number of clients and sites. The mechanism of AFS read-only replicas would allow a central point for staging software updates and automatic wide-area distribution. In practice, however, the replica servers need to be constantly connected to the master replica. Manual recovery is required in the case a replica server is disconnected during the propagation of updates. Therefore, AFS replication cannot be

effectively used for software distribution in WLCG. Instead, AFS is used at some sites as a cluster file system in conjunction with grid installation jobs.

ALICE Installation Framework

To make new releases available to the ALICE Installation Framework, a compressed tarball of the release is created and added to the initial BitTorrent seeder (cf. Figure 7.6). The throughput of the seeding is I/O bound, i. e. it equals the read/write throughput of the hard disk. Overall, the turn-around for the ATLAS software release is slightly more than 15 min. The ALICE installation framework does not have a concept of patch releases; a patch release results in a new full release with the patches applied.

CernVM-FS

To make new releases available in CernVM-FS, the new files have to be transformed into content-addressable storage (cf. Figure 7.6). The measured throughput of the transformation and compression is 4 MB/s. The resulting compressed files are replicated to the Stratum 1 servers. We have measured at least 10 MB/s as replication throughput. Finally, the worker nodes have to pick up the new root hash. The root hash expiry period is set to 15 min. Overall, the turn-around for the full release is 57 min. For the patch release, the turn-over is slightly more than 15 min.

7.3.2 Network Load

In this section, we measure the incoming network traffic of a worker node and the number of outgoing requests of a worker node when the ATLAS jobs starts. These numbers represent the load caused by the worker node. We distinguish between intra-site load and the load from sites to central services at CERN.

For the intra-site load, we measure the incoming network traffic devoted to software on the worker nodes. Figure 7.7 shows the results for "cold caches". The results for NFS and AFS are representative for the shared software area, as these file systems cover more than 80 % of grid sites that are using shared software areas. The BitTorrent distribution used by the ALICE distribution framework requests chunks of 256 kB. As worker nodes are usually synchronized and require the same software release when a job is executed, these requests are not issued to a distinct worker node or service but they are equally distributed among the site's worker nodes. AFS, GROW-FS, and CernVM-FS provide persistent caching and do not cause any network traffic in case of warm caches.

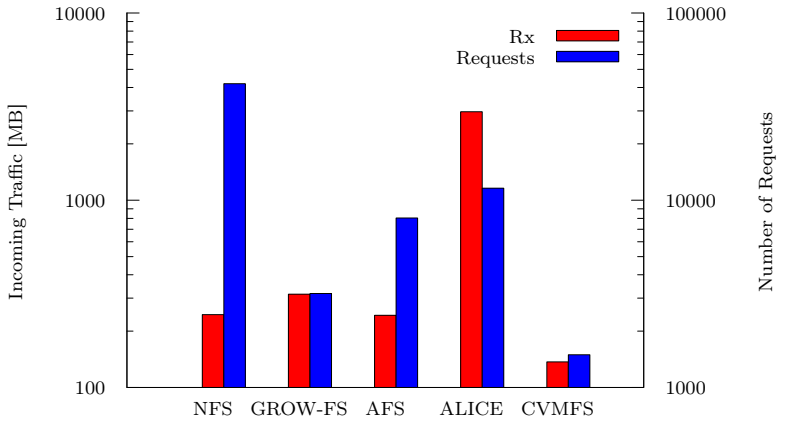


Figure 7.7: Incoming traffic and number of requests for running the ATLAS event viewer. NFS and AFS benefit from accessing only blocks of files. CernVM-FS benefits from compression and HTTP keep-alive connections. ALICE installation framework requests are usually distributed among the site's worker nodes.

For the ALICE installation framework and CernVM-FS, the load from the sites to CERN is difficult to predict because it is partially absorbed by site-local traffic and caches. The ALICE central seeder distributes on average some 30 MB/s. The CERN Stratum 1 distributes on average around 1 MB/s in 10 requests per second. While the number of worker nodes and sites connected to either of the services is comparable within a factor of two, the ALICE software repository is more than ten times smaller than the repositories served by CernVM-FS. It is safe to say that the load on central services is by factors lower for CernVM-FS than for the ALICE installation framework.

This is a result of the use of content-addressable storage. Independent jobs that use different experiment software frameworks still request identical files, e. g. the same version of ROOT. In this way, the CERN Tier 1 grid site observes, for a 10 GB local worker node cache, average hit rates of more than 99 % across all releases. The average worker node fetches only 2 kB/s from the network.

7.3.3 Runtime Penalty

In this section, we measure the extra time penalty Δt resulting from application binaries, search paths, and include paths residing on a network file system. The ALICE installation framework runs the software from local storage. However,

the framework has to *install* the software prior to running a job. So the extra time is given by the throughput of extracting the software tarball. Assuming warm caches, the extra time for GROW-FS, AFS, and CernVM-FS is only few seconds.

