SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems

Chair of Software Engineering

Sergey Morozov

Tiered File System: Optimization of Architecture and Management Algorithms

Master's Thesis

Scientific supervisor:
Dr. Sc. (Phys.-Math.), Professor Vyacheslav Nesterov

Reviewer:
Andrey Pakhomov,
Senior Solutions Manager at Dell EMC

Saint-Petersburg
2017

Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование
информационных систем

Кафедра системного программирования

Морозов Сергей Валерьевич

Многоярусная файловая система: оптимизация архитектуры и алгоритмов управления

Магистерская диссертация

Научный руководитель:
д. ф.-м. н., профессор Нестеров В. М.

Рецензент:
Пахомов А. В.,
рук. отд. разработки в Dell EMC

Санкт-Петербург
2017

Abstract

Automated storage tiering, from the business point of view, allows for a considerable cut of data storage costs while preserving the storage system performance at an acceptable level. The aim of this master's thesis is to design and implement a tiered policy-based file system having, as tiers, a local disk or distributed POSIX-conformant file system and cloud object storage. As a result of this study, the tiered file system called CloudTieringFS, has been devised. The primary use case of CloudTieringFS is mass file storage. CloudTieringFS uses a POSIX-conformant file system as a permanent storage for metadata and as "capacious cache" for data and cloud object storage as a permanent storage for data. The key advantage of CloudTieringFS is simultaneous provision of features such as configurability of data migration via policies, underlying file system-agnosticism, and fault tolerance.

The study is divided into four chapters. In the first chapter, the background required to familiarize the reader with the automated storage tiering in an environment that includes a file system and cloud object storage is given. A survey of similar solutions is provided and common problems to be overcome by the tiered file system developer are identified. The second chapter surveys the selected distributed file systems and compares some of their features important for design and implementation. The third chapter summarizes requirements for the tiered file system and proposes a detailed design for such a system. In the last chapter, the performance of CloudTieringFS is evaluated in single- and multi-node configurations with BtrFS and OrangeFS file systems correspondingly.

The performance results of CloudTieringFS are promising. The average file access latency for the selected file access pattern differs insignificantly versus the file access latency of the underlying file system. Summarizing, it can be argued that properly adjusted automated storage tiering policies can preserve the underlying file system's performance with neglectable overheads while reducing data storage costs.

Acknowledgments

The author would like to express sincere gratitude to Vyacheslav Nesterov for his wise supervision from the beginning to the end of this study and for thoughtful reviews of all the drafts of this master's thesis.

The author owes his deepest gratitude to Andrey Pakhomov, who initially brought up the idea of this study and guided the author throughout this long journey: suggested the work directions, introduced the author to the experts and reviewed all the drafts of this master's thesis.

Special gratitude should go to Igor Suslov for tons of system programming books which were brought nearly every week from his personal library, for regular technical discussions and professional code review.

Sincere thank also goes to Ivan Andreyev for providing an expert's view on the OrangeFS internals.

Contents

Introduction

There is always a trade-off between storage performance, capacity, and price. At a relatively small[1] price, one can count on a high-performance storage system with small-to-medium capacity or a storage system with low-to-average performance and large capacity. There is no low-cost solution that offers both high performance and large capacity; high-end products are always expensive. Software, as always, can dramatically reduce the cost in certain use cases. Often, access time requirements vary depending on the data. For example, access to an executable binary of a program with which employees work every day should be fast, and it is critical for business continuity, while the time required to access a corporate event photo archive does not affect business processes and may be quite long. For decades, the storage industry has been offering intelligent storage systems capable of demoting certain data to slower and hence cheaper storage devices, while promoting other data to faster, more expensive storage devices. These storage systems implement *hierarchical storage management* or *information lifecycle management* (a broader concept).

The idea of hierarchical storage management has been here for years. This is a very well studied area, extensively used in the industry. There is a closely related concept called *automated storage tiering*, automatic promotion, demotion, and movement of data between storage tiers based on a policy. Nowadays, there is another surge of interest in automated storage tiering caused by emergence of new storage technologies and increase in customer demands. Capabilities to seamlessly migrate data between traditional storage systems and cloud storage are taking a keen interest.

Cloud technologies provide many benefits. For example, one can easily deploy an HPC[2] [22] cluster at an affordable price or an object storage that can satisfy limitless capacity needs. In the latter case, despite the benefits offered there are also some serious restrictions such as, in most cases, a weaker consistency model [6] and an object interface instead of

[1] For enterprise markets.

[2] High-Performance Computing

POSIX. It means that cloud object storage is more useful for newer, “cloud-native” applications that do not require conformance to POSIX semantics. This severely impacts cloud object storage applicability since some types of applications require to be conformant at least to the POSIX consistency model. Also, there is an enormous amount of POSIX-conformant legacy applications that could be but will not be replaced by their cloud-native alternatives over the next couple of decades because they already solve business problems.

POSIX was not originally designed for distributed systems, and it is extremely hard to design and implement, for instance, a distributed file system that is fully POSIX-conformant. There is quite a lot of distributed file systems, both commercial and open source, that provide near-POSIX semantics, but only a few that are fully POSIX-conformant. Is it possible to have the best of two worlds, POSIX access to a file system with unlimited capacity and good performance while saving money on scaling up and out an expensive storage system?

In this study, the specific case of automated storage tiering is considered: file-level automated storage tiering between an arbitrary POSIX- or near-POSIX-conformant file system and cloud object storage. The cloud object storage acts as a capacity extender for the POSIX-conformant file system; data location is opaque to the file system client; storage tiering policies ensure adequate data access times. Some commercial solutions, such as Dell EMC CloudArray [17] and Dell EMC 2 TIERS [16], address the same problem. There are academic solutions, such as the BlueSky file system [63] and the SCFS file system [51], addressing similar problems but in a slightly different manner. The tiered file system proposed in this study differs from these file systems since it takes an existing file system and uses it as a permanent metadata storage and as “capacious non-volatile cache” for data. The key advantages of the solution are configurability of data migration via policies, file system-agnosticism, and fault tolerance.

Problem Statement

The primary aim of this study is to design and implement a file system-agnostic policy-based software component responsible for data synchronization between a POSIX-conformant file system and cloud object storage. The solution should be licensed under a free software license so that any interested party can use it or make a contribution. This software component should also become a solid platform for future research in the field of automated storage tiering policies.

To achieve this aim, the following tasks have been formulated[3]:

(I) Investigate automated storage tiering problems in an environment that includes a distributed file system and cloud object storage.

(II) Extract and compare important features of modern distributed file systems from the perspective of automated storage tiering.

(III) Design a software component that enables automated storage tiering between a POSIX-conformant file system and cloud object storage.

(IV) Implement the designed software component and evaluate its performance.

[3] Each task has a corresponding section with the same Roman number.

I Background and Related Work

I.1 Terminology

The following terms are used throughout this study[4].

Data. The digital representation of anything in any form [57].

Metadata. Data associated with other data [57].

Object. The encapsulation of data and associated metadata [57]. (Note that there are other definitions of «object» in [57] but they are omitted as inapplicable in the context of the current study.)

Information Lifecycle Management (ILM). The policies, processes, practices, services and tools used to align the business value of information with the most appropriate and cost-effective infrastructure from the time information is created through its final disposition. Information is aligned with business requirements through management policies and service levels associated with applications, metadata and data [57].

Data Lifecycle Management (DLM). The policies, processes, practices, services and tools used to align the business value of data with the most appropriate and cost-effective storage infrastructure from the time data is created through its final disposition. Data is aligned with business requirements through management policies and service levels associated with performance, availability, recoverability, cost, etc. DLM is a subset of ILM [57].

Data storage as a Service (DSaaS). Delivery of appropriately configured virtual storage and related data services over a network, based on a request for a given service level. Typically, DSaaS hides limits to scalability,

[4] Many of the terms in this section are taken from The 2016 SNIA Dictionary. The author is grateful to the Storage Networking Industry Association for the permission to use these definitions.

is either self-provisioned or provisionless and is billed based on consumption [57].

Cloud Storage. Synonym for Data storage as a Service [57].

Object Service. Object-level access to storage [57].

Object Storage. A storage device that provides object services. Object storage includes DSaaS [57].

Hierarchical Storage Management (HSM). The automated migration of data objects among storage devices, usually based on inactivity. Hierarchical storage management is based on the concept of a cost-performance storage hierarchy. By accepting lower access performance (higher access times), one can store objects less expensively. By automatically moving less frequently accessed objects to lower levels in the hierarchy, higher cost storage is freed for more active objects, and a better overall cost-to-performance ratio is achieved [57].

Tiered Storage. Storage that is physically partitioned into multiple distinct classes based on price, performance or other attributes. Data may be dynamically moved among classes in a tiered storage implementation based on access activity or other considerations [57].

Policy. Policy can be defined from two perspectives:

(1) A definite goal, course or method of action to guide and determine present and future decisions. Policies are implemented or executed within a particular context (such as policies defined within a business unit) [49].

(2) Policies as a set of rules to administer, manage, and control access to network resources [48].

Policy Goal. Goals are the business objectives or desired state intended to be maintained by a policy system. As the highest level of policy abstraction, these goals are most directly described in business rather than technical terms. For example, a goal might state that a particular application operate on a network as though it had its own dedicated network, despite using a shared infrastructure. Policy goals can include the objectives of a service-level agreement, as well as the assignment of resources to applications or individuals. A policy system may be created that automatically strives to achieve a goal through feedback regarding whether the goal (such as a service level) is being met [49].

Policy Processor. In an intelligent device, the processor that schedules the overall activities. Policy processors are usually augmented by additional processors, state machines, or sequencers that perform the lower-level functions required to implement overall policy [57].

Policy Rule. A basic building block of a policy-based system. It is the binding of a set of actions to a set of conditions, where the conditions are evaluated to determine whether the actions are performed [49].

Policy Condition. A representation of the necessary state and/or prerequisites that define whether policy rule actions should be performed. This representation need not be completely specified, but may be implicitly provided in an implementation or protocol. When the policy condition(s) associated with a policy rule evaluate to TRUE, then (subject to other considerations such as rule priorities and decision strategies) the rule should be enforced [49].

Policy Action. Definition of what is to be done to enforce a policy rule, when the conditions of the rule are met. Policy actions may result in execution of one or more operations to affect and/or configure network traffic and network resources [49]. Rule actions may be ordered [48].

Policy Repository. A specific data store that holds policy rules, their conditions and actions, and related policy data. A database or directory would be an example of such a store [49].

Automated Storage Tiering. Automatic movement of data between storage tiers based on a policy. The tiers may be within a single storage system or may span storage systems, including a cloud storage tier [57].

File System. A software component that imposes structure on the address space of one or more physical or virtual disks so that applications may deal more conveniently with abstract named data objects of variable size (files). File systems are often supplied as operating system components, but are also implemented and marketed as independent software components.

File. An abstract data object made up of (a.) an ordered sequence of data bytes stored on a disk or tape, (b.) a symbolic name by which the object can be uniquely identified, and (c.) a set of properties, such as ownership and access permissions that allow the object to be managed by a file system or backup manager. Unlike the permanent address spaces of storage media, files may be created and deleted, and in most file systems, may expand or contract in size during their lifetimes [57].

Extended Attributes. Extended attributes are name:value pairs associated permanently with files and directories. An attribute may be *defined* or *undefined.* If it is defined, its value may be empty or non-empty. They are often used to provide additional functionality to a file system. Extended attributes are accessed as atomic objects [64].

Monitor. A program that executes in an operating environment and keeps track of system resource utilization. Monitors typically record CPU utilization, I/O request rates, data transfer rates, RAM utilization, and similar statistics. A monitor program, which may be an integral part of an operating system, a separate software product, or a part of a related component, such as a database management system, is a necessary prerequisite to manual I/O load balancing [57].

Daemon. A daemon is a process with the following characteristics:

- It is long-lived. Often, a daemon is created at system startup and runs until the system is shut down.
- It runs in the background and has no controlling terminal. The absence of a controlling terminal ensures that the kernel never automatically generates any job-control or terminal-related signals (such as `SIGINT`, `SIGTSTP`, and `SIGHUP`) for a daemon [32].

I.2 File Systems

This section discusses file system-related concepts required for better understanding of the following sections.

I.2.1 Portable Operating System Interface

POSIX is an acronym for Portable Operating System Interface. The term POSIX refers to a group of standards developed under the auspices of the Institute of Electrical and Electronic Engineers (IEEE), specifically its Portable Application Standards Committee. The name POSIX was suggested by Richard Stallman. The term POSIX was originally used as a synonym for IEEE Std 1003.1-1988. A preferred term for that standard, POSIX.1, emerged [43]. POSIX.1 documents an API[5] for a set of services that should be made available to a program by a conforming operating system. An operating system that does this can be certified as POSIX.1 conformant [32].

The POSIX.1 standard was developed for local disk file systems. Most of the distributed file systems that claim to be POSIX-conformant relax some of restrictions imposed by the standard. For example, OrangeFS does not support file locking [41] and CephFS does not guarantee atomic

[5] Application Programming Interface

writes in shared simultaneous writer situations, when a write crosses object boundaries [14]. Conformance to POSIX for some distributed file systems is discussed in more detail in Section II.

