Bachelor Thesis

Performance and Energy Efficiency Analysis of Distributed File Systems on a Cluster of ARM-Based Single-Board Computers

Timm Leon Erxleben
timm.erxleben@ovgu.de

May 13, 2022

First Reviewer:
Jun.-Prof. Dr. Michael Kuhn

Second Reviewer:
Dr. Martin Köhler

Abstract

Ever-increasing demands on capacity and data throughput of storage clusters lead to growing systems and growing energy consumption. Currently, storage clusters are built from regular x86-based servers that are not energy-proportional and have a high idle power consumption. Most distributed file systems cannot fully utilize servers leading to low energy efficiency. ARM-based processors were designed for energy efficiency and low power consumption. Therefore, an ARM-based single-board computer storage cluster was built using the Odroid HC4. This cluster is evaluated using different distributed file systems, workloads, metrics, and a comparable x86-based setup as reference. Results show that ARM-based single-board computer storage clusters are an energy-efficient alternative to traditional x86-based clusters while providing comparable performance. However, based on the experiences with the Odroid HC4, future single-board computers for storage clusters should have a higher network throughput and should provide access to more storage devices.

Contents

1. Introduction **7**

2. Background **9**
2.1. Storage Systems . . . 9
2.2. ARM-based Single-Board Computers . . . 12
2.3. Benchmark Tools . . . 13

3. Survey of Distributed File Systems **15**
3.1. CephFS . . . 15
3.2. OrangeFS . . . 17
3.3. GlusterFS . . . 19
3.4. MooseFS . . . 21
3.5. BeeGFS . . . 23
3.6. JULEA . . . 24

4. Cluster Setup **27**
4.1. Hardware . . . 27
4.2. Software . . . 28
4.3. Theoretical Peak Performance . . . 29

5. Setup and Configuration of Distributed File Systems **32**
5.1. CephFS . . . 32
5.2. OrangeFS . . . 33
5.3. GlusterFS . . . 35
5.4. MooseFS . . . 36
5.5. BeeGFS . . . 38
5.6. JULEA . . . 38
5.7. Comparability of Configurations . . . 39

6. Performance and Energy Efficiency Analysis **40**
6.1. Workloads . . . 40
6.2. Data Collection and Discussion of Errors . . . 41
6.3. Performance and Energy Metrics . . . 42
6.4. Performance Comparison . . . 44
6.4.1. Parallel Sequential Data Access . . . 44
6.4.2. Metadata Access . . . 46
6.4.3. Source Code Compilation . . . 49
6.5. Energy Efficiency of the Parallel File Systems . . . 50
6.5.1. Parallel Sequential Data Access . . . 50
6.5.2. Metadata Access . . . 52
6.5.3. Source Code Compilation . . . 53

6.6. Design Recommendations for Future Single-Board Computer Storage Clusters 54

7. Related Work **56**

8. Conclusion and Future Work **58**

A. Benchmark Parameters **60**

B. Additional Measurements **66**

Bibliography **70**

Abbreviations

ADIO	Abstract I/O Device
AFR	Automatic File Replication
API	Application Programming Interface
ARM	Advanced RISC Machines
CPU	Central Processing Unit
CRUSH	Controlled Replication under Scalable Hashing
CSV	Comma Separated Values
DAOS	Distributed Asynchronous Object Storage
DFS	Distributed File System
DHT	Distributed Hash Table
EC	Erasure Coding
ECC	Error Correction Code
ECOS	Energy-Efficient Cluster Storage System
EDP	Energy-Delay Product
FS	File System
FUSE	Filesystem in Userspace
HDD	Hard Drive Disk
HDF	Hierarchical Data Format
HPC	High Performance Computing
I/O	Input/Output
ICMP	Internet Control Message Protocol
IOPS	I/O Operations per Second
LMDB	Lightning Memory-Mapped Database Manager
MDS	Metadata Service
MPI	Message Passing Interface
MPI-I/O	Message Passing Interface - Input/Output
NAS	Network Attached Storage
NetCDF	Network Common Data Format
NIC	Network Interface Card
NTP	Network Time Protocol
NUFA	Non Uniform File Access
NVMe	Non-Volatile Memory Express
OED	Open Ethernet Drive
OSD	Object Storage Devices

PG	Placement Group
POSIX	Portable Operating System Interface
PSU	Power Supply Unit
RADOS	Reliable Autonomic Distributed Object Store
RAID	Redundant Array of Inexpensive Disks
RAM	Random Access Memory
RBD	RADOS Block Device
RISC	Reduced Instruction Set Computer
SATA	Serial Advanced Technology Attachment
SBC	Single-Board Computer
SoC	System-on-Chip
SoM	System-on-Module
SSD	Solid State Disk
TSP	Trusted Storage Pool
USB	Universal Serial Bus
VFS	Virtual File System
VOL	Virtual Object Layer

Chapter 1.

Introduction

Most scientific domains are increasingly data-driven. Large-scale scientific applications create huge data sets by storing checkpoints or results, and more data than ever gets collected in large-scale experiments or observations. The consequence is that demands on capacity and data throughput of storage systems are increasing. These demands are satisfied by distributing storage over multiple servers in the network combining their capacity and performance. One possibility to manage data distribution and data access is to use *Distributed File Systems* (DFS). Different DFSs have emerged over the past decades, each designed with its own use cases and features. Though they differ in resource consumption, most of the DFSs cannot fully utilize the often over-provisioned, high-performance, x86-based servers currently used to build storage clusters. Such servers were found to have poor energy proportionality, meaning that high energy efficiency is only achieved for high utilization [9]. A consequence is that servers have a high energy consumption even when mostly idle. This leads to an increasing waste of energy when scaling-up systems.

The high idle power consumption per node is offset by adding large amounts of storage devices per server to achieve a high capacity. For new and fast storage technologies like *Non-Volatile Memory express* (NVMe) *Solid-State Drives* (SSD), this is no more possible without sacrificing the bandwidth of the storage devices. For example, a server with a 40 Gbit/s network connection could saturate about 38 *Hard Disk Drives* (HDD) but only one to two NVMe SSDs when assuming 125 MiB/s per HDD and 2500 MiB/s per SSD and no RAID technology. Therefore, scaling up SSD-based storage clusters is even more expensive than HDD-based storage clusters, not only because of the higher hardware costs but also because of the energy consumption.

For large *High-Performance Computing* (HPC) deployments, the energy consumption of the compute nodes is already a limiting factor [72]. One approach to improve energy efficiency is to use *Advanced Reduced Instruction Set Computer Machines* (ARM) processors instead of x86 processors. This *Central Processing Unit* (CPU) architecture was initially designed for usage in mobile or embedded devices and is henceforth designed for energy efficiency. Early proof-of-concept deployments like Tibidabo at Barcelona Supercomputing Center [62] validated the feasibility of this idea. The HPC machine Fugaku at the Riken Center for Computational Science [68] proved that ARM-based servers provide sufficient compute power for HPC applications and make even larger clusters possible due to their energy efficiency.