AFS and CernVM-FS provide secure remote access. In order to see the impact of wide area networks, we shape the outgoing network traffic using Linux's *tc*[1]. Thus, we artificially include latency and bandwidth restrictions. In case of cold caches, the results in Figure 7.8 show the impact of network latency and limited network bandwidth. Without having particularly tuned TCP parameters, we see the strong TCP throughput decrease caused by the high bandwidth-delay product. The rather linear growth of extra time can by explained by the fact that we request a sequence of many small files. In general, we see CernVM-FS performing consistently better than AFS in wide area networks. Significant performance losses due to low throughput are not seen until throughput rates are 15 Mbit/s to 18 Mbit/s.

7.3.4 Summary

With CernVM-FS, the delay from tagging a software release until it is usable in the distributed computing infrastructure is reduced from typically several days to less than an hour. For patch releases, the delay is reduced to a couple of minutes. Theoretically, the delay could be reduced even further by the ALICE installation framework or by AFS. However, the AFS replication only works for large computing centers that can guarantee a non-interrupted AFS service. In the case of the ALICE installation framework, the reduced delay is at the expense of a large overhead due to network load; it does not scale to the complexity of other LHC experiment software frameworks.

The network load for both intra-site traffic and central services is reduced by several factors with CernVM-FS. CernVM-FS allows effective wide-area access to software, which facilitates the use of opportunistic and volatile resources. The overhead of running software from CernVM-FS compared to having it locally installed is negligible. Due to the local caching, the time to execute jobs does not increase with more worker nodes on a site. While the ALICE installation framework has no penalty at all once a computing job starts, there is a considerable overhead *before* a job can start, which is independent of the job itself.

The comparison shows that the ideas presented in this thesis are applicable to a large-scale production system. Furthermore, volatile and opportunistic

[1] See `http://lartc.org`

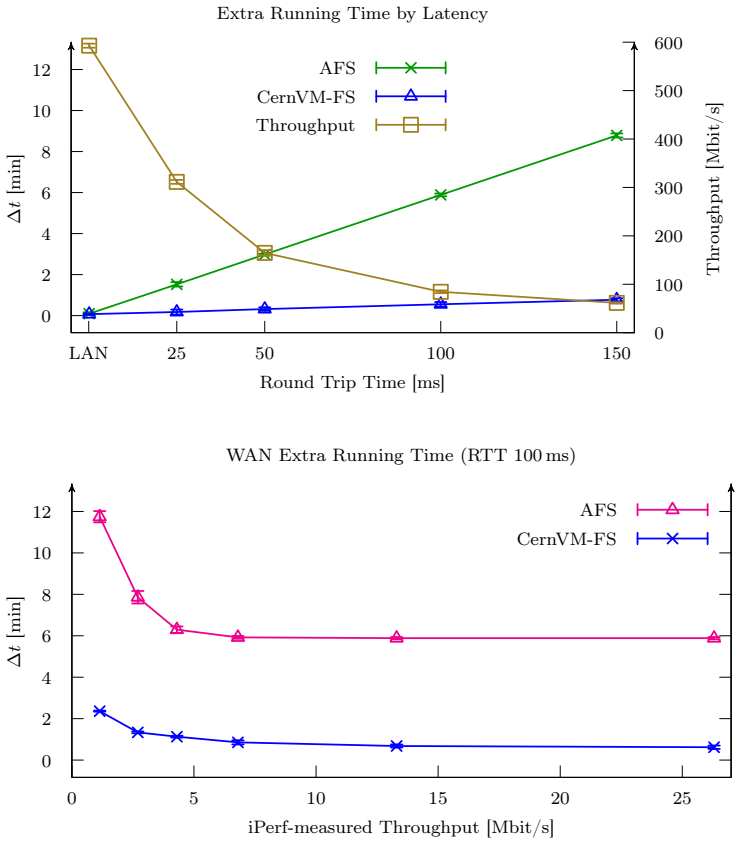


Figure 7.8: Time penalty as a function of latency and bandwidth comparing AFS and CernVM-FS to local storage for running the ROOT benchmark *stressHepix*. With standard window sizes of 16 KiB the TCP throughput is directly affected by latency. A round trip time of 100 ms corresponds to observed parameters between a worker node running on Amazon EC2 (U.S. East Coast) accessing services located at CERN.

resources are enabled to efficiently access software remotely. Compared to the state of the art, there are significant improvements for the use case of large collaborations needing to deploy their software on a world-wide distributed computing infrastructure.

8 Conclusion

During the development and the first exploitation phase of the Worldwide LHC Computing Grid, research was mainly focused on the distributed data management at the petabyte scale. Efficient access to the experiment data, however, is pointless without the ability to analyze and interpret these data. This data processing ability is provided by the runtime environment, involving the use of complex and frequently changing applications with many dependencies down to the operating system level.