I.2.2 Virtual File System

The virtual file system (VFS) is a kernel feature that provides an abstraction layer for file system operations. VFS defines a generic interface for file system operations. All programs that work with files specify their operations in terms of this generic interface. Each file system provides an implementation for the VFS interface. The VFS interface includes operations corresponding to all typical system calls used to work with file systems and directories, such as `open()`, `read()`, `write()`, `lseek()`, `close()`, `truncate()`, `stat()`, `mount()`, `umount()`, `mmap()`, `mkdir()`, `link()`, `unlink()`, `symlink()`, and `rename()` [32].

Some file systems do not support all of the VFS operations. For example, as of Linux kernel 4.11, the kernel module of the OrangeFS [40] parallel file system does not implement the `fallocate()` operation (according to the source code analysis), which, by the way, is Linux-specific and not specified in POSIX.1. In such cases, the underlying file system passes an error code back to the VFS layer indicating the lack of support, and VFS in turn passes this error code back to the application.

I.2.3 Distributed File Systems

This section provides definitions for different types of distributed file systems. Selected distributed file systems of these types are compared in Section II.

Distributed File System. A distributed file system enables programs to store and access remote files exactly as they do with the local files, allowing users to access files from any computer on a network. Performance and reliability experienced when accessing the files stored at a server should be comparable to that for files stored on local disks [19].

Cluster File System. A distributed file system that is not a single server with a set of clients, but instead a cluster of servers that all work together to provide high performance service to their clients. The cluster is transparent to the clients—it is just "the file system," but the file system software distributes requests to elements of the storage cluster [10].

Parallel File System. A file system that supports parallel applications, all nodes may be accessing the same files at the same time, concurrently reading and writing. Data for a single file is striped across multiple storage nodes to provide scalable performance to individual files [10].

I.3 Cloud Object Storage Systems

Various aspects should be considered when selecting cloud object storage for the cloud object storage tier in a tiered file system. The most important characteristics to be taken into account are the implemented consistency model and possible types of service-level agreements. The object protocol is also important since a particular protocol may offer unique features that could be utilized to optimize performance of the tiered file system.

I.3.1 Consistency Models

Consistency Model. A consistency model is essentially a contract between processes and the data store. It says that if processes agree to obey certain rules, the store promises to work correctly [59].

Strong Consistency. After the update completes, any subsequent access by any process will return the updated value [62].

Weak Consistency. The system does not guarantee that subsequent accesses will return the updated value. A number of conditions need to be met before the value will be returned. The period between the update and the moment when it is guaranteed that any observer will always see the updated value is referred to as the *inconsistency window* [62].

Eventual Consistency. This is a specific form of weak consistency; the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value [62].

Causal Consistency. If process A has communicated to process B that it has updated a data item, a subsequent access by process B will return the updated value, and a write is guaranteed to supersede the earlier write. Access by process C that has no causal relationship to process A is subject to normal eventual consistency rules [62].

Read-Your-Writes Consistency. This is an important model where process, after having updated a data item, always accesses the updated value and never sees an older value. This is a special case of the causal consistency model [62]. It can also be referred to as *read-after-write consistency.*

Consistency Models of Selected Cloud Object Storage Systems. According to the survey of cloud object storage systems made in [55], different cloud object storage systems implement different consistency models. For example, Amazon S3 provides read-after-write consistency for PUT requests of new objects in a S3 bucket in all regions with one caveat. The caveat is that if one makes a HEAD or GET request to the key name (to find if the object exists) before creating the object, Amazon S3 provides eventual consistency for read-after-write. Amazon S3 offers eventual consistency for overwrite PUT and DELETE requests in all regions [5]. Dell EMC ECS provides strong consistent views of data regardless of where it is stored. It achieves this strong consistency by representing each bucket, object, directory, and file as an entity, and applying the appropriate technique on each entity based on its traffic pattern. When the technique can avoid a WAN roundtrip, average latency is reduced [61]. Google Cloud Storage provides strong global consistency for the following operations, including both data and metadata: read-after-write, read-after-metadata-update, read-after-delete, bucket listing, object listing in regional locations, granting access to resources. The following operations are eventually consistent: list operations for objects in multi-regional locations and revoke access from resources [28].

I.3.2 Service-Level Agreements

A service-level agreement (SLA) is an agreement between a service provider and service customer about the required quality-of-service (QoS) characteristics of some service(s) delivered by the provider to the customer. The agreement as such is the intangible understanding, or accord, that exists between the provider and customer [31]. A typical SLA describes levels of service using various attributes such as availability, serviceability or performance. The SLA specifies thresholds and financial penalties associated with violations of these thresholds [46].

The following aspects should be considered in the SLA: data preservation and redundancy, data location, data seizure, data privacy, data availability, planned maintenance, network availability, storage availability, service response time, and others. All these may affect the choice of the cloud object storage provider. The cloud object storage tier of the tiered file system, in theory, can be represented as a combination of multiple cloud object storage systems, some of which will store more critical data than others.

I.3.3 Protocols

Cloud object storage systems can implement one of multiple object protocols. Popular object protocols are the following:

- **S3.** The S3 protocol is arguably the most commonly used object storage protocol. Some unique features of the S3 protocol include (1) bucket-level controls for versioning and expiration, (2) server-side copies of objects, and (3) the ability to set public access on an object and serve it via HTTP/HTTPS without authentication [15].

- **Swift.** The Swift protocol is very similar to the S3 protocol. It uses buckets (containers) that contain key-value objects. The unique features of the Swift API include (1) flexible authentication through a separate mechanism creating a “token” that can be passed around to authenticate requests, and (2) creation of objects of unknown size beforehand [15].

- **CDMI.** The CDMI protocol can be used to create, retrieve, update, and delete objects in a cloud. CDMI features include (1) ability to display Windows and NFS compatible access controls, (2) ability to discover whether a container (bucket) shall be deleted at the end of its retention period, and (3) ability to find legal holds that have been placed on a container [3].

I.4 Related Work

In this section, solutions that precede this study are presented, their advantages and disadvantages are described, and comparison with the tiered file system proposed in this study is given. Based on the surveyed papers, common problems arising when designing tiered file systems with POSIX file system and cloud object storage tiers as well as the problems revealed during the tiered file system development are listed.

I.4.1 Existing Automated Storage Tiering Solutions

There are several commercial, open source, and academic POSIX or near-POSIX file systems that use cloud object storage to store file data or both file data and metadata.

Dell EMC CloudArray [17] is perhaps the most similar solution to the one proposed in this study in terms of the covered use cases. Dell EMC CloudArray is a mature commercial product that solves the performance/capacity/price ratio problem by tiering storage between the file system and cloud object storage. It provides cloud-integrated storage that extends high-performance storage arrays with cost-effective cloud capacity. By providing access to a private or public cloud storage tier through standard interfaces, Dell EMC CloudArray technology simplifies storage management for inactive data and offsite protection. Dell EMC CloudArray's policy-driven cache ensures the proper level of accessibility and performance based on the data stored. The cache is local on the appliance and delivers high performance while asynchronously replicating data to the cloud. Each cache can be sized and assigned a policy to support a percentage of client's data based on current needs [20]. Dell EMC CloudArray provides a full set of features needed for enterprise customers. Unfortunately, Dell EMC CloudArray internal architecture is unknown to the public and the source code is proprietary.

There is also a solution called Dell EMC 2 TIERS [16], which is primarily intended for high-performance computing (HPC), and even extreme HPC use cases. The solution is tightly integrated with the parallel file system OrangeFS [40]. Dell EMC 2 TIERS software presents the POSIX interface and namespace to applications by virtue of the file system and maps the applications data into objects on the cloud object storage, with policy-driven tiering between the two. The unique characteristics of Dell EMC 2 TIERS include (1) single global namespace with dynamically loadable namespaces, (2) tiering of both data and metadata, (3) tiering and non-tiering modes, and (4) direct read-only access to the cloud object storage tier, bypassing the file system tier [45]. Note that metadata tiering is an important and unique feature of Dell EMC 2 TIERS because in HPC systems, metadata may consume space comparable with data. This solution is HPC-oriented; the files and dynamically loadable namespaces migrate to the file system at the direction of the scheduler, which makes the decisions based on the HPC tasks queue. As of May 2017, Dell EMC 2 TIERS is not released and not production-ready.

MarFS [36] is probably the most similar solution to the one proposed in this study in terms of architecture. It is a near-POSIX global scalable namespace over many POSIX and non-POSIX data repositories. MarFS is primarily intended as a file system for large data collections, but not for application execution. It focuses primarily on HPC use cases and relaxes POSIX semantics. MarFS does not (1) allow updating the file in place for object data repositories and (2) check for or protect against multiple writers into the same file. MarFS uses a POSIX-conformant file system as a metadata storage and as a cache for data. MarFS uses extended attributes to store file metadata, such as the file data object identifier in cloud object storage. The data is written to the data component, which can be a POSIX file system or an object store. All normal attributes, such as permissions, dates, and even file sizes, are kept up-to-date in semantically reasonable ways. The file size is updated by truncating the POSIX metadata file to the size of the desired file even though there may be no actual data in the file itself. For this reason, POSIX file systems used for the metadata component must support sparse files [37]. The solution is open source.

Saga [56] is a user mode file system based on cloud object storage service, designed to support POSIX with the goal of minimizing cost. Saga authors argue that Saga is efficient from the performance perspective and utilizes parallel characteristics of cloud object storage to boost performance. Since it was not explicitly stated whether Saga is a distributed file system or not, it is assumed that Saga is a single-node file system. All files stored in Saga are chunked into fixed-size blocks and all the fixed-size blocks are stored as objects in the cloud object storage. Saga can be divided into three modules: a cache module named Dragon Orb, a kernel module redirecting file system calls to Dragon Orb, and a network module taking charge of writing data to and reading data from the cloud object storage. The kernel module of Saga redirects all the file system calls to the user mode cache module Dragon Orb. Dragon Orb manages fixed-size cache on the local file system to store objects and utilizes a variant of the LRU cache replacement algorithm to evict objects when an object has to be loaded into the full object cache. Unfortunately, the link to the repository with Saga's source code was not found.

BlueSky [63] is a network file system backed by cloud storage. BlueSky stores data persistently in cloud object storage. Clients access the storage through a proxy running on-site, which caches data to provide lower-latency responses and additional opportunities for optimization. BlueSky provides standard POSIX file system semantics, including atomic renames and hard links. BlueSky supports multiple protocols—both NFS and CIFS—and is portable to different cloud providers. The central component of BlueSky is a proxy situated between clients and cloud providers. The proxy communicates with clients in an enterprise using a standard network file system protocol, and communicates with cloud providers using a cloud storage protocol. It supports multiple file system clients. Nevertheless, currently only one proxy can be used in the system, which can become a performance bottleneck and a single point of failure. The project is open source.

SCFS is a cloud-backed file system that provides strong consistency and near-POSIX semantics on top of eventually consistent cloud storage services. SCFS provides a pluggable backplane that allows it to work with various storage clouds or a cloud of clouds. SCFS does not rely on the features specific to the selected cloud object storage provider besides on-demand access to storage and basic access control lists. A primary goal of SCFS is to allow clients to share files in a controlled way, providing the necessary mechanisms to guarantee security. SCFS also aims to offer a natural file system API with strong consistency. SCFS is not intended to be a big-data file system, since file data is uploaded to and downloaded from one or more clouds. SCFS uses a fault-tolerant coordination service. The metadata and coordination services are assumed to run in the cloud on compute nodes, while clients connect to the file system via FUSE file systems, which in turn are connected to the SCFS agents (daemons). The project is open source.

Compared to the above solutions, the tiered file system presented in this study concentrates on the mass file storage use case, uses an existing local disk or distributed POSIX-conformant file system as a permanent metadata storage and as a hybrid of permanent storage and cache for data. The proposed tiered file system provides the levels of consistency and fault tolerance similar to the underlying file system's, and uses policies that define data migration rules.

I.4.2 Automated Storage Tiering Problems

There are several problems to be overcome by the tiered file system developer.

The first one is that some cloud object storage systems use the weak consistency model. The tiered file system should have a mechanism to verify that the data read from the cloud object storage are of the latest version.

The common problem of distributed systems is partitioning. The distributed tiered file systems should tolerate partitions within a cluster. Cloud object storage connection failures should also be handled properly.

The tiered file system may implement automated storage tiering based on a policy. No matter how advanced the policies are, there will always be "cache misses", accesses to files residing in the cloud object storage tier. This means that the tiered file system should be deployed in close proximity to the data center where cloud object storage is deployed to ensure that access latency for relatively large files is not very high.

In tiered file systems, each file usually has more metadata than in traditional file systems. It is unreasonable to demote files of sizes less then tens of kilobytes, since the file will consume more space due to increased total size of data and metadata.

Finally, there is a problem which is not evident in the beginning of the tiered file system development, but which may significantly impact the tiered file system's usability. Consider the following use case: one of the tiered file system directories contains thousands of images, and the user opens this directory with a graphical file manager. At first, for example, files are shown in a detailed view mode, where each file is represented as a file name with a small icon indicating the file type. Then, the user switches to the preview mode. Suppose images are encoded with the JPEG file interchange format (JFIF) [29], which allows for storing thumbnails using the JFIF APP0 marker segment. To get image thumbnails, the graphical file manager opens each file and reads a few kilobytes of data from the beginning. If the tiered file system does not provide any special handling for such image files, all images will be migrated to the file system tier even when the user needs only one image to be opened. Note that modern graphical file managers provide some sort of protection against previewing files that reside on network file systems. For example, the user can set an option to preview only files smaller than a certain size to prevent unnecessary data transfers [60].