Like for compute clusters, scaling-up storage clusters is also more and more limited by restrictions on energy consumption. This motivates the use of ARM-based machines for storage servers as well. However, high-performance compute nodes like those used in Fugaku are certainly not necessary for a storage cluster. Apart from using ARM-based processors on fully-featured servers, micro-servers [28] and single-board computers [3, 58, 61] were evaluated for

their energy efficiency. This approach is also interesting for storage clusters as it additionally enables administrators to add capacity when needed easily. Therefore, the goal of this thesis is to answer the following question: Can ARM-based single-board computers be used to build storage clusters that achieve a better energy efficiency than typical x86-based storage clusters while providing comparable performance?

To answer this question, a prototype ARM-based single-board computer cluster was built. The cluster uses six Odroid HC4 boards, which were designed as home *Network Attached Storage* (NAS) servers. The behavior of storage systems depends not only on the used hardware and the deployed DFS but also on the storage access patterns, in other words, the storage workload. Consequently, different benchmarks were run on different DFSs. The benchmarks represent workloads that are typical for HPC storage systems. The first set of benchmarks comprised parallel, coordinated access to a single or a few files from multiple clients. That workload could be created by scientific applications writing their results or reading checkpoints. The second set of benchmarks measured the *Input/Output* (I/O) operations that could be achieved per second while creating and accessing many small files. Such a workload could be created by applications that synchronize using the distributed file system. The third benchmark was a real-world benchmark that measured the performance of compiling source code on the DFS. This is a workload typically seen in the home directory of an HPC cluster. Though the home directory does not necessarily need to reside on a typical HPC file system, it was interesting to see how the systems behaved. The file systems chosen were CephFS, OrangeFS, MooseFS, GlusterFS, BeeGFS, and JULEA. They represent a selection of different design approaches that can be used to show the strengths and weaknesses of the ARM-based cluster. As a reference, the benchmarks were also run on a comparable x86-based reference setup. The results were analyzed using different performance and energy-efficiency metrics.

The contributions of this thesis are:

1. It is shown that ARM-based single-board computer storage clusters are a viable alternative to x86-based storage clusters.
2. Different design recommendations for future single-board computer storage clusters are made based on the gained experiences.

The rest of this thesis is organized as follows. In Chapter 2 background information and definitions of used technologies like file systems are given, followed by a short discussion of ARM-based single-board computers and the used benchmark tools. Chapter 3 surveys the six distributed file systems that are subject to practical analysis in this thesis. The survey will provide a solid base for later discussions. Chapter 4 describes the hardware and software setup of the ARM-based and x86-based clusters and models their theoretical peak performance for sequential and parallel file access as well as metadata performance. In Chapter 5 the setup and configuration of the distributed file systems on the ARM-based cluster is described. The configurations are validated using specific benchmarks based on the architecture and features of the respective systems. At the end of this chapter, the comparability of the resulting setups is shortly discussed. Chapter 6 starts with a description of the benchmarks done on the systems and a discussion of measurement errors. This is followed by a discussion of different performance and energy efficiency metrics. After the results are presented and analyzed, conclusions for the design of future energy-efficient ARM-based storage clusters are drawn. Related works are presented in Chapter 7. Finally, in Chapter 8 the thesis is summarized, and some interesting points for future work are identified.

Chapter 2.

Background

This chapter gives an overview of storage solutions and interfaces. After discussing file systems and their interfaces, ARM-based single-board computers are briefly described. The chapter ends with short descriptions of the benchmark tools used for this thesis.

2.1. Storage Systems

File Systems The most common way to organize permanent storage is to use *file systems*. File systems provide a hierarchical namespace using *directories* and *files*. Files are "named objects that exist from their explicit creation until their explicit destruction and are immune to temporary failures in the system" [46]. Directories are used to group files and, therefore, to create the hierarchical namespace. Local file systems like EXT4, XFS, and NTFS manage data stored on a single storage device. This storage device can be virtual, for example, to use underlying storage layers like *Redundant Array of Inexpensive/Independent Disks* (RAID) [60]. RAID combines several disks into a single device. Multiple levels exist which provide different data protection features. Some examples are:

- RAID 0 offers no protection but combines the disks' performance by distributing data blocks between them.
- RAID 1 mirrors data between the disks to protect against failures.
- RAID 5 uses parity blocks to protect against one disk failure among more than three disks.

Some modern file systems, like ZFS or BTRFS, include functionalities of underlying storage layers and can manage data on multiple disks. However, the scaling of those file systems is limited by the number of disks that can be attached to a single machine.

In contrast, *Distributed File Systems* (DFS) are file systems that manage data distribution over multiple storage servers and provide the data under a unified and hierarchical namespace [82]. Most DFSs split file metadata from file data and store it on dedicated servers along with the information on where to find the file's data. Another popular way to place and find files is to use various hashing techniques. The file data can be distributed using various schemes. Clients can look up the file data location to directly communicate with the data servers. A DFS is said to be *parallel* if "data blocks are striped, in parallel, across multiple storage devices on multiple storage servers" [82]. Clients can access DFSs via several interfaces.

POSIX Interface The *Portable Operating System Interface* (POSIX) standard [74] specifies, among other things, a set of functions to access file systems as well as the semantics of each function. The big advantage of POSIX is that applications written against the defined interface are portable between different POSIX-compliant file systems. Therefore, applications can be developed and tested on local file systems and then deployed on a cluster using a distributed file system. Ideally, the used file system is transparent to applications using POSIX I/O. In reality, however, the performance of a file system depends on the access patterns of applications. On distributed file systems, the relation between performance and access patterns is further complicated by multiple users that compete for disk access and interactions between different storage layers. In addition, accessing a file system through the POSIX interface does not guarantee POSIX semantics. Most file systems only comply with the standard to a certain degree because the strict semantics can complicate implementation and impair performance. For example, the POSIX standard states that written data should be returned by any following read call from every process. For local file systems, that property can easily be satisfied by the *Virtual File System* (VFS), a component of the Linux kernel that provides an abstraction layer for multiple file systems in use. Nevertheless, for distributed file systems, this imposes a strict constraint on client-side caching and throttles the system's performance. Some systems, like OrangeFS [11], therefore do not implement this behavior to boost performance. After all, this strict behavior is unnecessary for many applications, especially scientific ones [85].

Under Linux-based operating systems, three different architectures can be used to implement this access. The first two interact with the VFS. Using the VFS, a unified namespace is presented to the user. File systems can be mounted in arbitrary directories, making their root available under the path of that directory. The traditional approach to implementing file systems is to build them as kernel modules. A *kernel module* is a dynamically loadable object that runs in kernel space. File systems not included in the Linux kernel can be mounted if they provide a kernel module that implements the necessary callbacks.