So far, it was unknown how to provide a homogeneous runtime environment in a highly distributed and heterogeneous computing infrastructure. In order to survive, experiments developed ad-hoc solutions for software distribution based on standard technologies such as packet managers and cluster file systems. In practice, these solutions turned out to be error-prone, maintenance was man-power intensive, and reproducibility of data analyses in the long term was severely impacted. Certain grid sites had to artificially restrict the number of concurrent computing jobs in order to not overload their software distribution service. At the scale of LHC experiment software, existing software distribution methods complicated the use of opportunistic and volatile resources provided by cloud infrastructures.

The thesis studied a systematic approach to the problem at hand by providing a global and uniform file system access to the experiment software. LHC experiment software was analyzed from a file system point of view. The analysis revealed substantial differences compared to the average workload of distributed file systems. The biggest problem for general-purpose distributed file systems is the uncommon file size distribution with only few kilobytes per file on average. In conjunction with access patterns dominated by lookup requests, the main workload is on meta-data operations.

To improve the situation, the thesis proposed the use of content-addressable storage as a means to store, distribute, and cache file data and meta-data. The thesis contributed new ideas on how to efficiently interface large directory trees with content-addressable storage, which has been a necessary pre-condition for the feasibility of the approach. In return, content-addressable storage allows for stateless storage and cache services. In this way, the caching strategy of the

file system can be easily adapted to the respective computing environment at hand.

In order to optimize file system caching on the cluster scale, the thesis presented a fully decentralized and distributed file system memory cache. In contrast to existing distributed caches, the design and analysis of the presented algorithm takes into account the volatility of resources in virtualized computing centers.

The presented methods have passed the reality check [L+11] and are being adopted by several LHC experiment collaborations and other high energy physics experiments. They lay the foundation for decentralized data processing in the future context of cloud and volunteer computing [CAA+11, H+11].

Bibliography

[AA06] K. Adams and O. Agesen. *A Comparison of Software and Hardware Techniques for x86 Virtualization.* ACM SIGPLAN Notices, **41**(11) pp. 2–13, 2006. On pages 30 and 32.

[AAA+03] S. Agostinelli, J. Allison, K. Amako, et al. *Geant4 - A Simulation Toolkit.* Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment A, **506**(3) pp. 250–303, 2003. On page 41.

[ADN+96] T. E. Anderson, M. D. Dahlin, J. M. Neefe, et al. *Serverless Network File Systems.* ACM Transactions on Computer Systems, **14**(1) pp. 41–79, 1996. On page 75.

[Adv05] Advanced Micro Devices. *AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual*, 2005. On page 31.

[Adv08] Advanced Micro Devices. *AMD-V Nested Paging*, 2008. On page 32.

[Adv09] Advanced Micro Devices. *AMD I/O Virtualization Technology (IOMMU) Specification*, 2009. On page 32.

[AFM05] S. Annapureddy, M. J. Freedman, and D. Mazières. *Shark: Scaling File Servers via Cooperative Caching.* In *Proc. of the 2nd Symposium on Networked Systems Design and Implementation (NSDI'05)*, pp. 129–142, 2005. On pages 3 and 53.

[AGHK96] R. Ahlswede, L. Gargano, H. Haroutunian, et al. *Fault-Tolerant Minimum Broadcast Networks.* Networks, **27**(4) pp. 293–308, 1996. On page 90.

[And04] D. P. Anderson. *BOINC: A System for Public-Resource Computing and Storage.* In *Proc. 5th IEEE/ACM Int. Workshop on Grid Computing*, pp. 4–10, 2004. On page 26.

[ASBB+11] C. Aguado-Sanchez, J. Blomer, P. Buncic, et al. *Studying ROOT I/O performance with PROOF-Lite.* In *Proc. of the 18th int. conf. on Computing in High Energy Physics (CHEP'10)*, 2011. To appear. On page 33.

[ATL05] ATLAS Computing Group. *ATLAS Computing: Technical Design Report.* Tech. Rep. CERN-LHCC-2005-022, CERN, 2005. On page 41.

[AWZ04] A. Aranya, C. P. Wright, and E. Zadok. *Tracefs: A File System to Trace Them All.* In *Proc. of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)*, pp. 129–143, 2004. On page 71.

[B+05] I. Bird et al. *LHC Computing Grid: Technical Design Report.* Tech. Rep. LCG-TDR-001, CERN, 2005. On pages 1 and 19.

[BBB+01] G. Barrand, I. Belyaev, P. Binko, et al. *GAUDI: A software architecture and framework for building HEP data processing applications.* Computer Physics Communications, **140** pp. 45–55, 2001. On page 41.

[BBC+91] J. P. Baud, J. J. Bunn, F. Cane, et al. *SHIFT: the scalable heterogeneous integrated facility for HEP computing.* In K. Bos and B. van Eijk (eds.), *Proc. of the Workshop on detector and event simulation in high energy physics*, pp. 41–56, 1991. On page 17.

[BBC+03] R. Brun, P. Buncic, F. Carminati, et al. *Computing in ALICE.* Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, **502** pp. 339–346, 2003. On page 41.