II Comparison of Selected Distributed File Systems

The tiered file system proposed in this study uses an existing file system as the tier for fast data access. One of the goals is to provide tiering support for a distributed POSIX-conformant file system. In this regard, before starting to work on the design and implementation, several existing distributed file systems were surveyed. Based on the survey results, some features of the selected file systems, which are important for design and implementation, were identified and compared.

MooseFS. MooseFS [1] is an open source network distributed file system. It is fault-tolerant, highly performing, easily scalable, and POSIX-conformant. MooseFS spreads data over several physical commodity servers, which are visible to the user as one big volume. For standard file operations, MooseFS acts like an ordinary Unix-like file system: it provides a hierarchical directory structure, stores POSIX file attributes, supports ACLs, supports POSIX and BSD locks, supports special files (block and character devices, pipes and sockets), and supports symbolic and hard links. Distinctive MooseFS features are the following:

- high reliability,
- no single point of failure,
- parallel data operations,
- dynamic capacity expansion via addition of new computers,
- coherent, "atomic" snapshots of files, and
- data tiering (supports different storage policies for different files/directories).

MooseFS is an open source solution licensed under the GPLv2 license.

CephFS. CephFS [13] is a distributed near-POSIX file system that uses a Ceph Storage Cluster to store its data. CephFS provides dynamic distributed metadata management using a metadata cluster (MDS) and stores data and metadata in Object Storage Devices (OSD). CephFS aims to adhere to POSIX semantics wherever possible. CephFS maintains strong cache coherency across clients. The goal is for processes communicating via the file system to behave the same way when they are on different hosts as when they are on the same host. There are a few places where CephFS diverges from strict POSIX semantics for various reasons:

- If a client is writing to a file and fails, its writes are not necessarily atomic.
- In shared simultaneous writer situations, a write that crosses object boundaries is not necessarily atomic.
- A `seekdir()` [52] to a non-zero offset may often work but is not guaranteed to do so.
- Sparse files propagate incorrectly to the `st_blocks` member of `struct stat`.
- When a file is mapped into memory via `mmap()` [35] on multiple hosts, writes are not coherently propagated to other clients' caches.

CephFS also provides some tools to relax consistency. For example, the `O_LAZY` option allows users to read a file even if it is currently being rewritten [18]. CephFS is fault-tolerant and supports tiering. There are two tiers named *Cache Tier* and *Storage Tier*. Cache Tier is usually made of relatively fast/expensive storage devices while Storage Tier is made of relatively slower/cheaper devices. CephFS is an open source solution licensed under the LGPLv2.1 license.

GlusterFS. GlusterFS [2] is a scalable network file system suitable for data-intensive tasks such as cloud storage and media streaming. It has a client-server design with no metadata server. Instead, GlusterFS stores data and metadata on multiple devices attached to different servers. In GlusterFS, when a server becomes unavailable, it is removed from the system and no I/O operations to it can be performed [18]. GlusterFS is fully POSIX-conformant. GlusterFS supports tiering [27]. The tiering feature enables different storage types to be used by the same logical volume. In GlusterFS, the two types are classified as "cold" and "hot", and are represented as two groups of bricks[6]. The hot group acts as cache for the cold group. GlusterFS is an open source solution licensed under the dual GPLv2 / LGPLv3 or later license.

OrangeFS. OrangeFS [40] is a scale-out network file system designed for use on high-end computing (HEC) systems that provides very high-performance access to multi-server-based disk storage, in parallel. It is designed specifically to scale to very large numbers of clients and servers. OrangeFS is also used with data-intensive systems and projects for commodity networks, big data, and business applications. Since Linux kernel 4.6 release, OrangeFS kernel module is a part of the Linux kernel. OrangeFS features include:

- file data distribution among multiple file servers,
- support of simultaneous access by multiple clients,
- storage of file data and metadata on servers using the local file system, and
- statelessness.

OrangeFS is fault-tolerant. Given enough hardware, OrangeFS can even handle server failures. OrangeFS does not support tiering (as of OrangeFS 2.9.6), but according to [8] there is a plan to add this feature in OrangeFS 3.0 release. OrangeFS is open source and licensed under the LGPLv2.1 license.

Comparison. The choice of the distributed file system for the file system tier of the tiered file system was based on four characteristics: (1) POSIX conformance, (2) tiering support, (3) fault tolerance, and (4) license. The distributed file system should provide at least near-POSIX semantics to be compatible with legacy applications. It should be fault-tolerant to be used in production environments. The distributed file system should be available for use in a binary or source code form to perform experiments without buying it or violating a proprietary license agreement. That is

[6] Brick is the basic unit of storage in GlusterFS, represented by an export directory on a server in the trusted storage pool.

why file systems, such as GPFS [30], are not included in this short survey. The tiering mechanism can be reused while adding support of the cloud object storage tier. However, the absence of a tiering mechanism in the distributed file system provides the possibility for its creation. The above-listed distributed file systems are compared in Table 1 based on these four characteristics.

	POSIX Conformance	Tiering Support	Fault Tolerance	License
MooseFS	full	+	+	GPLv2
CephFS	near	+	+	LGPLv2.1
GlusterFS	full	+	+	GPLv2/LGPLv3
OrangeFS	near	-	+	LGPLv2.1

Table 1: Comparison of selected distributed file systems.

During the design of the tiered file system proposed in this study, it was decided to make the tiered file system agnostic to the file system used as the file system tier, and the absence of a tiering feature has become a virtue—possibility to make a contribution to the open source community. Concerning this, as well as some other reasons, such as file system's maturity and the market interest in it, the OrangeFS parallel file system was chosen to evaluate performance of the designed tiered file system in a distributed configuration. Performance evaluation of the designed tiered file system with OrangeFS as the file system tier is done in Section IV.2.2.

III Requirements and Design

This section summarizes requirements for a file system implementing automated storage tiering and proposes a design for such a system. The storage tier represented by the POSIX-conformant file system is referred as the *hot tier*, and the storage tier represented by the cloud object storage is referred as the *cold tier*.

III.1 Requirements

The tiered file system should be able to run in various Linux operating environments. It is not required to ensure its compatibility with an arbitrary UNIX operating system. Any common Linux-specific features can be used.

III.1.1 Use Cases

The primary use case for the tiered file system is mass file storage within an organization. It is assumed that the tiered file system will be used for file sharing, storage of scientific computation artifacts, and storage of personal files.

III.1.2 User Interaction

The tiered file system should provide the POSIX semantics. Tiered file system's client applications should not be aware of the tier in which files reside. The time required to access a file in the cold tier of the tiered file system should linearly depend on the file size and network bandwidth. Tiered file system's predictive data promotion policies should minimize the probability of accessing a file in the cold tier.

III.1.3 Administration

The tiered file system should be highly configurable. The system administrator should be able to optimize it for specific workloads and file access patterns. Besides the ability to define custom storage tiering policies, the tiered file system should feature multiple operating modes, such as migration of data to the cold tier followed by removal of data from the hot tier (data demotion) or backup mode, when the cold tier is an eventually consistent replica of the hot tier.

Automated storage tiering policies can range from very simple and straightforward to very complex and adaptive. For example, a relatively simple policy is: demote files last accessed more than 10 minutes ago; promote files upon client's request. An example of a relatively complex policy is: demote files larger than 5 megabytes, for which the last access time difference with the current time is equal or greater than 1 hour; when a user from the group `accounting` logs into the operating system, promote all files from the `/mnt/fs/accounting` directory (start cold-to-hot tier data migration in background); when a user from the group `hr` accesses at least one file from the `/mnt/fs/hr` directory, promote all files in that directory; for users from other groups, provide on-demand file access, taking into account file permissions and ACLs[7] [32], if configured. An example of an adaptive policy could be the following: collect information about peaks and troughs of a number of accesses per minute for each file in the directory `/mnt/fs/share` within 24 hours; next day, promote and demote files based on the collected statistics, while collecting new statistics for the following day.

III.2 Design

Designing and implementing a new file system from scratch is inexpedient. It is more reasonable to devise a separate software component, responsible for automated storage tiering, and make it compatible with the majority of POSIX-conformant file systems. Such an approach provides the following benefits: (1) independent development cycles, (2) ability to use the this software component with different file systems, and (3) ability to use one instance of the software component with multiple file systems simultaneously.

In this section, a software component responsible for automated storage tiering is referred as the *tiered component*. The combination of the tiered

[7] Access Control List

component and the underlying file system is referred as the *tiered file system*. The underlying file system itself is simply referred as the *file system*.

The tiered file system consists of two high-level components: the tiered component and the file system. This section proposes the tiered component architecture and identifies the method of interaction between the tiered component and the file system.

Taking into consideration the use case requirements for the tiered file system, it is reasonable to design a system supporting file-level tiering instead of block-level tiering. File storage systems allow for less use cases as compared to block storage systems, but they fully meet the primary use case requirement—to provide mass file storage. Moreover, file-level tiering reduces tiered file system complexity and facilitates loose coupling of the tiered component and the file system.

III.2.1 Components

The tiered component consists of the following components:

- Daemon
    - Policy Processor
    - File System Scanner
    - Monitor
    - Data Mover
- System Call Interceptor

A UML component diagram [54] for the tiered component is presented in Figure 1.

Daemon. The daemon is the most intelligent part of the tiered component. It consists of a *policy processor*, *file system scanner*, *monitor*, and *data mover*. The policy processor schedules file demotion and promotion tasks which, in turn, are processed by the data mover. The file system scanner is constantly scanning the file system and feeds files to the policy processor. The monitor collects statistics and feeds these data to the policy processor. There is one daemon per operating system.

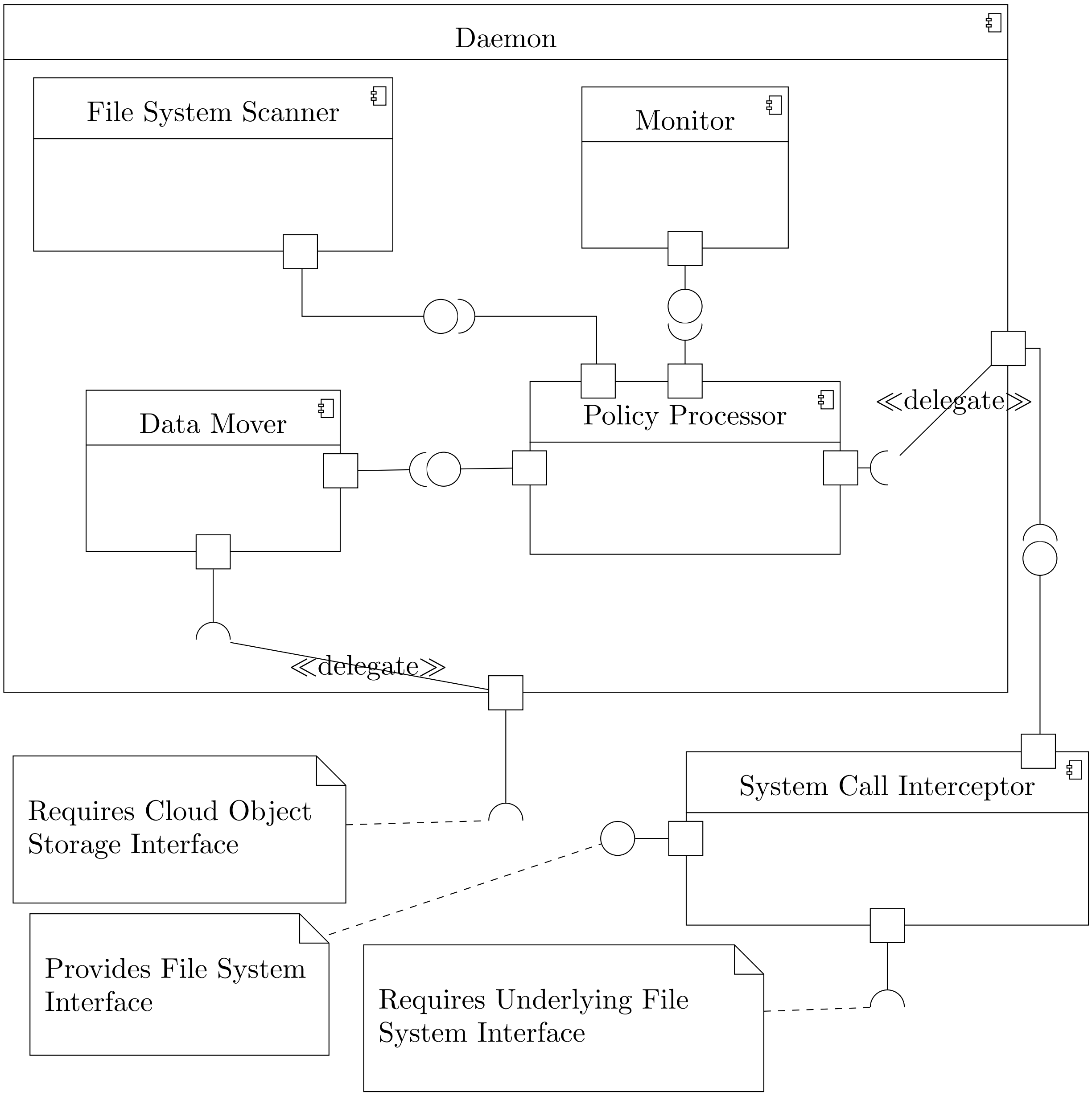

Daemon
File System Scanner
Monitor
Data Mover
Policy Processor
≪delegate≫
≪delegate≫
System Call Interceptor
Requires Cloud Object Storage Interface
Provides File System Interface
Requires Underlying File System Interface

Figure 1: Component diagram for the tiered component.