Instead of kernel modules that run in kernel space and therefore complicate the development and allow for errors that result in system crashes, the *File System in User Space* (FUSE) library can be used to implement the DFS clients. FUSE file systems run entirely in user space. This architecture is possible via the FUSE kernel module that interacts with the VFS and exposes an interface to user space applications. Debugging a FUSE file system is more manageable than debugging a kernel module because deadlocks will not freeze the entire system, and bugs can not cause kernel panics. Normal debugging tools such as *gdb* or *valgrind* can be used. Besides easing the development of file systems, another benefit is that unprivileged users can mount FUSE file systems. Nevertheless, FUSE introduces another software layer that may impair the file system's performance.

Most distributed file systems offer libraries to bypass interactions with the kernel. That can be done on several abstraction levels. The most straightforward way from the users' perspective is to pre-load a DFS-specific library that implements the I/O functions of the standard C library *libc*. Because *libc* is dynamically linked by default, like most libraries in modern programs, this will work for most applications. In doing so, the usual I/O calls are overridden by file system specific functions. Therefore, existing programs do not need to be recompiled. Another possibility is to write programs using I/O functions provided by the respective DFS. However, such a library will require changes to existing code and insight into the respective file system's specific details.

The three ways to implement POSIX access come with their advantages and disadvantages. As

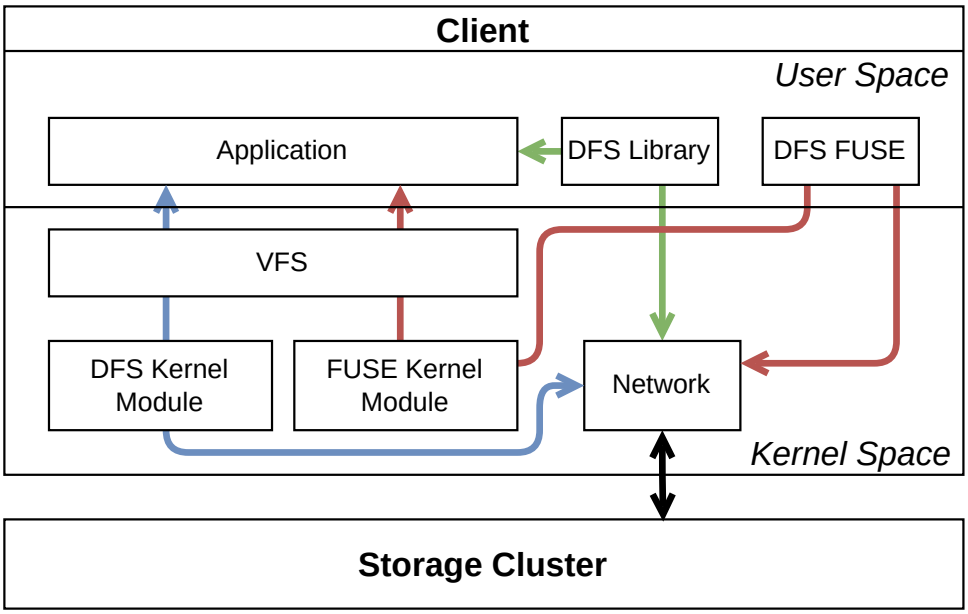
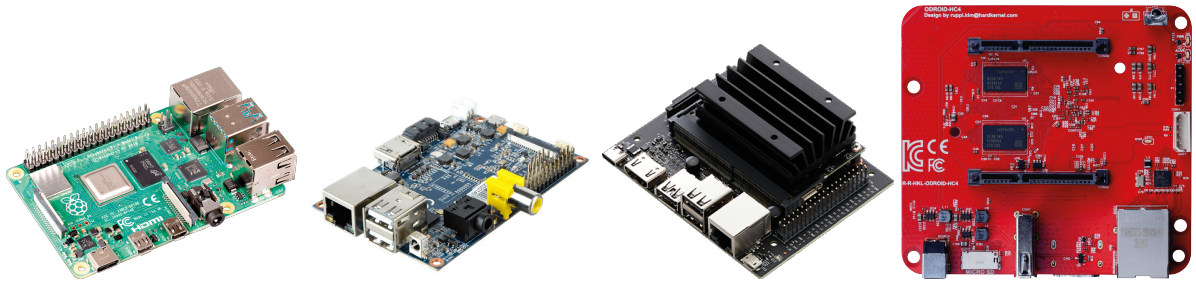


Figure 2.1.: There are multiple ways to implement POSIX access to a DFS. The blue (kernel module), red (FUSE), and green (DFS library) arrows follow the data path for each method. In the end, data is transferred via the network.

always, minimizing the software layers is likely to improve performance and reduce latency. In addition, each context switch between kernel and user space introduces overhead to I/O. Figure 2.1 illustrates the data path for each implementation. The most overhead is introduced by FUSE, which has two more context switches than access via the DFS library or the kernel module and passes data through most software layers. The DFS library realizes the most efficient access. However, using the library, some useful caching features of the VFS can not be used, and client-side caching needs to be implemented in the library.

MPI-I/O Interface Apart from the POSIX interface, several other interfaces can be used to access files in a DFS. One example is the I/O interface of the *Message Passing Interface* (MPI) [47]. MPI is a standard that was created to unify communication between the various processes in distributed-memory applications. Conceptually, this is done by sending and receiving messages via the network. Version 2 of MPI introduced I/O functionality via the MPI-I/O interface. It is designed for highly parallel access to a single shared file. MPI-I/O shares many concepts with MPI as storing data is treated like sending and loading data like receiving messages. In contrast to POSIX, MPI-I/O does not work on byte streams but on streams of typed elements. In addition to built-in data types like `float` or `int`, complex types can be defined by users. MPI-I/O also defines the semantic of I/O operations which is more relaxed than the POSIX semantic. Conflicting accesses, e.g., parallel writing to and reading from overlapping file parts, are not defined by the standard, and the result of such operations is undefined. These relaxed semantics are sufficient for most scientific applications [85]. In theory, this enables lock-free I/O in a parallel environment. In practice, however, few systems provide a native MPI-I/O interface. If no native interface is available, POSIX is used to implement MPI-I/O, adding yet another layer to the storage stack. Depending on the I/O semantics of the DFS, most performance advantages of MPI-I/O are lost in this case.



(a) Raspberry Pi 4B[1] (b) BananaPi M1[2] (c) Nvidia Jetson Nano[3] (d) Odroid HC4[4]

Figure 2.2.: Some popular examples of single-board computers.

High-Level Interfaces Both POSIX and MPI-I/O are rather complicated to use in practice, especially for scientists of various domains developing simulations. To abstract from details of the low-level interfaces, several high-level I/O libraries are built on top of the low-level ones. Some of them, like HDF5 [37] or NetCDF [63], are specifically designed for scientific applications. They offer a self-describing data model and file specification tuned to store scientific data sets. Though they were not built for parallel access from the start, this functionality was later added via MPI-I/O. Although they add another software layer to the storage stack, they provide valuable features for data sharing and archiving due to their self-describing nature.