[BBM+87] R. Brun, F. Bruyant, M. Maire, et al. *GEANT3.* Tech. Rep. DD/EE/84-1, CERN, 1987. On page 41.

[BCM03] J. Byers, J. Considine, and M. Mitzenmacher. *Simple Load Balancing for Distributed Hash Tables.* In *Peer-to-Peer Systems II.* Springer, 2003. On page 78.

[BDF+03] P. Barham, B. Dragovic, K. Fraser, et al. *Xen and the art of virtualization.* Operating Systems Review, **37**(5) pp. 164–177, 2003. On page 31.

[BHO+99] K. P. Birman, M. Hayden, O. Ozkasap, et al. *Bimodal Multicast.* ACM Transactions on Computer Systems, **17**(2) pp. 41–88, 1999. On page 89.

[Bir07] K. Birman. *The Promise, and Limitations, of Gossip Protocols.* ACM SIGOPS Operating Systems Review, **41**(5) pp. 8–13, 2007. On page 89.

[Bir10] I. Bird. *WLCG: Progress and Challenges*, 2010. Plenary talk at the 18th Int. Conf. on Computing in High Energy Physics. On pages 15 and 19.

[BKM05] M. Bienkowski, M. Korzeniowski, and F. Meyer auf der Heide. *Dynamic Load Balancing in Distributed Hash Tables.* In *Peer-to-Peer Systems IV.* Springer, 2005. On page 78.

[BMM05] E.-J. Bos, E. Martelli, and P. Moroni. *LHC Tier-0 to Tier-1 High-Level Network Architecture.* Tech. rep., CERN, 2005. On page 20.

[BMP02] M. Beck, T. Moore, and J. S. Plank. *An End-to-End Approach to Globally Scalable Network Storage.* ACM SIGCOMM Computer Communication Review, **32**(4) pp. 339–346, 2002. On pages 3 and 60.

[BR97] R. Brun and F. Rademakers. *ROOT - An Object Oriented Data Analysis Framework.* Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment A, **389** pp. 81–86, 1997. On page 40.

[Bra98] P. J. Braam. *The Coda Distributed File System.* Linux Journal, **50** pp. 46–51, 1998. On page 53.

[Bru04] D. L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* Ph.D. thesis, Massachusetts Institute of Technology, 2004. On page 30.

[BSNP95] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk. *Cryptographic Hash Functions: A Survey.* Tech. rep., University of Wollongong, 1995. On page 60.

[CAA+11] A. Charbonneau, A. Agarwal, M. Anderson, et al. *Data Intensive High Energy Physics Analysis in a Distributed Cloud.* In *Proc. of*

the Int. Conf. on High Performance Computing and Simulation (HPCS'11), 2011. On page 112.

[Cas10] T. Cass. *Trusted Virtual Machine Images: A Step Towards Cloud Computing for HEP?*, 2010. Talk at the 18th Int. Conf. on Computing in High Energy Physics (CHEP'10). On page 35.

[CDG+06] F. Chang, J. Dean, S. Ghemawat, et al. *Bigtable: A Distributed Storage System for Structured Data.* In *Proc. of the 7th Conf. on USENIX Symposium on Operating Systems Design and Implementation*, pp. 205–218, 2006. On page 78.

[CGL+10] G. Compostella, S. P. Griso, D. Lucchesi, et al. *CDF software distribution on the Grid using Parrot.* Journal of Physics: Conference Series, **219**, 2010. On pages 3, 55, and 63.

[CGM+10] T. Cass, S. Goasguen, B. Moreira, et al. *CERN's Virtual Batch Farm.* In *Proc. of the 2nd Cloud Computing International Conference*, pp. 21–32, 2010. On page 27.

[Cha01] F. W. Chang. *Using speculative execution to automatically hide I/O latency.* Ph.D. thesis, Carnegie Mellon University, 2001. On page 65.

[CJM+08] M. Coppola, Y. Jégou, B. Matthews, et al. *Virtual Organization Support within a Grid-Wide Operating System.* Internet Computing, **12**(2) pp. 20–28, 2008. On page 38.

[Coh03] B. Cohen. *Incentives Build Robustness in BitTorrent.* In *1st Workshop on Economics of Peer-to-Peer Systems*, 2003. On page 54.

[DÖ3] D. Düllmann. *The LCG POOL Project: General Overview and Project Structure*, 2003. `arXiv:physics/0306129`. On page 41.

[DEFH05] A. Dorigo, P. Elmer, F. Furano, et al. *XROOTD - A highly scalable architecture for data access.* WSEAS Transactions on Computers, **4**(4) pp. 348–353, 2005. On page 1.

[DHJ+07] G. DeCandia, D. Hastorun, M. Jampani, et al. *Dynamo: Amazon's Highly Available Key-value Store.* ACM SIGOPS Operating Systems Review, **41**(6) pp. 205–220, 2007. On page 78.

[DKK+01] F. Dabek, M. F. Kaashoek, D. Karger, et al. *Wide-area cooperative storage with CFS.* ACM SIGOPS Operating Systems Review, **35**(5) pp. 202–215, 2001. On pages 1, 60, and 61.