Policy Processor. The policy processor is responsible for scheduling of file promotion and demotion tasks. The system administrator defines a set of policy rules in the configuration file. The policies are selected either from a set of predefined policy rules in the policy repository or defined by means of a domain-specific language (DSL). There are three types of input: (1) file names from the file system scanner, (2) file names from the system call interceptor, and (3) statistics from the monitor, such as available capacity, RAM utilization, number of open file descriptors, and I/O request rate. Based on policy rules and statistics, the policy processor schedules file demotion and promotion tasks with a certain priority for files coming from the file system scanner. File promotion tasks for files coming from the system call interceptor are scheduled with the highest priority. Only the files that must be promoted are fed to the policy processor by the system call interceptor since there is no need to contact the daemon for files residing in the hot tier.

File System Scanner. The file system scanner is constantly traversing the file system's directory tree. It feeds all regular files to the policy processor for further processing.

Monitor. The monitor periodically collects information about the available capacity and the number of open file descriptors. It also collects useful statistics from all other daemon components. All data are then fed to the policy processor.

Data Mover. The data mover is responsible for execution of file demotion and promotion tasks scheduled by the policy processor. During task execution, the data mover performs predefined ordered actions on the file data and metadata conforming to a protocol (described in Section III.2.2) that ensures that the files are accessed correctly.

System Call Interceptor. The system call interceptor redefines some system calls and notifies the daemon about the need to promote a file when the tiered file system client accesses the file residing in the cold tier. The system call interceptor can be implemented at least in four different ways: (1) as a loadable kernel module intercepting some system calls [9], (2) as a new file system to be used in conjunction with OverlayFS [42], (3) as a FUSE file system [38], and (4) as a dynamically loaded shared library [32]. Each approach has its advantages and disadvantages. Implementation of the system call interceptor as a loadable kernel module intercepting some system calls is a controversial solution. The interception occurs before the system call reaches the VFS kernel subsystem. Additional logic will apply to all invocations of this system call, which may lead to dramatic performance degradation of the whole operating system. Implementing the system call interceptor as a new file system kernel module used in conjunction with OverlayFS is the best option in terms of architecture and performance. The tiered file system's underlying file system and the new file system can be mounted as "upper" and "lower" file systems using OverlayFS. There will be negligible overhead due to accessing files in the hot tier; the overhead for accessing files in the cold tier will depend on the daemon implementation efficiency and network bandwidth. This solution has a complication common for any kernel code—the Linux kernel is constantly changing and guarantees neither a stable internal API nor ABI; consequently, code compilation and adjustments for each new Linux kernel release will be required. Implementation of the system call interceptor as a FUSE file system is an elegant solution. It allows implementing all additional logic for system calls in a user space, but it imposes overheads due to additional context switches and memory copies between kernel and user spaces [50]. The system call interceptor implemented as a dynamically loaded shared library is a slightly more efficient solution than the FUSE file system, because it has two context switches less. However, it also has substantial shortcomings—it intercepts each of the special system calls made by the tiered file system client, even to files residing on other file systems, and does not intercept system calls made by statically linked programs. Notwithstanding these drawbacks, in this study, the dynamically loaded library approach has been chosen for the system call interceptor implementation because of its simplicity, portability, and the ability of quick prototyping. A similar approach for system call interception has been taken in [24].

III.2.2 Data Migration

The file demotion process includes migration of file data to the cold tier. Similarly, the file promotion process includes migration of file data to the hot tier. File data residing in the cold tier cannot be read and written directly with POSIX; therefore, when the tiered file system client accesses the file, the file data should be migrated to the hot tier first. Obviously, a client requesting a read of a byte range in the middle of the file should be blocked until this byte range is available in the hot tier. In the simplest case, read and write operations are allowed only after all file data is migrated to the hot tier. However, reads of file data residing in the cold tier can be optimized, for example, using object retrieval in parts by the Amazon S3 protocol [5]. Migration of large files to the cold tier can also be optimized with the Multipart Upload S3 feature. Other object storage protocols exist, but they are not considered in this study.

The system call interceptor intercepts file system requests made by clients and, based on the requested file location, contacts the daemon. The file location is defined in the file metadata using the extended attributes. Extended attributes are often used to provide additional functionality to a file system [64]. The tiered component makes use of extended attributes to designate the tier in which files reside. A file resides in the hot tier when both its data and metadata is in the hot tier. A file resides in the cold tier when its data is in the cold tier and metadata is in the hot tier. The tiered component's data migration protocol ensures file data integrity.

Protocol. In the following protocol description, a file residing in the hot tier is referred as the *original file* and a file residing in the cold tier is referred as the *stub file*. Extended attributes are accessed as atomic objects [64]. Atomicity of extended attributes enables atomic transitions between original and stub file states. The tiered component uses the following extended attributes:

- `lock`: This is an undefined extended attribute that is used by daemons to synchronize their activities. (A distributed tiered file system includes multiple daemons.)
- `object_id`: This is a defined extended attribute that is used to store an object identifier corresponding to file data.
- `stub`: This is an undefined extended attribute that is used as a file location indicator. Its absence indicates an original file and its presence indicates a stub file.
- `size`: This is a defined extended attribute that stores the file data

size in bytes as if all file data was in the hot tier; namely, the value of the `st_size` member of `struct stat` [53].

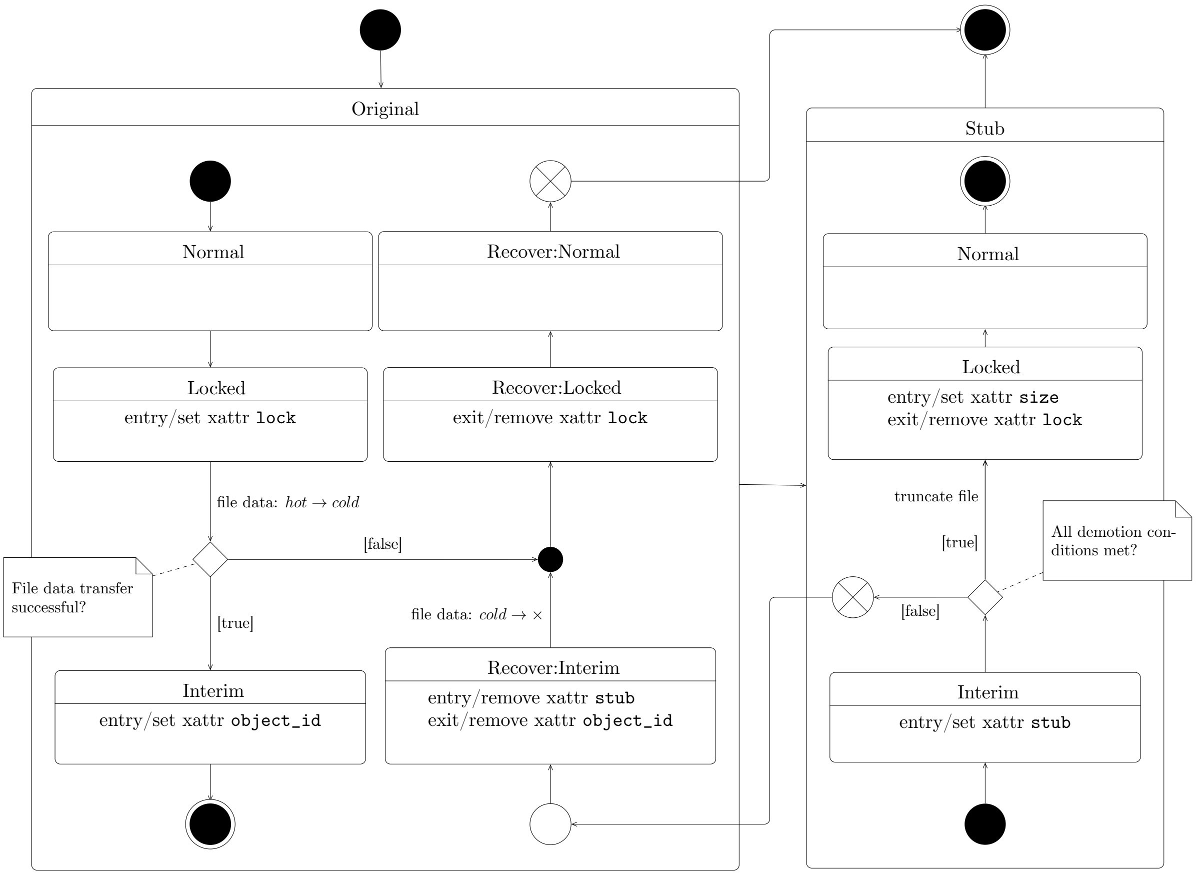


Figure 2: State diagram of the file demotion process.

Demotion Protocol. The file demotion process is illustrated in Figure 2. When the policy processor schedules a file demotion task, the data mover starts migrating the file data to the cold tier. Since the file resides in the hot tier, it is an original file. The original file state has several substates in the context of the file demotion process: *normal*, *locked*, *interim*, *recover interim*, *recover locked*, and *recover normal*. When the file demotion process completes, the file becomes a stub file. The *stub* file state has several substates in the context of the file demotion process: *interim*, *locked*, and *normal*. The initial file state is (*original*, *normal*) when there are no extended attributes set. Then it atomically switches to the (*original*, *locked*) state by setting the `lock` undefined extended attribute. Then migration of file data to the cold tier starts, which can take quite a long time. In case of failure, a recovery process starts, providing transitions to the (*original*, *recover locked*) state and then to the (*original*, *recover normal*) state. It includes atomic removal of the `lock` extended attribute. Otherwise, in case of success, the file state atomically switches to (*original*, *interim*) by setting the `object_id` defined extended attribute. The attribute value is an identifier of an object representing the file data in the cloud object storage. Then the file becomes the stub file through an atomic state switch to (*stub*, *interim*) with the `stub` undefined extended attribute. Note that the file data resides in both tiers in this state. From this point on, all file access requests go through the daemon. The next transition is determined by a predicate composed of demotion policy conditions. Some of tiered file system clients might have accessed this file during previous state transitions. In this case, a recovery process starts, providing transitions to the (*original*, *recover interim*) state, then to the (*original*, *recover lock*) state, and then to the (*original*, *recover normal*) state. The process includes subsequent removals of the previously set extended attributes in the reverse order; the object containing the file data can optionally be removed from the cloud object storage to reduce storage charges. Otherwise, if the file still satisfies all demotion conditions, it is truncated [58] to zero length, and its state atomically switches to the (*stub*, *locked*) state by setting the `size` defined extended attribute. The attribute value equals to the `st_size` member of `struct stat` obtained prior to the file truncation. Then the last transition, to the (*stub*, *normal*) state, occurs via atomic removal of the `lock` extended attribute. After that, the file demotion process is considered complete.

Promotion Protocol. A file promotion process is illustrated in Figure 3. When the policy processor schedules a file promotion task, the data mover starts migrating the file data to the hot tier. Since the file resides in the cold tier, it is a stub file. The stub file state has several substates in the context of the file promotion process: *normal*, *locked*, *interim*, *recover locked*, and *recover normal*. When the file promotion process completes, the file becomes an original file. The *original* file state has several sub-

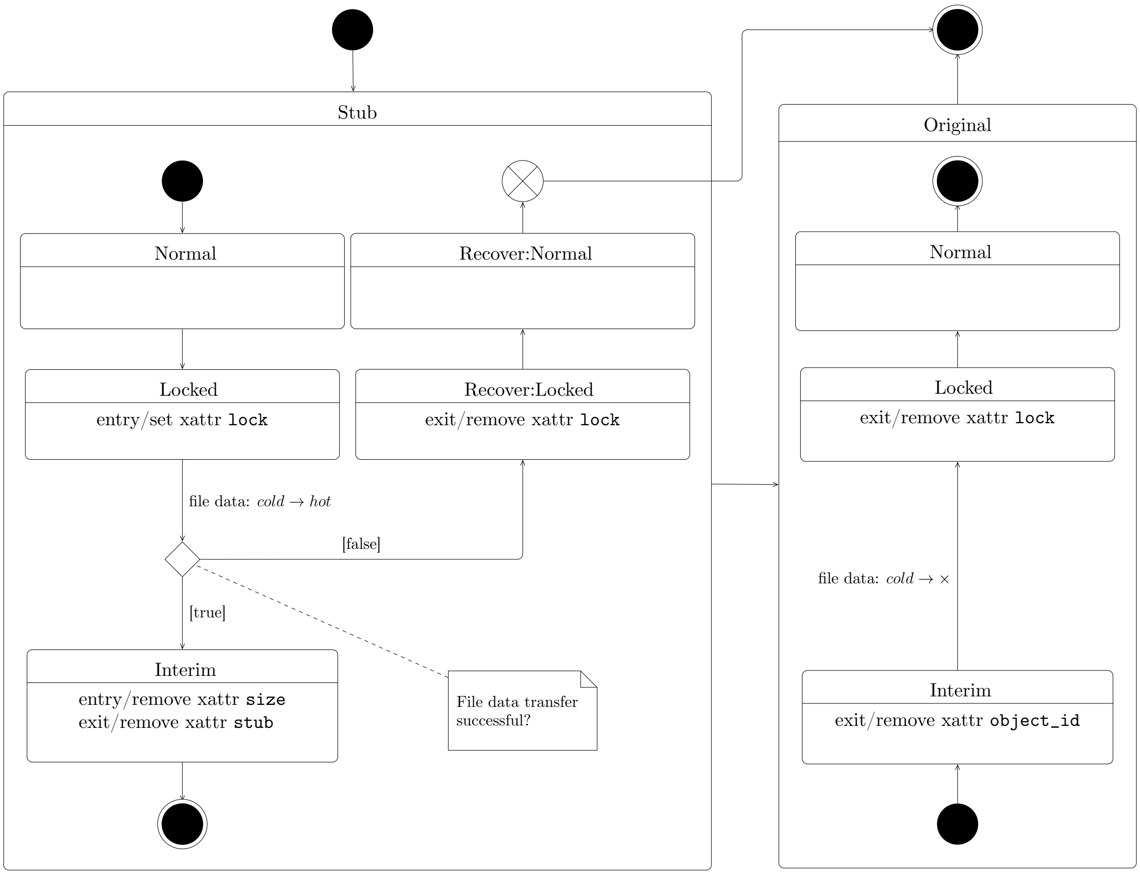


Figure 3: State diagram of the file promotion process.