Other storage systems In addition to file systems, other storage abstractions with different semantics and data models exist. Though the focus of this thesis is on distributed file systems, these technologies are used as building blocks for DFSs and should therefore be mentioned as well.

Databases are a traditional approach to store structured data [21]. While relational databases, like SQLite or MariaDB, store data in the form of tables and offer access via the query language SQL, there are also non-SQL databases like MongoDB [21]. Due to their structured data model, databases are used by some DFS to store metadata.

Other popular modern storage interfaces are *Object and Key-Value Stores* [21]. Unlike file systems, they often provide multiple namespaces that contain mappings from keys to the respective data objects. Whereas object stores are intended to store larger objects, key-value stores should be used to store small objects, similar to databases. An example is the *Distributed Asynchronous Object Storage* (DAOS) [15] which is developed by Intel and is optimized for non-volatile memory and fast NVMe SSDs. Some of the DFSs analyzed in this work do also offer access to their data as an object store, e.g., OrangeFS.

2.2. ARM-based Single-Board Computers

Single-Board Computers (SBC) were designed as development boards, embedded controllers, and a platform for education [59]. Most SBCs run on a *System-on-Chip* (SoC) device. SoC

[1] `https://cdn-reichelt.de/bilder/web/xxl_ws/A300/RASP_PI_4_B_8GB_01.png` retrieved at 03.05.2022
[2] `https://wiki.banana-pi.org/images/4/45/Banana_pi_BPI-M1_1.jpg` retrieved at 03.05.2022
[3] `https://www.nvidia.com/content/dam/en-zz/Solutions/gtcf20/jetson-nano-education-projects/jetson-nano-enthusiast-2gb-dev-kit-2c50-d.jpg` retrieved at 03.05.2022
[4] `https://cdn-reichelt.de/bilder/web/xxl_ws/A300/ODROID_HC4_04.png` retrieved at 03.05.2022

devices integrate most or all computer components into a single chip. Typically the SoC uses the ARM CPU architecture, though there are some exceptions like the UDOO Bolt v8, which is powered by an x86 CPU [75]. ARM-based processors were developed for mobile and embedded use cases and are henceforth designed with power efficiency in mind [27]. Famous examples of SBCs are the Raspberry Pi, the BananaPi, Nvidia Jetson devices, and Odroid devices. These are shown in Figure 2.2. Each board offers a different set of external connectors. SBCs that feature storage connectors like SATA are of particular interest for this work. One example is the Odroid HC4 [30].

Recently, ARM-based systems gained attention in HPC contexts. There are multiple examples for evaluations of SBC clusters [3, 57, 58]. Apart from single-board computers, ARM CPUs are also used to build typical servers. HPC machines like Tibidabo at Barcelona Super Computing Center [62] and Fugaku at the Riken Center for Computational Science in Kobe [68] prove that ARM-based servers are a viable and more energy-efficient alternative to the traditional x86 architecture.

2.3. Benchmark Tools

In this section, the benchmark tools that will later be used for the evaluation are shortly described.

Network The benchmark tool *iperf* [20] is used to measure network throughput. Because it is limited to a single thread, care must be taken when measuring high-performance networks starting at 40 Gbit. However, on 1 Gbit networks *iperf* can saturate the connection and measure the maximum achievable throughput.

The *ping* [1] utility was developed to measure the round-trip time of network connections, which is defined as the period between sending a message and receiving a response. *Ping* uses the echo function of the *Internet Control Message Protocol* (ICMP) [1]. The flood mode and pipeline options help measure the maximum I/O operations per second that can be transferred via the network.

Local I/O *Fio* is a versatile I/O benchmark tool. It was designed by Jens Axboe to enable flexible testing of the Linux I/O subsystem and schedulers and features lots of parameters [6].

Complex, multi-threaded workloads can be defined in configuration files. In addition, it is possible to replay recorded I/O traces of real-world applications so that they do not need to run on the target system.

In this work, only a tiny subset of *fio*'s features are used to measure the maximum data throughput and IOPS that can be achieved on the disks. Because *fio* can not coordinate file access from multiple clients in the network, it can not be used to benchmark the I/O behavior of distributed memory applications that are executed on multiple machines.

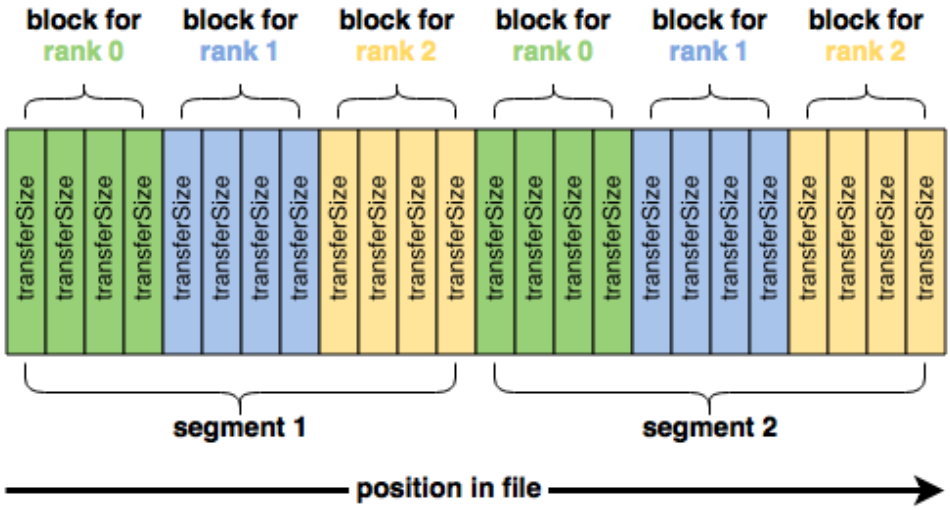


Figure 2.3.: Each participating *IOR* process reads or writes one block per segment. The transfer size specifies the size of a single I/O call. The image is taken from the *IOR* documentation [78].

Parallel I/O *IOR* [73, 78] is a parallel benchmark tool designed to measure throughput using workloads typical for scientific applications. Each client, coordinated via MPI, writes and reads non-overlapping blocks of data to and from multiple files per process or to a single shared file. Many parameters, like the transfer size, can be configured. Most importantly, the storage interface backend can be chosen among POSIX, MPI-IO, HDF5, and others.

Figure 2.3 shows how access to one or multiple files is coordinated. Each participating client process writes and reads one block of data per segment using access sizes specified by the transfer size. The blocks can be shifted after the write-phase so that each client accesses blocks written by another client. Therefore, reads can not be served out of the clients' caches. The influence of client-side write-behind caching can be reduced by allocating fixed percentages of the available main memory.