[EB08] L. Evans and P. Bryant. *LHC Machine.* Journal of Instrumentation, **3**, 2008. On pages 1 and 8.

[Edw85] H. T. Edwards. *The Tevatron Energy Doubler: A Superconducting Accelerator.* Annual Review of Nuclear and Particle Science, **35** pp. 605–660, 1985. On page 9.

[EGHK03] P. T. Eugster, R. Guerraoui, S. B. Handurukande, et al. *Lightweight Probabilistic Broadcast.* ACM Transactions on Computer Systems, **21**(4) pp. 341–374, 2003. On page 89.

[EGKM04] P. Eugster, R. Guerraoui, A.-M. Kermarrec, et al. *Epidemic Information Dissemination in Distributed Systems.* Computer, **37**(5) pp. 60–67, 2004. On page 89.

[EK02] K. M. Evans and G. H. Kuenning. *A Study of Irregularities in File-Size Distributions.* In *Int. Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'02)*, 2002. On page 43.

[EPC+05] F. C. Eigler, V. Prasad, W. Cohen, et al. *Architecture of SystemTAP: a Linux trace/probe tool.* Tech. rep., `http://sourceware.org/systemtap`, 2005. On page 71.

[FCAB00] L. Fan, P. Cao, J. Almeida, et al. *Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol.* IEEE/ACM Transactions on Networking, **8**(3) pp. 281–293, 2000. On pages 4 and 75.

[Fel68] W. Feller. *An Introduction to Probability Theory and Its Applications*, vol. 1. Wiley, 1968. On pages 62 and 92.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, et al. *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616, Internet Engineering Task Force, 1999. URL `http://www.rfc-editor.org/rfc/rfc2616.txt`. On page 95.

[Fit04] B. Fitzpatrick. *Distributed Caching with Memcached.* Linux Journal, **2004**(124), 2004. On page 95.

[FK97] I. Foster and C. Kesselman. *Globus: A Metacomputing Infrastructure Toolkit.* Int. Journal of Supercomputer Applications, **11**(2) pp. 115–128, 1997. On pages 18 and 41.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. *The Anatomy of the Grid.* Int. Journal of High Performance Applications, **15**(3), 2001. On pages 1 and 18.

[Fos02] I. Foster. *What is the Grid? A Thee Point Checklist.* GRID today, **1**(6) pp. 22–25, 2002. On page 18.

[FSFR05] A. Ferrari, P. R. Sala, A. Fassò, et al. *FLUKA: A Multi-Particle Transport Code.* Tech. Rep. SLAC-R-773, Stanford Linear Accelerator Center (SLAC), 2005. On page 38.

[Fur11] F. Furano. *Data management in HEP: An approach.* The European Physical Journal Plus, **126**(1), 2011. On page 1.

[GA94] J. Griffioen and R. Appleton. *Reducing File System Latency using a Predictive Approach.* In *Proc. of the USENIX Summer Technical Conference*, 1994. On page 65.

[GGL03] S. Ghemawat, H. Gobioff, and S.-T. Leung. *The Google File System.* ACM SIGOPS Operating Systems Review, **37**(5) pp. 29–43, 2003. On page 1.

[GLS+04] B. Godfrey, K. Lakshminarayanan, S. Surana, et al. *Load Balancing in Dynamic Structured P2P Systems.* IEEE INFOCOM, **4** pp. 2253–2262, 2004. On pages 4 and 78.

[H+] R. Hipp et al. *SQLite.* `http://www.sqlite.org`. On page 97.

[H+11] A. Harutyunyan et al. *CernVM CoPilot: a Framework for Orchestrating Virtual Machines Running Applications of LHC Experiments on the Cloud.* In *Proc. of the 18th Int. Conference on Computing in High Energy Physics (CHEP'10)*, 2011. To appear. On page 112.

[Har10] A. Harutyunyan. *Development of Resource Sharing System Components for AliEn Grid Infrastructure.* Ph.D. thesis, State Engineering University of Armenia, 2010. On page 33.

[HHK+08] M. Högqvist, S. Haridi, N. Kruber, et al. *Using Global Information for Load Balancing in DHTs.* In *Proceedings of the 2nd IEEE Int.*

Conf. on Self-Adaptive and Self-Organizing Systems Workshops (SASOW'08), pp. 236–241, 2008. On page 78.

[HHL88] S. M. Hedetniemi, S. T. Hedetniemi, and A. L. Liestman. *A survey of gossiping and broadcasting in communication networks.* Networks, **18**(4) pp. 319–349, 1988. On page 89.

[Hig64] P. W. Higgs. *Broken Symmetries and the Masses of Gauge Bosons.* Physical Review Letters, **13**(16) pp. 508–509, 1964. On page 7.

[HLPW06] A. Heavey, K. Lassila-Perini, and J. Williams. *The CMS Offline Workbook.* Tech. rep., CERN, 2006. On page 41.