states in the context of the file promotion process: *interim*, *locked*, and *normal*. The initial file state is (*stub*, *normal*) when there are the `size`, `stub`, and `object_id` extended attributes set. Then it atomically switches to the (*stub*, *locked*) state by setting the `lock` undefined extended attribute. Then migration of file data to the hot tier starts, which can take quite a long time. In case of failure, a recovery process starts, providing transitions to the (*stub*, *recover locked*) state and then to the (*stub*, *recover normal*) state. It includes atomic removal of the `lock` extended attribute. Otherwise, in case of success, the file state atomically switches to (*stub*, *interim*) by removing the `size` extended attribute. Then the file becomes an original file through an atomic state switch to the (*original*, *interim*) state via the stub extended attribute removal. From this point on, all file access requests go directly through the file system, avoiding interactions with the daemon. The next transition, to the (*original*, *locked*) state, occurs via atomic removal of the `object_id` extended attribute; the object containing the file data can optionally be removed from the cloud object storage to reduce storage charges. Then the file state atomically switches to the (*original*, *normal*) state via atomic removal of the `lock` extended attribute, which is the last state transition in the file promotion process. After that, the file promotion process is considered complete.

Theorem. *The file data migration protocol proposed above ensures file data integrity and guarantees that a file is processed by no more than one daemon at a time.*

Proof. The daemon starts either the file demotion or promotion process with setting the `lock` extended attribute atomically and ends the process with removal of the attribute. This guarantees exclusive access to this file among all daemons in the tiered file system.

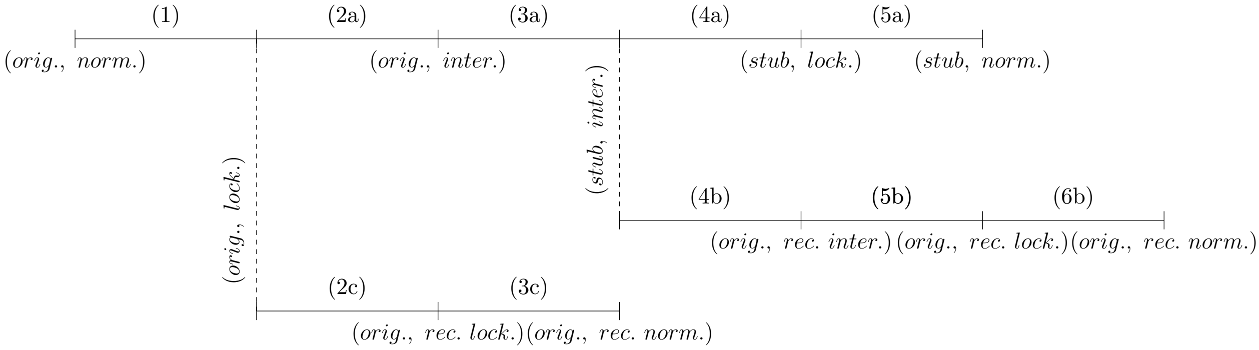


Figure 4: Demotion process timeline.

Consider the file demotion process in Figure 4. Client access to file data is not restricted during (1), (2a), (2c), (3a), (3c), (5b), and (6b) state transitions. During these transitions, the file is always in the (*original*, $*$) state, that is, there are no interactions between the system call interceptor and the daemon during file access. If an tiered file system client accesses the file during (1), (2a), and (3a) state transitions, the recovery process will be triggered, and the file will end up in the (*original*, *normal*) state. The file would not be truncated by the daemon if there were accesses to this file by other tiered file system clients. In cases (4a), (4b), and (5a), the system call interceptor will contact the daemon, which, in turn, guarantees exclusive access to the file among all daemons in the tiered file system. Thus, file integrity is ensured during the file demotion process.

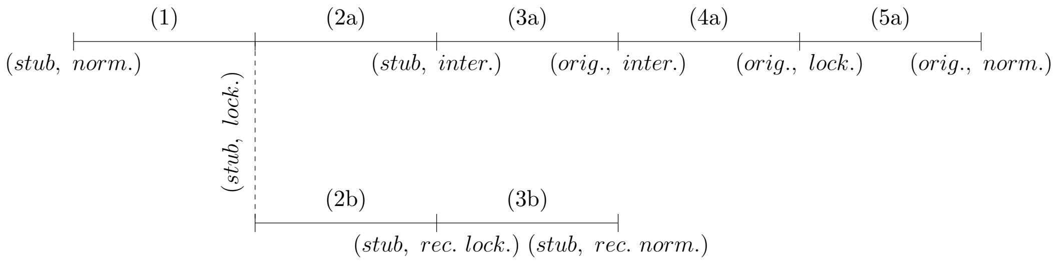


Figure 5: Promotion process timeline.

Consider the file promotion process in Figure 5. During (1), (2a), (2b), (3a), and (3b) state transitions, the system call interceptor will contact the daemon during file access. This guarantees exclusive access to the file among all daemons in the tiered file system. Client access to file data is not restricted during (4a) and (5a) state transitions. In these cases, the file data is already in the hot tier, and there will be no interactions between the system call interceptor and the daemon during file access. Thus, file integrity is ensured during the file promotion process. □

III.2.3 Policy Setting

A policy repository storing low-level policy rules, their conditions and actions, and related policy data is built into the policy processor. An example of a low-level policy condition is evaluation whether a file is of the regular type [32] or not. Examples of low-level policy actions are "demote file" and "promote file." Complex policies are defined using a configuration file that contains a DSL description of high-level policy rules [25]. This DSL is capable of describing static policy rules, which simply map policy conditions to policy actions, and adaptive policy rules, which are based on collected statistics of clients' access patterns for the particular tiered file system deployment. The policy definition DSL, expressing policy goals in a machine-readable way, is subject to further research.

III.2.4 System Call Interception

As noted earlier, there are several options for how to implement the system call interceptor. In this study, the dynamically loaded library option is considered. When discussing this option, it is more natural to say "wrap a system call" or "provide a system call replacement" instead of "intercept a system call," but the verb "intercept" will be used for consistency.

When an executable starts, the dynamic linker loads all of the shared libraries in the program's dynamic dependency list. It is possible to selectively override functions (and other symbols) that would normally be found by the dynamic linker using the rules described in [33]. To do so, one can define the environment variable `LD_PRELOAD` as a string consisting of space- or colon-separated names of shared libraries that should be loaded prior to any other shared libraries. Since these libraries are loaded first, any functions they define will automatically be used whenever required by the executable, thus overriding any other functions of the same name that the dynamic linker would otherwise have searched for [32].

In the light of the above, the system call interceptor can be implemented as a shared library, with the library name prepending the `LD_PRELOAD` variable value. In this case, all processes, except those that are statically linked with `libc.so.6` [34]—"standard C library," will use redefined versions of system calls with signatures listed below.

- `open()`-Family Calls

```
int open( const char *pathname, int flags, ... );
int openat( int dirfd, const char *pathname,
            int flags, ... );
```

- `stat()`-Family Calls

```
int stat( const char *pathname, struct stat *statbuf );
int lstat( const char *pathname, struct stat *statbuf );
int fstatat( int dirfd, const char *pathname,
             struct stat *statbuf, int flags );
```

- `truncate()`-Family Calls

```
int truncate( const char *path, off_t length );
```

Both `open()`-family calls will schedule demotion of the requested file if required. Their execution will last until the requested file becomes the original file, except for the cases when `O_NONBLOCK` or `O_NDELAY` flags are specified [39]. A sample implementation in a C-like pseudocode of the `open()` call is presented in Listing 1, Appendix A. All `stat()`-family calls replace the value of the the `st_size` member of `struct stat` with the value of the `size` defined extended attribute if required. A sample C-like pseudocode implementation of the `stat()` call is presented in Listing 2, Appendix A. Note that there is no need to intercept `stat()`-family system calls if the file system supports the `fallocate()` system call with the `FALLOC_FL_PUNCH_HOLE` mode argument [23]. In this case, one can use an atomic system call to make the file sparse. Most file systems do not allocate space for sparse files, so there is no storage overhead. Interception of the `truncate()` system call enables very useful optimization—if the file was either shrunk or extended, one can record a data end offset in the defined extended attribute `data_offset` (optional) and migrate the required data range only during the file promotion process[8]. A sample C-like pseudocode implementation of the `truncate()` call is presented in Listing 3, Appendix A.

III.2.5 Deployment

The use of extended attributes as a synchronization mechanism makes it possible to run multiple daemons per tiered file system and enables the use of the tiered component in conjunction with a distributed file system. In a distributed environment, the tiered component comprises several daemons and system call interceptors; namely, one daemon and system call interceptor per node in a cluster. Each daemon, which contains a file system scanner, can be configured to scan a single subtree of the directory tree to speed up identification of candidate files for demotion or promotion. This configuration does not need to be static. Fault-tolerant solutions suitable for distributed configuration management, such as Apache ZooKeeper [7] and etcd [66], can be utilized to dynamically map subtrees of the directory tree to daemons' file system scanners. Apache Zookeeper and etcd can also be used as a communication mechanism between daemons, for example, for load balancing of file data migration tasks, as network resources on some nodes may be overutilized, and others underutilized.

[8] The use of the `data_offset` extended attribute is optional and does not affect the correctness of the data migration protocol if handled properly in the code.

The deployment diagram for the tiered file system is shown in Figure 6.

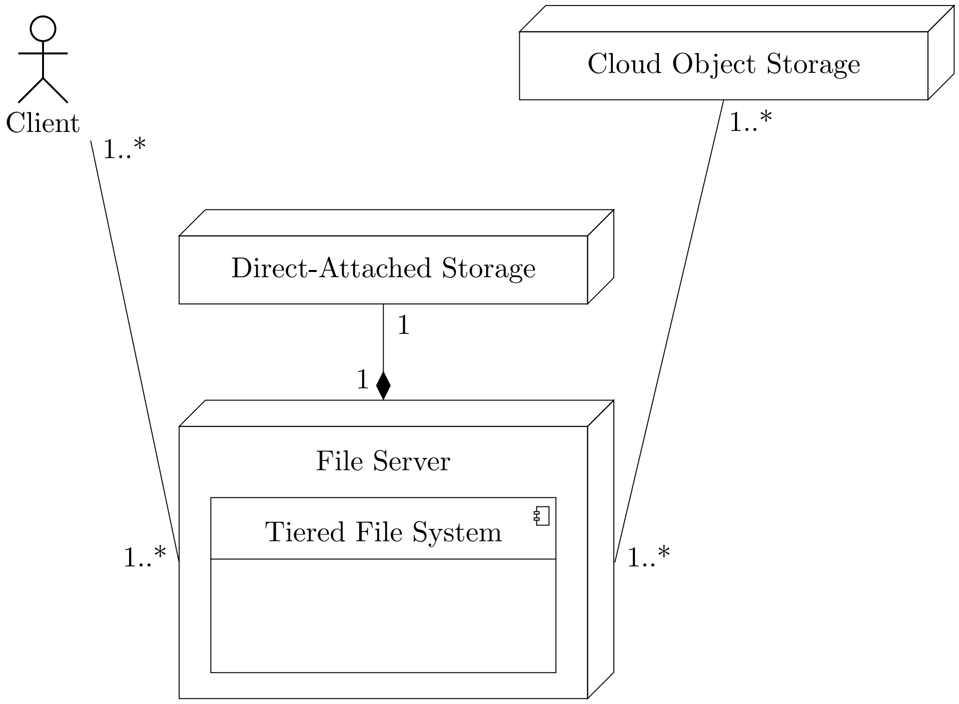


Figure 6: Deployment diagram of the tiered file system.

III.2.6 Possible Improvements and Optimizations

As discussed in Section I.4.2, one of the main problems that arises during development of tiered file systems is the weakened consistency model of the cloud object storage tier (depends on cloud object storage). In the tiered file system, this problem can be addressed using the `hash` defined extended attribute. This attribute stores a hash of file data calculated during demotion. The stored `hash` value can be used during the file promotion process to verify that the data is of the latest version. The versioning feature offered by some cloud object storage systems, such as Amazon S3 [5], can also be used for this purpose. In this case, it is more natural to name the extended attribute as `version` instead of `hash`. This concept is similar to the *consistency anchors* concept proposed in [51]; the file system will act as a strongly consistent store for metadata (consistency anchor) in the tiered file system.