The *mdtest* benchmark tool is managed in the same code repository as *IOR*. In contrast to *IOR*, *mdtest*'s focus is to measure the metadata performance of workloads with many small files. For this purpose, a directory tree containing files and directories is created by multiple clients. All clients perform operations such as creates, writes, reads, and `stat`-calls on a specified number of files and directories per directory of the tree. The `stat` system call returns the status of a file, including the file owner, permissions, and the access time [2]. Additional time stamps are needed to synchronize the phases of *mdtest* with the power measurement. Therefore, the *mdtest* version used in this thesis was modified.[5]

Both tools are common for measuring the parallel performance of distributed file systems. For example, the benchmarks done for the IO500 list [42] are performed using these tools.

[5]The modified source code is available at `https://github.com/t-erxleben/ior/tree/3.3`.

Chapter 3.

Survey of Distributed File Systems

This chapter contains a survey of the currently popular distributed file systems CephFS, OrangeFS, GlusterFS, MooseFS, BeeGFS, and JULEA. Each file system is described by its architecture, interfaces, metadata handling, data distribution, synchronization and caching behavior, and data resiliency.

A lot of distributed file systems have emerged over the past decades. Some examples other than the ones surveyed in this chapter are Lustre[1], SeaweedFS[2], LizardFS[3], and Hadoop Distributed File System[4]. Each system was designed for specific use cases resulting in different behavior of each system. The systems for evaluation in this thesis were chosen due to their recent popularity and because they represent different design choices. GlusterFS and CephFS use hash-based methods to place and locate files. In contrast, JULEA, MooseFS, BeeGFS, and OrangeFS separate metadata for client lookups. The way metadata management and storage are handled is unique for each system. Evaluating these different approaches gives a good overview of the ARM-based cluster's usability as a storage cluster.

3.1. CephFS

Ceph [86] is a free and open-source distributed storage cluster licensed under LGPLv2.1. The development of Ceph is nowadays managed by RedHat, which also offers commercial support and storage solutions built with Ceph. CephFS is a parallel distributed file system built on top of Ceph. The following paragraphs are based on the Ceph documentation [12].

Architecture CephFS is based on Ceph's *Reliable Autonomic Distributed Object Store* (RADOS). A Ceph Storage Cluster comprises Ceph Monitors, Ceph *Object Storage Device* (OSD) services, and Ceph Managers. The monitor service manages the cluster map based on heartbeat reports from OSDs. Each OSD manages one storage device using Ceph's Bluestore [4], a thin object-store operating on raw block devices. Therefore, Ceph does not rely on external local file systems. Manager services provide additional monitoring and interfaces for external monitoring tools and an interface for extension modules. Optional services in a Ceph Cluster are *Metadata Servers* (MDS), handling metadata management for CephFS, and Ceph Object Gateway services, providing a RESTful[5] interface for RADOS.

[1] `https://www.lustre.org/`
[2] `https://github.com/chrislusf/seaweedfs`
[3] `https://lizardfs.com/`
[4] `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`
[5] Representational State Transfer (REST) is an architecture style for distributed hypermedia systems [23].

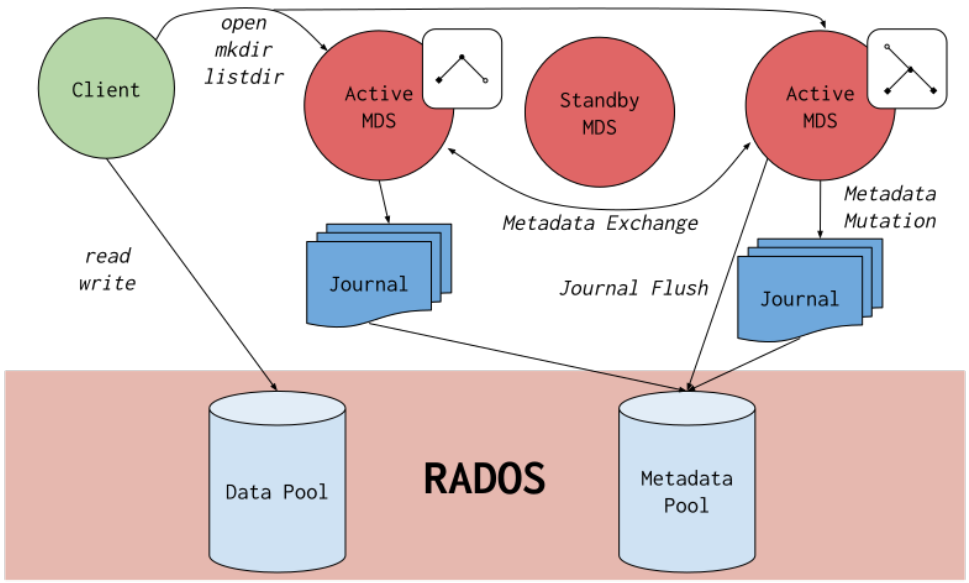


Figure 3.1.: Architecture of CephFS which uses RADOS to implement a POSIX file system [12].

Interfaces As mentioned above, several interfaces can be used to interact with the storage cluster. The same cluster might be used to operate the RESTful Ceph RADOS Gateway, the Ceph File System or Ceph Block Devices. It is also possible to interact directly with the cluster using the library *librados*. Access to CephFS is possible via a FUSE file system or a kernel module.

Metadata Handling Each MDS maintains a metadata journal in a separate metadata RADOS pool. Because the MDS stores no metadata, the journal can be used to recover bad file system states in case of failure. Clients must send requests to MDSs for all metadata-related operations. However, an MDS may grant certain capabilities to clients so that they can manipulate inodes themselves or serve metadata requests out of their cache. In this case, clients operate directly on the inode stored in their cache or the metadata RADOS pool. If multiple MDSs are configured to be active, they will distribute the responsibility for inodes among them so that exactly one MDS manages each inode. The handling of metadata is summarized in Figure 3.1. Nevertheless, clients do not need to know which MDS to contact for access to specific files because requests will be forwarded if necessary.

Data Distribution Files in CephFS are split into stripes which in term are stored as RADOS objects. Each object in a RADOS data pool is assigned to a *Placement Group* (PG) according to the *Controlled Replication Under Scalable Hashing* (CRUSH) algorithm. PGs are used to manage the replication of objects without managing each object on its own. The inputs for the CRUSH algorithm are the CRUSH map and the object key. The CRUSH map includes the clusters structure and placement rules per pool. The administrator can create rules to define failure domains and set storage classes for pools. For example, a rule might be created to distribute data over several server racks using only SSDs. PGs are stored on multiple OSDs according to the replication goal of the pool. They are also used to automatically balance storage nodes as they can be moved between different OSDs.

Synchronization and Caching CephFS is mostly POSIX compliant. To guarantee strict synchronization requirements, the MDSs manage client caches using capabilities. A capability is the permission to cache certain metadata or data objects and defines what operations may be performed on the cache. Both clients and MDSs try to enforce cache size limits. To avoid conflicting caches, an MDS might revoke capabilities.