[HLST11] M. Hogan, F. Liu, A. Sokol, et al. *NIST Cloud Computing Standards Roadmap.* Tech. Rep. NIST CCSRWG – 092, NIST, 2011. On page 24.

[HMR09] J. Harvey, P. Mato, and L. Robertson. *The Large Hadron Collider: a Marvel of Technology*, chap. 5.6 (LHC Data Analysis and the Grid). EPFL Press, 2009. On pages 14 and 37.

[Hrb05] F. Hrbata. *Callback Framework for VFS layer.* Master's thesis, Brno University of Technology, 2005. On page 71.

[HS] C. Henk and M. Szeredi. *Filesystem in Userspace (FUSE).* `http://fuse.sourceforge.net`. URL `http://fuse.sourceforge.net/`. On pages 69 and 95.

[IDG06] A. Iamnitchi, S. Doraimani, and G. Garzoglio. *Filecules in High-Energy Physics: Characteristics and Impact on Resource Management.* In *Proc. of the 15th IEEE Int. Symposium on High Performance Distributed Computing*, pp. 69–80, 2006. On page 66.

[Int11] Intel. *Intel Virtualization Technology for Directed I/O*, 2011. On page 32.

[KBC+00] J. Kubiatowicz, D. Bindel, Y. Chen, et al. *OceanStore: An Architecture for Global-Scale Persistent Storage.* ACM SIGPLAN Notices, **35**(11) pp. 190–201, 2000. On pages 1, 53, and 73.

[KK09] R. Královič and R. Královič. *Rapid Almost-Complete Broadcasting in Faulty Networks.* Theoretical Computer Science, **410**(14) pp. 1377–1387, 2009. On page 90.

[KKL+07] A. Kivity, Y. Kamay, D. Laor, et al. *kvm: the Linux Virtual Machine Monitor.* In *Proc. of the 2007 Linux Symposium*, pp. 225–230, 2007. On page 32.

[KL96] T. M. Kroeger and D. D. E. Long. *Predicting Future File-System Actions From Prior Events.* In *Proc. of the USENIX Annual Technical Conference*, pp. 319–328, 1996. On page 65.

[KLL+97] D. Karger, E. Lehman, T. Leighton, et al. *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web.* In *Proc. of the 29th Annual ACM Symposium on Theory of Computing*, pp. 654 – 663, 1997. On pages 4 and 77.

[KRZ08] M. Kirsanov, A. Ribon, and O. Zenin. *Development, validation and maintenance of Monte Carlo event generators and generator services in the LHC era.* PoS, **ACAT**, 2008. On page 40.

[Kut08] K. Kutzner. *The Decentralized File System Igor-FS as an Application for Overlay-Networks.* Ph.D. thesis, University of Karlsruhe, 2008. On pages 1, 53, 63, and 73.

[L+11] E. Lanciotti et al. *An alternative model to distribute VO specific software to WLCG sites: a prototype at PIC based on CernVM file system.* In *Proc. of the 18th Int. Conference on Computing in High Energy Physics (CHEP'10)*, 2011. To appear. On pages 50 and 112.

[Law10] D. Lawrence. *GlueX Offline Software: Preparing for Big Data Volumes on a Small Manpower Budget*, 2010. Talk at the 18th Int. Conf. on Computing in High Energy Physics (CHEP'10). On page 43.

[LHP+04] E. Laure, F. Hemmer, F. Prelz, et al. *Middleware for the next generation Grid infrastructure.* Tech. Rep. EGEE-PUB-2004-002, EGEE, 2004. On pages 3, 18, 19, and 41.

[LM09] A. Lakshman and P. Malik. *Cassandra: Structured Storage System on a P2P Network.* In *Proceedings of the 21th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SSPA'09)*, 2009. On page 78.

[Lov05] R. Love. *Kernel korner: intro to inotify.* Linux Journal, **2005**(139) p. 8, 2005. On page 71.

[LPGM08] A. W. Leung, S. Pasupathy, G. Goodson, et al. *Measurement and Analysis of Large-Scale Network File System Workloads*. In *Proc. of the USENIX Annual Technical Conference*, pp. 213–226, 2008. On page 43.

[LRTB97] M. Livny, R. Raman, T. Tannenbaum, et al. *Mechanisms for High Throughput Computing*. SPEEDUP, **11**(1), 1997. On page 12.

[LUC+05] J. LeVasseur, V. Uhlig, M. Chapman, et al. *Pre-Virtualization: Slashing the Cost of Virtualization*. Tech. Rep. 2005-30, University of Karlsruhe, 2005. On page 31.

[MA10] R. McDougall and J. Anderson. *Virtualization Performance: Perspectives and Challenges Ahead*. ACM SIGOPS Operating Systems Review, **44**(4) pp. 40–56, 2010. On page 75.

[Mae08] T. Maeno. *PanDA: Distributed Production and Distributed Analysis System for ATLAS*. Journal of Physics: Conference Series, **119**(6), 2008. On pages 21 and 41.