Files can be quite big. If it is known in advance that the tiered file system will store many large files, it is reasonable to split large files into chunks and promote file chunks instead of the whole file. Such an approach requires addition of extended attributes check to each `read()` and `write()` system call invocation. If the system call interceptor is implemented as a dynamically loaded shared library or as a FUSE file system, these checks will cause dramatic performance degradation. However, this might be a good option for the system call interceptor implemented as a file system kernel module to be used with OverlayFS. In the latter case, there is no overhead due to additional context switches between user and kernel spaces. This is particularly useful for resolution of the graphical file managers' problem discussed in Section I.4.2. Upon file access, the file promotion process starts, which now comprises several stages of chunk promotions determined by the number of chunks in the file. As soon as the first chunk of the file is migrated to the hot tier, the graphical file manager can read the first few kilobytes and close the file. On `close()`, the daemon can abort the file promotion process leaving the rest of file chunks in the cold tier. It is worth noting that, in most cases, such "on-`close()`" behavior should not be applied to the whole directory tree of the file system, but rather to individual directories which are known to store large collections of images and videos. The access latency could also be decreased via interception of the `posix_fadvise()` system call [44]. When the client application takes advantage of this system call, the daemon can migrate file chunks corresponding to the `offset` and `len` parameters first.

Nonetheless, it is strongly believed that the maximum performance increase can be gained only via careful adjustment of data migration policies by either the system administrator or the policy processor itself, if it implements some advanced adaptive policy.

IV Implementation

The tiered file system proposed in Section III.2 has been implemented and its performance was evaluated with the BtrFS [11] local disk file system and the OrangeFS [40] parallel file system on a four-node cluster. This section provides an overview of the implementation and then discusses performance evaluation results. The developed tiered file system is called CloudTieringFS. The implementation is open source, licensed under the GNU General Public License v3.0 (GPLv3) [26] and freely available on GitHub[9].

IV.1 Overview

CloudTieringFS is developed in the C language. It consists of only 5361 lines of code, including comments and excluding documentation and license files. It depends on the `libs3` library [67], which is used to work with S3-compatible cloud object storage systems, and the `dotconf` library [65], which is used to parse configuration files. POSIX Threads [12] are used to enable parallelism.

The components are as proposed in Section III.2: the daemon, which contains the policy processor, monitor, file system scanner and data mover, and the system call interceptor in the form of a dynamically loaded shared library. There is also a test suit covering most of the implemented functionality. The number of threads belonging to the data mover is configurable. As of now, each file demotion/promotion task runs in its own thread; this mechanism will be replaced with `epoll()` [21], leaving only two threads for the data mover—one for file demotions and the other for file promotions. Currently, there are two policies implemented:

1. Demote files that have not been accessed for X seconds and promote files on demand.
2. Demote files that have not been accessed for X seconds and promote all files in the directory in case the client accessed at least one file in

[9] https://github.com/aoool/CloudTieringFS

that directory.

The policies are defined in the code; the policy definition DSL is subject to further research. The implementation passes the *pjdfstest* [68] POSIX conformance test.

IV.2 Performance

CloudTieringFS performance was evaluated in single- and multi-node configurations. For the single-node configuration, the BtrFS file system was used as the file system tier. For the multi-node configuration, the OrangeFS file system was used as the file system tier. Amazon EC2 [4] was used to deploy compute nodes and Amazon S3 [5] was used as the cloud object storage tier. Compute nodes and S3 object storage were located in North Virginia. SUSE Linux Enterprise Server 12 SP2 were installed on all nodes. Nodes were of the *t2.micro* type and had 1 virtual CPU, 1 GiB RAM and 30 GB of EBS[10] space. For OrangeFS, a four-node cluster was deployed containing three nodes with the OrangeFS Server and one node with the OrangeFS Client[11].

The following performance test scenario was performed: open and read all files in the directory in the order in which `readdir()` [47] produces the files. According to observations, BtrFS's `readdir()` lists the files in random order, while OrangeFS's `readdir()` lists the files sorted by file creation time. For both BtrFS and OrangeFS, file access latencies were measured as follows:

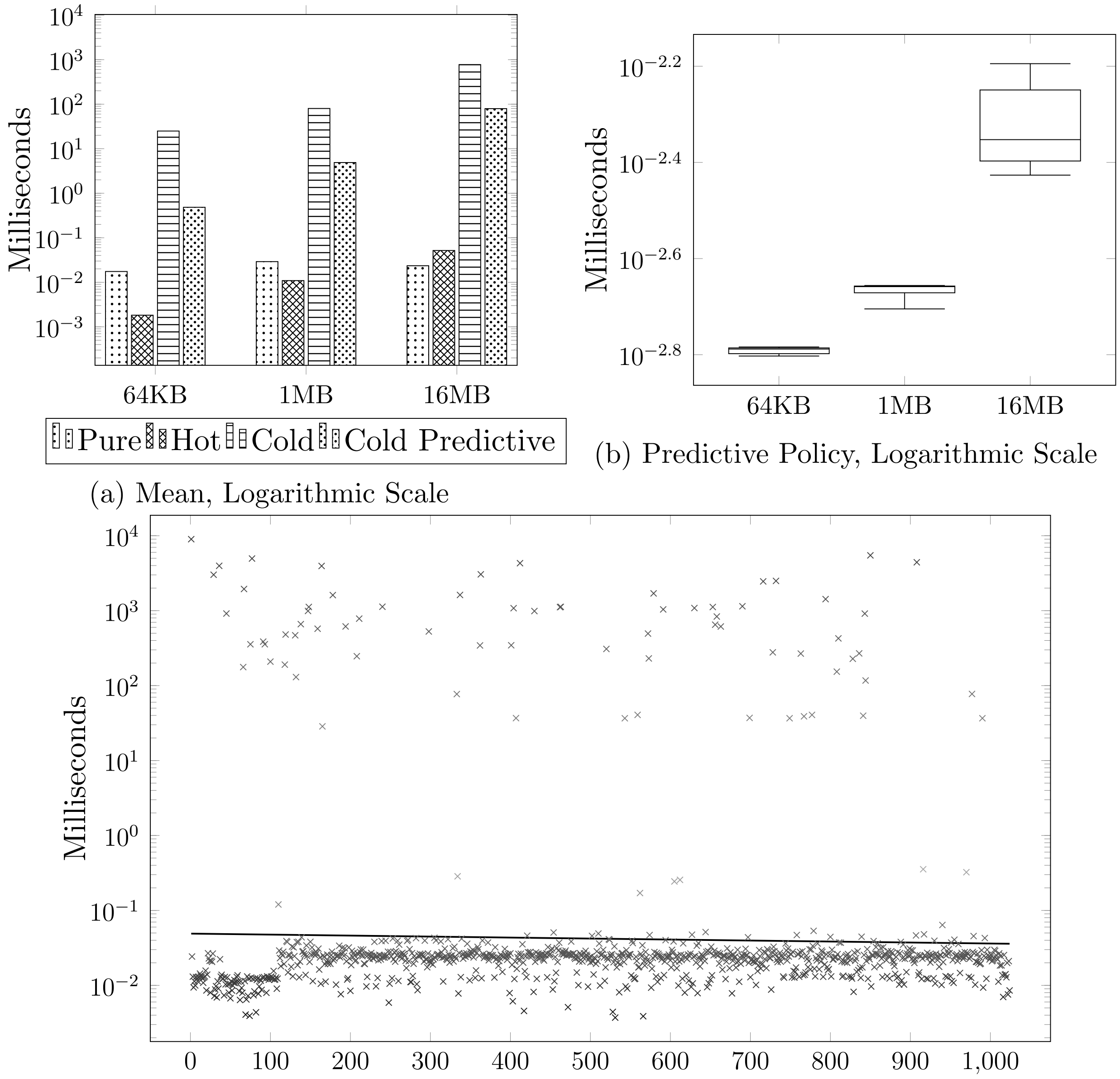
- "pure" file system,
- tiered file system when files reside in the hot tier,
- tiered file system when files reside in the cold tier and on-demand file promotion policy—policy (1)—is used, and
- tiered file system when files reside in the cold tier and the predictive file promotion policy—policy (2)—is used.

The results of this performance test are analyzed in the following sections.

[10] Elastic Block Storage

[11] The OrageFS Client was installed on a separate compute node to save some memory; there was only 1GiB of RAM available.

IV.2.1 Single Node (BtrFS)



(a) Mean, Logarithmic Scale

(b) Predictive Policy, Logarithmic Scale

(c) Predictive Policy, Logarithmic Scale, 16MB

Figure 7: `open()` system call latency (BtrFS).

CloudTieringFS having BtrFS as the file system tier was tested with three file sizes: 64KB, 1MB, and 16MB. For each test run, the directory contained 1024 files of the selected size. There were four test runs per file size: open and read 1024 files on "pure" BtrFS, open and read 1024 files in the BtrFS tier of CloudTieringFS, open and read 1024 files in the Amazon S3 tier of CloudTieringFS while the daemon implements policy (1), and open and read 1024 files in the Amazon S3 tier of CloudTieringFS while the daemon implements policy (2). The results are shown in Figure 7. Figure 7a is a bar plot presenting the mean file access latencies for various cases. Figure 7b presents box plots corresponding to the file access latencies for each size while the daemon implements policy (2); lower and upper whiskers are the second and ninety-eighth percentiles; outliers are not shown. Figure 7c is a scatter plot presenting the distribution of file access latencies for 16MB case while the daemon implements policy (2). All three plots have time axis of logarithmic scale.

Consider Figure 7a. The first two bars in each bar group correspond to "pure" BtrFS file system and BtrFS tier of the CloudTieringFS. Since BtrFS is a local disk file system and system call overheads are small, it is not clear in which configuration the `open()` system call runs faster. In theory, "pure" BtrFS's `open()` executes faster, because in CloudTieringFS's implementation `open()` performs several additional system calls, causing a number of user/kernel space context switches. But the compute node was an Amazon EC2 virtual machine with limited resources, so performance deviations were expected. The third bar in each bar group corresponds to the mean file access latency with on-demand file access. An average network bandwidth can be derived from this value. The fourth bar corresponds to the mean file access latency while the daemon implements predictive policy—policy (2). For 64KB files, performance increased 51 times, for 1MB files it increased 16 times, and for 16MB files there was a 10-times performance gain in comparison with on-demand access to Amazon S3 tier. There is also some important statistics: the difference between the latencies of the file accesses to the first file in a sequence of 1024 files in predictive and on-demand Amazon S3 tier cases. The first access to the 64KB file takes 33 milliseconds longer than in on-demand case, to the 1MB file it was 191 milliseconds longer, and to the 16MB file—8 seconds longer. These performance degradations are in more than payback with the next file accesses.

Consider Figure 7b. As noted earlier, the outliers are not shown and the lower and upper whiskers are the second and ninety-eighth percentiles. This means that there were less than 20 file accesses of 1024 lasting longer than 1 millisecond for each file size.

Consider Figure 7c. It shows the distribution of `open()` system call latencies for the 16MB file sizes and predictive policy. Most of the latencies are less than a millisecond. The overlaid linear prediction plot has negative slope, which means that the probability to open file faster is higher for the files in the second half of the sequence of 1024 files. This means that the daemon performs file promotions faster than the performance test application reads the sequence of 1024 files. (In the test run, the daemon's data mover had 20 threads for the file promotion tasks execution.)