Replication and Fault Tolerance Failure of all components is treated as the norm instead of an exception in Ceph. If a cluster node is not reachable, the CRUSH map gets dynamically updated, and the PGs that got lost will be restored from their replicas. Though the client performance can drop during rebalancing, all the data is still accessible. In addition to replicated pools, it is also possible to create erasure-coded [7] pools for a better trade-off between capacity and resilience.

Quorum among the monitors is determined by Paxos algorithm [43] to prevent split-brain scenarios.

Monitors, MDSs, and Managers can be configured with automatic fail-overs, so a spare service will take over even when they crash.

3.2. OrangeFS

OrangeFS [11], formerly *Parallel Virtual File System 2* (PVFS2), is a traditional free and open-source HPC parallel distributed file system initially developed for research purposes at Clemson University and the Argonne National Laboratory. It is distributed under the LGPLv2.1. The first production version of OrangeFS was released in 2010. Commercial support is available by Omnibond. The following paragraphs are based on [48] if not referenced otherwise.

Architecture OrangeFS uses just one service type that handles data storage and metadata management. Even though it is possible to separate data and metadata on multiple servers as each service can be configured to handle a specific range of data and metadata object IDs [79]. The server interacts with the local file system of the server through the *Trove* interface. Request from a client may be sent to an arbitrary server which will communicate with other nodes in the cluster to fulfill the request.

Interfaces Notable are the many interfaces that may be used to access OrangeFS. Besides classical POSIX access via FUSE or a kernel module, WebDAV, S3, Hadoop, or a RESTful web interface is possible. Because of the implementation of the ADIO layer of ROMIO, MPI-I/O is well supported. All interfaces are summarized in Figure 3.2.

Metadata Handling Metadata is stored as key-value pairs either in Berkley DB or in *Lightning Memory-Mapped Database Manager* (LMDB) [79]. A metadata object consists of traditional file metadata, a list of the associated data objects, distribution, layout information, and user-defined attributes. Typically metadata is distributed over several servers, using extensible hashing to achieve load balancing for many parallel accesses.

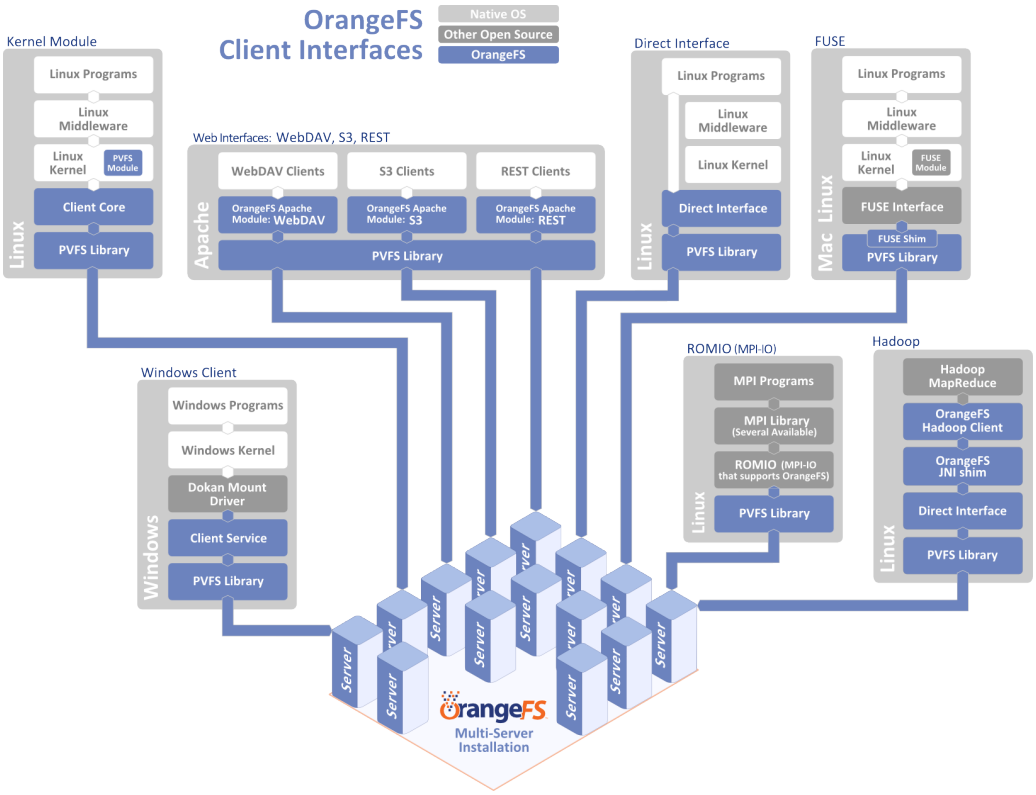


Figure 3.2.: An OrangeFS cluster provides various interfaces to access the data [79].

Directories are implemented as files containing metadata objects handles as content. The contents of a directory can be distributed as well. In this case, hashing is used to decide which entry gets stored to which data object.

Data Distribution Per default, file contents are striped in OrangeFS. The resulting stripes can be distributed using various schemes that may be configured per directory or file. The default is to start at a random server and distribute stripes in a round-robin fashion across all servers. Other schemes include distributing whole files, a 2D grouping of servers over which stripes are distributed and round-robin distribution with stripe sizes configured per server [41].

Synchronization and Caching Despite access via POSIX-defined interfaces, OrangeFS is not POSIX compliant. Similar to MPI-I/O semantics, overlapping accesses are not synchronized, leading to undefined behavior [85]. There is no support for file locks as the server is stateless.

Though unfamiliar at first, those relaxations of semantics enable an easy and performant implementation tuned towards throughput in workflows typical for scientific applications.

OrangeFS use multiple caching strategies. On the server-side, the *Trove* layer handles access to local storage devices. Those accesses can be configured to be synchronous or cached for data and metadata. Per default, data is cached, and metadata is written synchronously.

On the client-side, the Linux page cache is used [80].

Replication and Fault Tolerance As there is no central server, OrangeFS, in theory, has no single point of failure. However, no replication is available yet, though planned for the next major release OrangeFS v3 [22]. The first step towards resilience features is that files can be replicated after configuring settings for copies and fail-over. When they get marked as immutable, their metadata and data will be copied to other servers. As for now, to prevent data loss in general, software or hardware RAID needs to be used on the nodes. In case of a crashing or unreachable node, data will get unavailable.

3.3. GlusterFS

Gluster is a free and open-source distributed file system licensed under GPLv2 and LGPLV3+. The development of Gluster is nowadays managed by RedHat, which also offers commercial support and storage solutions built with Gluster. The following paragraphs are based on the GlusterFS documentation [77].

Architecture Like OrangeFS, GlusterFS has only one type of server. All GlusterFS severs form a *Trusted Storage Pool* (TSP), which provides attached storage, called *bricks*, for volumes. Each volume provides its own namespace that clients can mount. Multiple volumes of different types may be created on top of a TSP.