[MB11] D. T. Meyer and W. J. Bolosky. *A Study of Practical Deduplication*. In *Proc. of the 9th USENIX conference on File and Storage Technologies (FAST'11)*, 2011. On pages 43 and 50.

[Mel07] R. Melhem. *Low Diameter Interconnections for Routing in High-Performance Parallel Systems*. IEEE Transactions on Computers, **56**(4) pp. 502–510, 2007. On pages 90 and 91.

[Men07] P. B. Menage. *Adding Generic Process Containers to the Linux Kernel*. In *Proc. of the Ottawa Linux Symposium*, pp. 45–57, 2007. On page 33.

[Mer06] R. C. Merkle. *A Digital Signature Based on a Conventional Encryption Function*, vol. 293 of *Lecture Notes in Computer Science*, pp. 369–378. Springer, 2006. On page 3.

[MM10] P. Malzacher and A. Manafov. *PROOF on Demand*. Journal of Physics: Conference Series, **219**, 2010. On page 35.

[MMGC02] A. Muthitacharoen, R. Morris, T. M. Gil, et al. *Ivy: A Read/Write Peer-to-Peer File System*. ACM SIGOPS Operating Systems Review, **36**(SI) pp. 31–44, 2002. On pages 53 and 61.

[MON00] MONARC Members. *MONARC: Models of Networked Analysis at Regional Centers for LHC Experiments*. Tech. Rep. CERN-LCB-2000-001, CERN, 2000. On page 20.

[MSC+86] J. H. Morris, M. Satyanarayanan, M. H. Conner, et al. *Andrew: A distributed personal computing environment*. Communications of the ACM, **29**(3) pp. 184–201, 1986. On pages 51 and 53.

[NB+09] A. Naumann, R. Brun, et al. *ROOT - A C++ framework for petabyte data storage, statistical analysis and visualization*. Computer Physics Communications, **180**(12) pp. 2499–2512, 2009. On page 40.

[NSL+06] G. Neiger, A. L. Santoni, F. H. Leung, et al. *Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization*. Intel Technology Journal, **10**(03), 2006. On page 32.

[NWO88] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. *Caching in the Sprite Network File System*. ACM Transactions on Computer Systems, **6**(1) pp. 134–154, 1988. On page 75.

[Ope] Open Source Initiative. *The Open Source Definition*. `http://opensource.org/docs/osd`. On page 38.

[Per00] D. H. Perkins. *Introduction to High Energy Physics*. Cambridge University Press, 4th ed., 2000. On pages 7 and 8.

[PG74] G. J. Popek and R. P. Goldberg. *Formal Requirements for Virtualizable Third Generation Architectures*. Communications of the ACM, **17**(7) pp. 412–421, 1974. On page 28.

[PS06] K. Panagiotou and A. Souza. *On Adequate Performance Measures for Paging*. Annual ACM Symposium on Theory Of Computing, **38** pp. 487–496, 2006.

[QD02] S. Quinlan and S. Dorward. *Venti: a new approach to archival storage*. In *Proc. of the 1st USENIX Conf. on File and Storage Technologies (FAST'02)*, pp. 89–102, 2002. On pages 60 and 66.

[RD01] A. Rowstron and P. Druschel. *Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility*. ACM SIGOPS Operating Systems Review, **35**(5) pp. 188–201, 2001. On pages 61 and 73.

[RI00] J. S. Robin and C. E. Irvine. *Analysis of the Intel Pentium's ability to support a secure virtual machine monitor.* In *Proc. of the 9th Int. Conf. on USENIX Security Symposium (SSYM'00)*, vol. 9, pp. 129–143, 2000. On pages 28 and 30.

[RO91] M. Rosenblum and J. K. Osterhout. *The Design and Implementation of a Log-Structured File System.* ACM SIGOPS Operating Systems Review, **25**(5), 1991. On page 69.

[Roy09] A. Roy. *Building and testing a production quality grid software distribution for the Open Science Grid.* Journal of Physics: Conference Series, **180**(1), 2009. On page 52.

[RP05] N. Ravichandran and J.-F. Paris. *Making Early Predictions of File Accesses.* In *Proc. of the 4th Int. Information and Telecommunication Technologies Symposium*, pp. 122–129, 2005. On page 65.

[SAB+03] P. Saiz, L. Aphecetche, P. Buncic, et al. *AliEn - ALICE environment on the GRID.* Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, **502**(2-3) pp. 437–440, 2003. On pages 21 and 41.

[SBM05] A. Sharma, A. Bestavros, and I. Matta. *dPAM: A Distributed Prefetching Protocol for Scalable Asynchronous Multicast in P2P Systems.* IEEE INFOCOM, **2** pp. 1139–1150, 2005. On page 65.

[SBQ+10] B. Segal, P. Buncic, D. G. Quintas, et al. *LHC Cloud Computing with CernVM.* PoS, **ACAT**(004), 2010. On page 28.