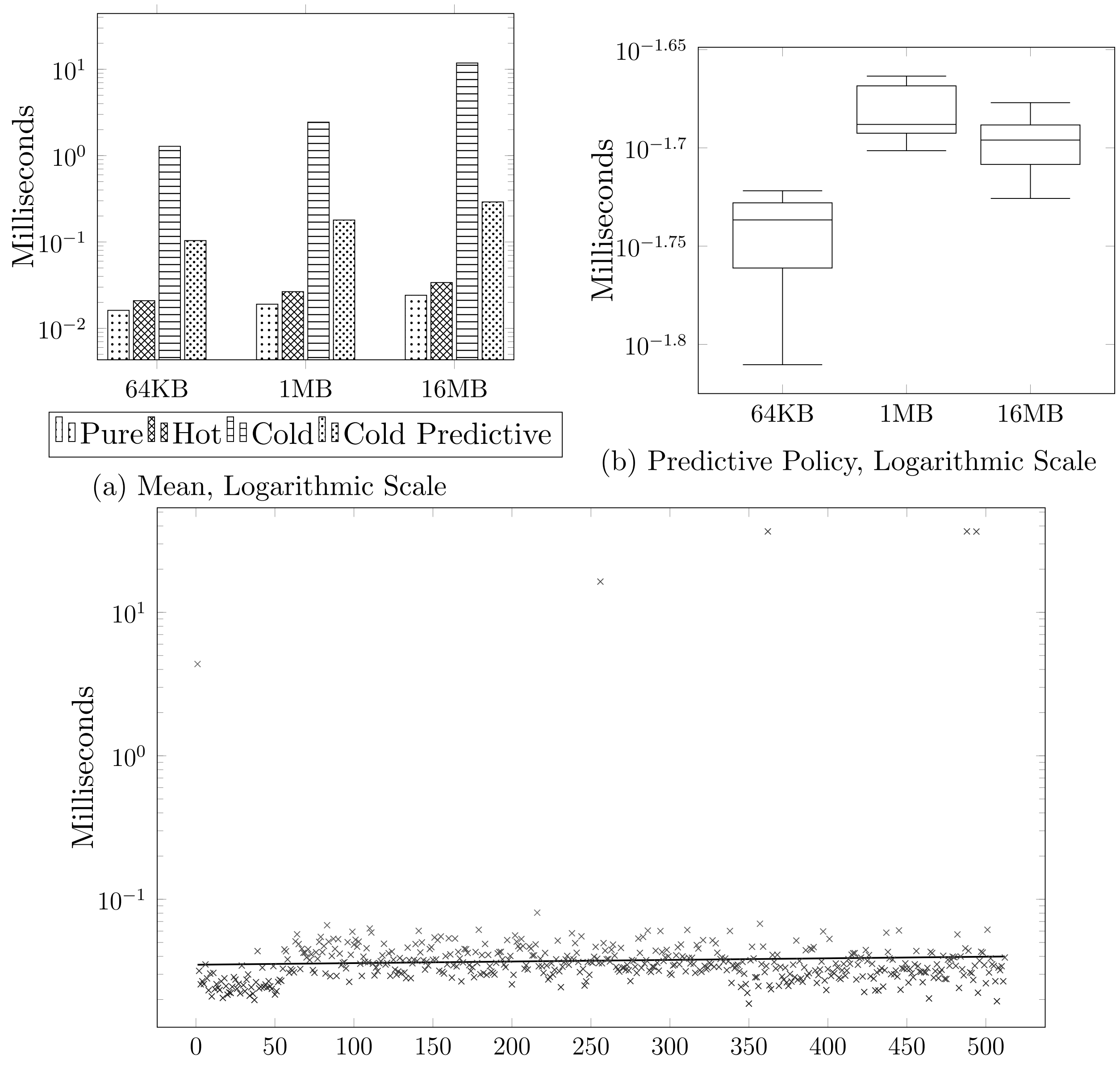
Some statistics for "pure" BtrFS and predictive CloudTieringFS file access latencies is shown in Table 2. The first value corresponds to the "pure" BtrFS case and the second corresponds to the predictive CloudTieringFS with BtrFS tier case.

(ms)	64KB	1MB	16MB
Mean	0.01748 | 0.48379	0.02913 | 4.89184	0.02353 | 79.53151
1st Quartile	0.00125 | 0.00159	0.00413 | 0.00213	0.0191 | 0.00401
Median	0.0013 | 0.00163	0.00608 | 0.0022	0.01968 | 0.00444
3rd Quartile	0.0013 | 0.00164	0.00649 | 0.0022	0.02008 | 0.00563
Variance	0.25811 | 40.89284	0.14525 | 431.14388	0.0 | 257196.85321
Std. Dev.	0.50805 | 6.39475	0.38112 | 20.764	0.00215 | 507.14579

Table 2: Comparison of "pure" BtrFS and CloudTieringFS with BtrFS tier.

IV.2.2 Multiple Nodes (OrangeFS)

CloudTieringFS having OrangeFS as the file system tier was tested with three file sizes: 64KB, 1MB, and 16MB. For each test run, the directory contained 512 files of the selected size. There were four test runs per file size: open and read 512 files on "pure" OrangeFS, open and read 512 files in the OrangeFS tier of CloudTieringFS, open and read 512 files in the Amazon S3 tier of CloudTieringFS while the daemon implements policy (1), and open and read 512 files in the Amazon S3 tier of CloudTieringFS while the daemon implements policy (2). The results are shown in Figure 8. Figure 8a is a bar plot presenting the mean file access latencies for various cases. Figure 8b presents box plots corresponding to the file access latencies for each size while the daemon implements policy (2); lower and upper whiskers are



(a) Mean, Logarithmic Scale

(b) Predictive Policy, Logarithmic Scale

(c) Predictive Policy, Logarithmic Scale, 16MB

Figure 8: `open()` system call latency (OrangeFS).

the second and ninety-eighth percentiles; outliers are not shown. Figure 8c is a scatter plot presenting the distribution of file access latencies for 16MB case while the daemon implements policy (2). All three plots have time axis of logarithmic scale.

Consider Figure 8a. The first two bars in each bar group correspond to “pure” OrangeFS file system and OrangeFS tier of the CloudTieringFS. OrangeFS is a distributed parallel file system and system call overheads are considerable, it is clear that the `open()` system call runs faster on “pure” OrangeFS, which meets the theoretical expectations. The third bar in each bar group corresponds to the mean file access latency with on-demand file access. The fourth bar corresponds to the mean file access latency while the daemon implements predictive policy—policy (2). For 64KB files, performance increased 12 times, for 1Mb files it increased 14 times, and for 16MB files there was a 41-times performance increase in comparison with on-demand access to Amazon S3 tier. Why the results are different in comparison with BtrFS, where this ratio decreased with the file size increase? Perhaps, because of the design of OrangeFS which is optimized to work well for scientific applications in a cluster environment. BtrFS performs good with small files, while OrangeFS performs good with larger files by virtue of its parallel architecture. The differencies between the latencies of the file accesses to the first file in a sequence of 512 files in predictive and on-demand Amazon S3 tier cases are the following: 12 milliseconds longer than in on-demand case for 64KB file size, 14 milliseconds longer for 1MB file size, and 41 milliseconds longer for 16MB file size. Such performance degradation is insignificant, again, because of the parallel architecture of OrangeFS.

Consider Figure 8b. As noted earlier, the outliers are not shown and the lower and upper whiskers are the second and ninety-eighth percentiles. This means that there were less than 11 file accesses of 512 lasting longer than 1 millisecond for each file size.

Consider Figure 8c. It shows the distribution of `open()` system call latencies for the 16MB file sizes and predictive policy. Nearly all measurements are less than 0.1 millisecond. There are only a couple of outliers. The average read time of the 16MB size was 10 milliseconds. The overlaid linear prediction plot has negative slope, which means that the probability to open file faster is higher for the files in the second half of the sequence of 512 files. One can conclude that the daemon's data mover performs file promotions faster than the performance test application reads the sequence of 512 files. Such good results may also be caused by the order in which `readdir()` produces file names. The performance test application and the CloudTieringFS's daemon use this function to read directory entries, so the daemon schedules file promotions in the same order as the performance test application accesses the files. (In the test run, the daemon's data mover had 20 threads for the file promotion tasks execution.)

Some statistics for "pure" OrangeFS and predictive CloudTieringFS file access latencies is shown in Table 3. The first value corresponds to the "pure" OrangeFS case and the second corresponds to the predictive CloudTieringFS with OrangeFS case.

(ms)	64KB	1MB	16MB
Mean	0.01744 | 0.10378	0.01903 | 0.17946	0.02415 | 0.29098
1st Quartile	0.01313 | 0.01733	0.01514 | 0.02029	0.02012 | 0.01957
Median	0.01329 | 0.01833	0.01515 | 0.0205	0.02037 | 0.02013
3rd Quartile	0.01339 | 0.01871	0.01518 | 0.02145	0.0205 | 0.02049
Variance	0.0 | 2.59954	0.0 | 5.24181	0.0 | 8.34067
Std. Dev.	0.00438 | 1.61231	0.00396 | 2.2895	0.00402 | 2.88802

Table 3: Comparison of "pure" OrangeFS and CloudTieringFS with OrangeFS tier.

IV.2.3 Summary

The main goal of the performed performance test was to prove that carefully adjusted automated storage tiering policies can make the performance of the tiered file system comparable to the original file system's. A certain file access pattern was chosen and corresponding policies were implemented. It was shown that the performance of CloudTieringFS with the BtrFS tier is comparable to the original BtrFS performance. The performance of CloudTieringFS with the OrangeFS tier is nearly identical to the original OrangeFS performance.

The excellent performance results inspire to further research in the field of automated storage tiering policies. To make CloudTieringFS suitable for production use, additional policy rules should be implemented. This means that the policy definition DSL should be devised to allow system administrators to configure CloudTieringFS specific usage patterns.

Conclusion

Results

As a result of this study, the following tasks were accomplished[12]:

(I) Problems of automated storage tiering in an environment that includes a distributed file system and cloud object storage were investigated.

(II) Important features of modern distributed file systems (MooseFS, CephFS, GlusterFS, OrangeFS) related to the automated storage tiering were identified and compared.

(III) A software component enabling automated storage tiering between a POSIX-conformant file system and cloud object storage was designed. The component is file system-agnostic and can be used in a distributed environment.

(IV) The designed software component was implemented and its performance evaluated in single- and multi-node configurations. It was shown that carefully adjusted automated storage tiering policies can preserve the underlying file system's performance with neglectable overheads.

Thus, the primary aim of this study, to design and implement a file system-agnostic, policy-based software component responsible for data synchronization between a POSIX-conformant file system and cloud object storage, has been achieved.

[12] Each task has a corresponding section with the same Roman number.

Future Work

This study unveils a number of directions for future research and development.

The largest research topic is in the field of automated storage tiering policies. A clear and concise DSL capable of describing storage tiering policies in a general form could be developed. Machine learning could be utilized for intelligent file promotions to improve the system ability to adapt to new workloads and predict file accesses.

There is room for implementation improvements as well. The system call interceptor in a form of a new file system kernel module to be used in conjunction with OverlayFS could be implemented and compared with the dynamically loaded shared library implementation. Various optimizations proposed in Section III.2.6 could be implemented. `epoll()` mechanism could be implemented in the data mover to reduce the duration of data transitions between tiers. An eventually consistent backup mode could be introduced with relatively small amount of modifications. The developed tiered file system could be integrated with a cluster resource manager to facilitate migration tasks scheduling. The implementation needs further testing with POSIX-conformant file systems in different configurations in order to ensure its stability.

Any contribution to the project is welcome. The reference implementation used in this study can be found on GitHub[13].

[13] https://github.com/aoool/CloudTieringFS

References

[1] URL: http://www.moosefs.org.

[2] URL: https://www.gluster.org (online; accessed: 05/29/2017).

[3] Alan G. Yoder. — S3 and CDMI: A CDMI Guide for S3 Programmers. — Storage Networking Industry Association. — URL: https://www.snia.org/sites/default/files/S3-like_CDMI_v1.0.pdf (online; accessed: 05/23/2017).

[4] Amazon Web Services. — Amazon Elastic Compute Cloud: User Guide for Linux Instances, 2017.

[5] Amazon Web Services. — Amazon Simple Storage Service Developer Guide. — API Version 2006-03-01.

[6] Andrew S. Tanenbaum, Maarten Van Steen. Distributed Systems: Principles and Paradigms, Second Edition. — Pearson Prentice Hall, 2007. — P. 686.

[7] Apache ZooKeeper, The Apache Software Foundation. — URL: https://zookeeper.apache.org/ (online; accessed: 05/17/2017).

[8] Architecture of a Next-Generation Parallel File System. — Presentation. — URL: https://www.socallinuxexpo.org/scale11x-supporting/default/files/presentations/OrangeFS%20-%20Scale11x.pdf (online; accessed: 05/29/2017).

[9] Baweja Kunal. Intercept System Calls. — URL: https://github.com/bawejakunal/Intercept-System-Calls (online; accessed: 05/13/2017).

[10] Brent Welch. What is a Cluster Filesystem? — URL: http://www.beedub.com/clusterfs.html (online; accessed: 05/21/2017).

[11] BtrFS: Main Page. — URL: https://btrfs.wiki.kernel.org (online; accessed: 05/27/2017).

[12] Butenhof David R. Programming with POSIX Threads. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1997. — ISBN: 0-201-63392-2.

[13] Ceph File System, Red Hat, Inc, and contributors. — URL: http://docs.ceph.com/docs/master/cephfs/ (online; accessed: 05/29/2017).

[14] Inktank Storage, Inc. and contributors. — CephFS: Differences from POSIX. — URL: http://docs.ceph.com/docs/jewel/cephfs/posix/ (online; accessed: 05/21/2017).

[15] Choosing an Object Storage Protocol, Dell EMC. — URL: https://community.emc.com/docs/DOC-33680 (online; accessed: 05/23/2017).

[16] Dell EMC 2 TIERS, Dell EMC. — URL: http://orangefs.com/2tiers (online; accessed: 05/22/2017).

[17] Dell EMC CloudArray, Dell EMC. — URL: https://www.emc.com/storage/cloudarray/index.htm (online; accessed: 05/22/2017).

[18] Depardon Benjamin, Le Mahec Gaël, Séguin Cyril. Analysis of Six Distributed File Systems. — 2013.

[19] Coulouris George, Dollimore Jean, Kindberg Tim, Blair Gordon. Distributed Systems: Concepts and Design (5th edition). — 2012.

[20] Dell EMC. — EMC CloudArray (Version 6.0): Administrator Guide (Revision 1.1), 2015. — URL: https://uk.emc.com/collateral/TechnicalDocument/docu60786.pdf (online; accessed: 05/26/2017).

[21] EPOLL(7) Linux Programmer's Manual EPOLL(7). — URL: http://man7.org/linux/man-pages/man7/epoll.7.html (online; accessed: 05/27/2017).

[22] Eijkhout Victor. Introduction to High Performance Scientific Computing. — Lulu.com, 2012. — ISBN: 1257992546, 9781257992546.

[23] FALLOCATE(2) Linux Programmer's Manual FALLOCATE(2). — URL: http://man7.org/linux/man-pages/man2/fallocate.2.html (online; accessed: 05/19/2017).