GlusterFS is designed using a pipeline scheme. *Translators*, the pipeline layers, are stackable, dynamically loaded objects that are used on the client-side and the server-side to process I/O requests and implement most features of GlusterFS. They can modify the destination, flags, and data of I/O requests or even spawn additional requests. Important translators are the *Distributed Hash Table* (DHT), *Automatic File Replication* (AFR), and *Erasure Coding* (EC) cluster translators. As suggested by their names, they implement the core features of file distribution and data safety. Using the DHT translator, each client can place and find files using a hash of the file's name. AFR and EC translators implement the two data safety strategies of GlusterFS. Using different caching techniques like read-ahead, write-behind, or metadata caching, other translators may improve performance. For converged clusters, i.e., computation and storage are performed on the same servers, the *Non Uniform File Access* (NUFA) translator can be used to place files on local bricks of clients if possible. Translators on the server-side are, among other things, used to access local file systems, enforce quotas, and implement file locking. They may also be used to implement I/O tracing or debugging for both clients and servers. Figure 3.3 shows an example of a GlusterFS setup with both translator stacks and summarizes the architecture.

Interfaces GlusterFS provides POSIX access via FUSE. Apart from that, there is an NFS Ganesha[6] integration that enables native NFS and pNFS access.

Metadata Handling In GlusterFS, no separate metadata handling is needed because all participants can determine file positions. All POSIX file attributes are stored within the inode of the files in the local files systems of the servers. In cases where files are striped or replicated, metadata will be replicated and stored in each inode.

[6] `https://github.com/nfs-ganesha/nfs-ganesha`

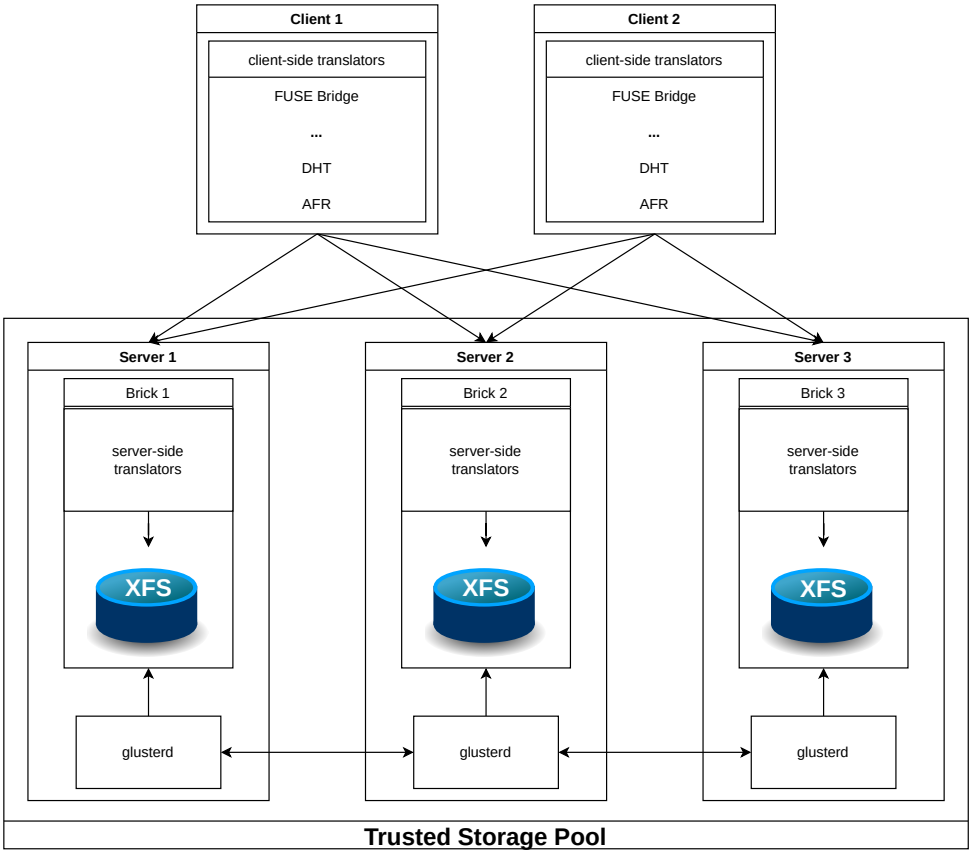


Figure 3.3.: An example of a GlusterFS setup with two clients and three servers. This graphic is based on [77].

Directories are always created on all bricks of a volume.

Data Distribution Data distribution is determined by the type of GlusterFS volume in use. Volume types are defined by default sets of translators. There are three basic types: distributed, replicated, and dispersed volumes.

Distributed volumes pseudo-randomly map whole files to different participating storage servers based on file name's hashes.

Replicated volumes store files to every participating server and, therefore, ensure high availability of data.

Dispersed volumes split files in stripes and use erasure coding to add redundant stripes of data. The redundancy count, i.e., the number of redundant blocks, can be specified when the volume gets created. In dispersed volumes, data can be restored when several servers up to the redundancy count crash. This approach is more space-efficient than replicated volumes. A replicated volume that enables one server to fail will double the amount of data. In contrast, a dispersed volume only needs to increase storage consumption by one-third for three bricks. The downside is that the stripe size depends on the number of bricks and the redundancy count as described by Equation (3.1).

$$\text{stripe size} = 512 \cdot (\#\text{storage servers} - \text{redundancy}) \tag{3.1}$$

Both replicated and dispersed volumes can be combined with the concepts of a distributed volume. Therefore, files can be distributed between dispersed or replicated groups.

Synchronization and Caching GlusterFS is near POSIX compliant. Writes are performed as transactions using locks implemented on the server-side to implement the strict semantics.

Several translators enable multiple caching strategies. On the client-side, both data and metadata are cached. For data, read-ahead and write-behind caches are implemented. In addition, the Linux page cache is used. For metadata, the `struct stat` and extended attributes are cached. Caches will be invalidated after a timeout passes.

Replication and Fault Tolerance As described above, replicated and dispersed volumes implement data safety features for GlusterFS. Both volumes allow for a certain number of bricks to fail simultaneously. When the failed bricks are available again, self-healing is triggered to synchronize data. The pro-active self-healing service will also repair data in case of interrupted transactions.

However, automatic self-healing can not resolve split-brain scenarios for replicated volumes. A file is in a split-brain state if all data replicas are marked as dirty and no source of the correct version can be determined. Different approaches are possible to prevent or resolve split-brain conflicts. The correct version can be determined by quorum among the bricks for three replicas. Arbiter volumes can be used to reduce the amount of consumed storage space. Some bricks, called *arbiter bricks*, will not store the data but only information on data versions. Therefore, a quorum can be determined without storing three replicas. As a last resort, split-brain conflicts can be manually resolved using a tool provided by GlusterFS.