[SH96] P. Sarkar and J. Hartman. *Efficient Cooperative Caching Using Hints.* ACM SIGOPS Operating Systems Review, **30**(SI) pp. 35–46, 1996. On pages 4 and 75.

[SKH10] I. Sriram and A. Khajeh-Hosseini. *Research Agenda in Cloud Technologies*, 2010. `arXiv:1001.3259`. On page 24.

[SLB+08] D. Spiga, S. Lacaprara, W. Bacchi, et al. *CRAB: the CMS distributed analysis tool development and design.* Nuclear Physics B: Proceedings Supplements, **177–178** pp. 267–268, 2008. On page 41.

[SN05] J. E. Smith and R. Nair. *Virtual Machines*. Morgan Kaufmann, 2005. On pages 2 and 25.

[Sop97] D. E. Soper. *Parton Distribution Functions*. Nuclear Physics B: Proceedings Supplements, **53**(1-3) pp. 69–80, 1997. On page 38.

[SSR08] T. Schütt, F. Schintke, and A. Reinefeld. *Scalaris: Reliable Transactional P2P Key/Value Store*. Proc. of the 7th ACM SIGPLAN Workshop on ERLANG (ERLANG'08), pp. 41–48, 2008. On page 78.

[TBB+08] A. Tsaregorodtsev, M. Bargiotti, N. Brook, et al. *DIRAC: A Community Grid Solution*. Journal of Physics: Conference Series, **119**(6), 2008. On pages 21 and 41.

[THB06] A. S. Tanenbaum, J. N. Herder, and H. Bos. *File Size Distribution on UNIX Systems—Then and Now*. ACM SIGOPS Operating Systems Review, **40**(1), 2006. On pages 43, 44, and 45.

[The08a] The ALICE Collaboration. *The ALICE experiment at the CERN LHC*. Journal of Instrumentation, **3**, 2008. On page 11.

[The08b] The ATLAS Collaboration. *The ATLAS experiment at the CERN LHC*. Journal of Instrumentation, **3**, 2008. On page 11.

[The08c] The CMS Collaboration. *The CMS experiment at the CERN LHC*. Journal of Instrumentation, **3**, 2008. On page 11.

[The08d] The LHCb Collaboration. *The LHCb experiment at the CERN LHC*. Journal of Instrumentation, **3**, 2008. On page 11.

[The08e] The LHCf Collaboration. *The LHCf experiment at the CERN LHC*. Journal of Instrumentation, **3**, 2008. On page 11.

[The08f] The TOTEM Collaboration. *The TOTEM experiment at the CERN LHC*. Journal of Instrumentation, **3**, 2008. On page 11.

[The09a] The ICHFA DPHEP International Study Group. *Data Preservation in High-Energy Physics*, 2009. `arXiv:0912.0255v1`. On page 28.

[The09b] The MoEDAL Collaboration. *Technical Design Report of the MoEDAL Experiment*. Tech. Rep. CERN-LHCC-2009-006, CERN, 2009. On page 11.

[TKS+03] N. Tolia, M. Kozuch, M. Satyanarayanan, et al. *Opportunistic Use of Content Addressable Storage for Distributed File Systems.* In *Proc. of the USENIX Annual Technical Conference*, 2003. On pages 60 and 63.

[TL05] D. Thain and M. Livny. *Parrot: An Application Environment for Data-Intensive Computing.* Scalable Computing: Practice and Experience, **6**(3) p. 9, 2005. On page 69.

[TvS07] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms.* Prentice-Hall, 2007. On page 59.

[UNR+05] R. Uhlig, G. Neiger, D. Rodgers, et al. *Intel Virtualization Technology.* Computer, **38**(5) pp. 48–56, 2005. On page 31.

[vRDGT08] R. van Renesse, D. Dumitriu, V. Gough, et al. *Efficient Reconciliation and Flow Control for Anti-Entropy Protocols.* In *Proc. of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS'08)*, 2008. On page 91.

[VRMCL08] L. M. Vaquero, L. Rodero-Merino, J. Caceres, et al. *A Break in the Clouds: Towards a Cloud Definition.* ACM SIGCOMM Computer Communication Review, **39**(1), 2008. On page 24.

[WCM+10] R. Wartel, T. Cass, B. Moreira, et al. *Image Distribution Mechanisms in Large Scale Cloud Providers.* In *Proc. of the 2nd IEEE Int. Conf. on Cloud Computing Technology and Science (CloudCom'10)*, pp. 112–117, 2010. On page 34.

[WDG+04] C. P. Wright, J. Dave, P. Gupta, et al. *Versatility and Unix Semantics in a Fan-Out Unification File System.* Tech. Rep. FSL-04-01b, Stony Brook University, 2004. On page 71.

[ZH04] Y. Zhu and Y. Hu. *Towards Efficient Load Balancing in Structured P2P Systems.* In *Proc. 18th Int. Parallel and Distributed Processing Symposium*, 2004. On pages 4 and 78.