[24] Fast In-Memory Checkpointing with POSIX API for Legacy Exascale-Applications / Jan Fajerski, Matthias Noack, Alexander Reinefeld et al. // Software for Exascale Computing - SPPEXA 2013-2015 / Ed. by Hans-Joachim Bungartz, Philipp Neumann, Wolfgang E. Nagel. — Cham : Springer International Publishing, 2016. — P. 427–441. — ISBN: 978-3-319-40528-5. — URL: http://dx.doi.org/10.1007/978-3-319-40528-5_19.

[25] Fowler Martin. Domain-Specific Languages. — Pearson Education, 2010.

[26] GNU General Public License. — URL: https://www.gnu.org/licenses/gpl-3.0.en.html (online; accessed: 05/27/2017).

[27] GlusterFS: Tiering. — URL: http://staged-gluster-docs.readthedocs.io/en/release3.7.0beta1/Features/tier/ (online; accessed: 05/29/2017).

[28] Google Cloud Storage: Consistency. — URL: https://cloud.google.com/storage/docs/consistency (online; accessed: 05/24/2017).

[29] Hamilton Eric. JPEG File Interchange Format. — 2004.

[30] IBM Spectrum Scale. — URL: https://www-03.ibm.com/systems/storage/spectrum/scale/ (online; accessed: 05/29/2017).

[31] Kearney Keven T., Torelli Francesco. The SLA Model // Service Level Agreements for Cloud Computing / Ed. by Philipp Wieder, Joe M. Butler, Wolfgang Theilmann, Ramin Yahyapour. — New York, NY : Springer New York, 2011. — P. 43–67. — ISBN: 978-1-4614-1614-2. — URL: http://dx.doi.org/10.1007/978-1-4614-1614-2_4.

[32] Kerrisk Michael. The Linux Programming Interface. — No Starch Press, 2010. — P. 1506.

[33] LD.SO(8) Linux Programmer's Manual LD.SO(8). — URL: http://man7.org/linux/man-pages/man8/ld.so.8.html (online; accessed: 12/26/2016).

[34] LIBC(7) Linux Programmer's Manual LIBC(7). — URL: http://man7.org/linux/man-pages/man7/libc.7.html (online; accessed: 05/15/2017).

[35] MMAP(2) Linux Programmer's Manual MMAP(2). — URL: http://man7.org/linux/man-pages/man2/mmap.2.html (online; accessed: 05/29/2017).

[36] MarFS. — URL: https://github.com/mar-file-system/marfs (online; accessed: 05/26/2017).

[37] MarFS Requirements, Design, Configuration, and Administration. — URL: https://raw.githubusercontent.com/mar-file-system/marfs/master/Documentation/MarFS-Requirements-Design-Configuration-Admin.docx (online; accessed: 05/26/2017).

[38] Miklos Szeredi, Nikolaus Rath, et al. libfuse. — URL: https://github.com/libfuse/libfuse (online; accessed: 05/13/2017).

[39] OPEN(2) Linux Programmer's Manual OPEN(2). — URL: http://man7.org/linux/man-pages/man2/open.2.html (online; accessed: 05/16/2017).

[40] The OrangeFS Project. — URL: http://www.orangefs.org/ (online; accessed: 05/21/2017).

[41] The OrangeFS Project: Frequently Asked Questions. — URL: http://www.orangefs.org/faq/ (online; accessed: 05/21/2017).

[42] Overlay Filesystem. — URL: https://github.com/torvalds/linux/blob/v4.11/Documentation/filesystems/overlayfs.txt (online; accessed: 05/17/2017).

[43] POSIX 1003.1 Frequently Asked Questions (FAQ Version 1.15). — URL: http://www.opengroup.org/austin/papers/posix_faq.html (online; accessed: 05/21/2017).

[44] POSIX_FADVISE(2) Linux Programmer's Manual POSIX_FADVISE(2). — URL: http://man7.org/linux/man-pages/man2/posix_fadvise.2.html (online; accessed: 05/26/2017).

[45] Percy Tzelnic, Sorin Faibish. 2 Tiers Architecture: POSIX-Like Namespace Layered above an Object Store. — Presentation at the EMC World 2016 Conference. — 2016. — URL: https://regmedia.co.uk/2016/11/18/octott_03_final.pdf (online; accessed: 05/26/2017).

[46] Cloud Standards Customer Council. — Practical Guide to Cloud Service Agreements, Version 2.0, 2015. — URL: http://www.cloud-council.org/deliverables/CSCC-Practical-Guide-to-Cloud-Service-Agreements.pdf (online; accessed: 05/24/2017).

[47] READDIR(3) Linux Programmer's Manual READDIR(3). — URL: http://man7.org/linux/man-pages/man3/readdir.3.html (online; accessed: 05/28/2017).

[48] RFC 3060: Policy Core Information Model – Version 1 Specification / B Moore, E Ellesson, J Strassner, A Westerinen // IETF, February. — 2001. — URL: https://tools.ietf.org/html/rfc3060 (online; accessed: 05/10/2017).

[49] RFC 3198: Terminology for Policy-Based Management / A Westerinen, J Schnizlein, J Strassner et al. // The Internet Engineering Task Force (IETF). — 2001. — URL: https://tools.ietf.org/html/rfc3198 (online; accessed: 05/10/2017).

[50] Rajgarhia Aditya, Gehani Ashish. Performance and Extension of User Space File Systems // Proceedings of the 2010 ACM Symposium on Applied Computing / ACM. — 2010. — P. 206–213.

[51] SCFS: A Shared Cloud-backed File System / Alysson Bessani, Ricardo Mendes, Tiago Oliveira et al. // Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference. — USENIX ATC'14. — Berkeley, CA, USA : USENIX Association, 2014. — P. 169–180. — URL: http://dl.acm.org/citation.cfm?id=2643634.2643652.

[52] SEEKDIR(3) Linux Programmer's Manual SEEKDIR(3). — URL: http://man7.org/linux/man-pages/man3/seekdir.3.html (online; accessed: 05/29/2017).

[53] STAT(2) Linux Programmer's Manual STAT(2). — URL: http://man7.org/linux/man-pages/man2/stat.2.html (online; accessed: 05/14/2017).

[54] Scott W. Ambler. UML 2 Component Diagrams: An Agile Introduction. — URL: http://agilemodeling.com/artifacts/componentDiagram.htm (online; accessed: 05/12/2017).

[55] Sergey Morozov. Cloud Object Storage Verifier. — 2015. — Bachelor's Thesis. URL: http://se.math.spbu.ru/SE/diploma/2015/bmo/444-Morozov-report.pdf (online; accessed: 05/21/2017).

[56] Shi Wei, Ju Dapeng, Wang Dongsheng. Saga: A Cost Efficient File System Based on Cloud Storage Service // Economics of Grids, Clouds, Systems, and Services: 8th International Workshop, GECON 2011, Paphos, Cyprus, December 5, 2011, Revised Selected Papers / Ed. by Kurt Vanmechelen, Jörn Altmann, Omer F. Rana. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2012. — P. 173–184. — ISBN: 978-3-642-28675-9. — URL: http://dx.doi.org/10.1007/978-3-642-28675-9_13.

[57] Storage Networking Industry Association, et al. SNIA Dictionary. — 2016. — P. 316.

[58] TRUNCATE(2) Linux Programmer's Manual TRUNCATE(2). — URL: http://man7.org/linux/man-pages/man2/truncate.2.html (online; accessed: 05/14/2017).

[59] Tanenbaum Andrew, Van Steen Maarten. Distributed Systems: Principles and Paradigms. — Pearson Prentice Hall, 2007. — P. 686.

[60] Ubuntu 17.04: File Manager Preview Preferences. — URL: https://help.ubuntu.com/stable/ubuntu-help/nautilus-preview.html (online; accessed: 05/24/2017).

[61] VIPR 2.1 - EMC VIPR Data Services: Geo-Protection and Multisite Access. — URL: http://www.emc.com/techpubs/vipr/geo_overview-2.htm (online; accessed: 05/27/2015).

[62] Vogels Werner. Eventually Consistent // Communications of the ACM. — 2009. — Vol. 52, no. 1. — P. 40–44.

[63] Vrable Michael, Savage Stefan, Voelker Geoffrey M. BlueSky: A Cloud-backed File System for the Enterprise // Proceedings of the 10th USENIX Conference on File and Storage Technologies. — FAST'12. — Berkeley, CA, USA : USENIX Association, 2012. — P. 19–19. — URL: http://dl.acm.org/citation.cfm?id=2208461.2208480.

[64] XATTR(7) Linux Programmer's Manual XATTR(7). — URL: http://man7.org/linux/man-pages/man7/xattr.7.html (online; accessed: 05/26/2016).

[65] dotconf. — URL: https://github.com/williamh/dotconf (online; accessed: 05/27/2017).

[66] Anthony Romano, Brandon Philips, Fanmin Shi et al. etcd. — URL: https://github.com/coreos/etcd (online; accessed: 05/17/2017).

[67] libs3. — URL: https://github.com/bji/libs3 (online; accessed: 05/27/2017).

[68] pjdfstest: File System Test Suite. — URL: https://github.com/pjd/pjdfstest (online; accessed: 05/29/2017).

Appendix A `open()`, `stat()`, and `truncate()` Listings

Listing 1: Sample implementation of the `open()` call in a C-like pseudocode. Error handling is omitted for brevity.

```
#define _GNU_SOURCE

#include <dlfcn.h>
#include <stdarg.h>
#include <sys/types.h>
#include <unistd.h>


/**
 * Open and possibly create a file.
 */
int open( const char *path, int flags, ... ) {
    /* obtain the address of the next symbol ‘‘open’’ occurrence
       using the default shared object search order */
    int (*next_open)( const char *, int, ... ) =
        dlsym( RTLD_NEXT, ”open” );

    /* get the mode agrument of the ‘‘open’’ variadic function,
       if available */
    mode_t mode;
    va_list ap;
    va_start( ap, flags );
    mode = va_arg( ap, mode_t );
    va_end( ap );

    /* open the file with the ‘‘next_open’’ */
    int fd = next_open( path, flags, mode );

    /* contact the daemon, if the file is a stub file */
    if ( is_stub_file( fd, flags ) ) {
        schedule_file_demotion( ”/proc/%ld/fd/%d”,
                                (long)getpid(), fd );
        wait_until_original_file( &fd, flags );
    }

    return fd;
}
```

Listing 2: Sample implementation of the `stat()` call in a C-like pseudocode. Error handling is ommited for brevity.

```
#define _GNU_SOURCE

#include <dlfcn.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>


/**
 * Get file status.
 */
int stat( const char *pathname, struct stat *statbuf ) {
    /* obtain the address of the next symbol ‘‘open’’ occurrence
       using the default shared object search order */
    int (*next_open)( const char *, int, ... ) =
        dlsym( RTLD_NEXT, "open" );

    /* obtain the address of the next symbol ‘‘fstat’’ occurrence
       using the default shared object search order */
    int (*next_fstat)( int, struct stat * ) =
        dlsym( RTLD_NEXT, "fstat" );

    /* use file descriptor for further operations to ensure
       that subsequent system calls process the same file */
    int fd;
    if ( ( fd = next_open( pathname, O_RDONLY ) ) == -1 ) {
        return -1;
    }

    /* with a small probability, may need to repeat the
       following sequence of system calls */
    do {
        /* initialize statbuf with the ``next_stat'' */
        if ( next_fstat( fd, statbuf ) == -1 ) {
            close( fd );
            return -1;
        }

        /* replace the st_size member of statbuf,
           if it is required */
        if ( is_stub_file( fd ) ) {
            off_t size;
            if ( get_stub_file_size( fd, &size ) == -1 ) {
                /* the sequence of system calls was executed
                   during file state transition
                   (stub, interim)->(stub, locked) or
                   between entry and exit actions in
                   (stub, interim) */
                continue;
            }

            statbuf->st_size = size;
        }

        close( fd );
        return 0;
    } while ( 1 );

    /* unreachable place */
    close( fd );
    return -1;
}
```

Listing 3: Sample implementation of the `truncate()` call in a C-like pseudocode. Error handling is omitted for brevity. Only “truncate to zero length” optimization is shown; other optimizations are possible.

```
#define _GNU_SOURCE

#include <dlfcn.h>
#include <fcntl.h>
#include <sys/types.h>
```

```
#include <sys/xattr.h>
#include <unistd.h>


/**
 * Truncate the file to the specified length.
 */
int truncate( const char *path, off_t length ) {
    /* obtain the address of the next symbol ``open'' occurrence
       using the default shared object search order */
    int (*next_open)( const char *, int, ... ) =
        dlsym( RTLD_NEXT, "open" );

    /* obtain the address of the next symbol ``ftruncate''
       occurrence using the default shared object search order */
    int (*next_ftruncate)( int, off_t ) =
        dlsym( RTLD_NEXT, "ftruncate" );

    /* use file descriptor for further operations to ensure
       that subsequent system calls process the same file */
    int fd;
    if ( ( fd = next_open( path, O_WRONLY ) ) == -1 ) {
        return -1;
    }

    /* set size value to zero, if it is required */
    if ( length == 0 ) {
        off_t size = 0;
        fsetxattr( fd, "user.size", &size, sizeof( off_t ),
                   XATTR_REPLACE );
    }

    /* truncate file with the ``next_ftruncate'' */
    int ret = next_ftruncate( fd, length );

    close( fd );
    return ret;
}
```