3.4. MooseFS

MooseFS is a parallel distributed file system that is written in C and is available under the GPL 2.0 license. In addition, a pro-version is sold that adds fail-over mechanisms for the metadata server, data redundancy via erasure coding, and a native client for Microsoft Windows. The following paragraphs are based on the MooseFS documentation [39] if not referenced otherwise.

Architecture A MooseFS cluster consists of a master server, multiple metalogger servers, multiple chunk servers, and optional monitoring servers.

The master server manages metadata, synchronizes concurrent write accesses to single data chunks, and maintains the clients' caches. The master server is single-threaded and periodically checks all chunks. Multiple actions can be performed for encountered chunks: Corrupted chunks will be restored from copies if possible. Chunks may be moved to balance the data distribution of the cluster. Chunks that are marked for deletion will be freed. Those actions are performed within the limits specified in the master server's configuration. Also, the number of chunks checked per second can be configured.

Metaloggers periodically backup the metadata from the master server. In case of failure, they can be configured to replace the master.

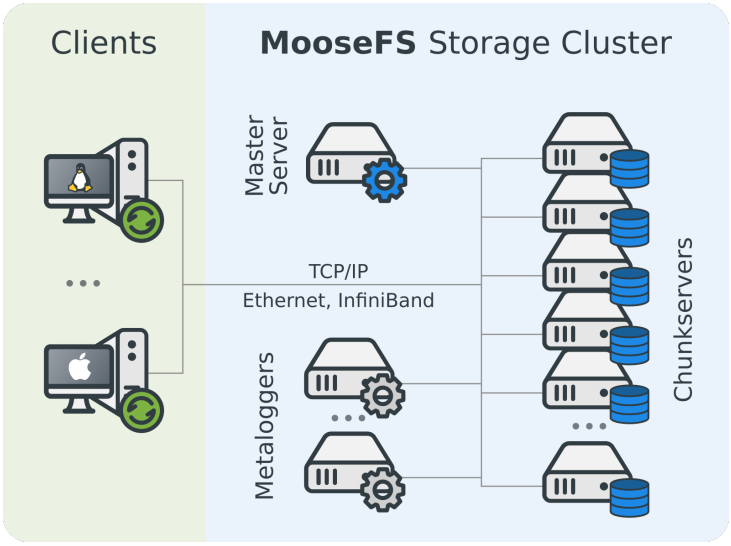


Figure 3.4.: An example of a MooseFS cluster with multiple chunkservers, two metaloggers, and two clients [76].

Chunk servers store the actual data blocks called *chunks* in MooseFS.

Figure 3.4 summarizes the architecture.

Interfaces MooseFS offers POSIX access via a multi-threaded FUSE client.

Metadata Handling Metadata is managed by the master server and is entirely held in memory. According to the documentation [39], one metadata entry consumes 300 – 350 B. Therefore, approximately 3 million files can be stored per Gibibyte main memory. All metadata is regularly stored on a disk for persistence and safety as a single file. Per default, this will happen once every hour. Multiple copies of that metadata file can be kept. As an additional safety measure, changes to metadata are written to an on-disk journal before modifying metadata in memory. In case of failure, the metadata file gets loaded, and the operations stored in the journal are executed.

Data Distribution Data in MooseFS is always striped with a fix stripe size of 64 MiB. Those stripes are called *chunks* in MooseFS. Chunks will be randomly distributed while prioritizing targets with a lower load [8].

If the file has a replication goal bigger than one, the copies are written to a chain of chunk servers. Consider the following example of a client A and three chunk servers B, C, and D: A writes to a file with a replication goal of three. B receives one chunk of that file and writes a copy of that chunk to C. C, in terms, will then write another copy to D.

Synchronization and Caching MooseFS is POSIX compliant and supports distributed locks. For reads and writes, no synchronization is needed if operations are performed on different chunks. The master server synchronizes conflicting accesses to the same chunk. Different client cache modes are possible. The default and recommended way is to use automatic caching managed by the master server. The other options are to always or never cache data.

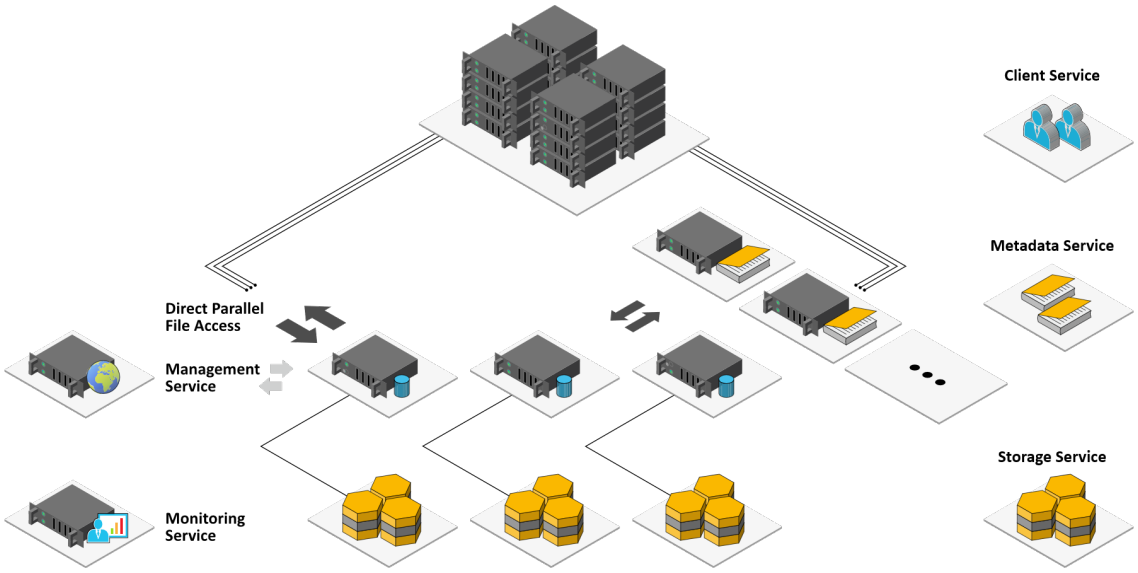


Figure 3.5.: An example of a BeeGFS cluster with multiple storage and metadata services. The cluster is connected to multiple clients that form a HPC cluster [84].

Replication and Fault Tolerance Users can set replication goals per file or directory using the utility tools shipped with MooseFS. This setting can be constrained by upper and lower bounds. Whether replication is done between servers or only between disks can be configured. Chunks are versioned so that no conflicts occur when chunks are modified while one server holding a copy of that chunk gets unreachable for a short amount of time. In addition, chunks are verified by checksums. If a chunk is replicated and a check fails on a copy, it will be restored automatically. For metadata, no checksums are computed. Because errors can be expected to occur in memory, the use of *Error Correction Code* (ECC) memory is strongly recommended to prevent metadata corruption.

3.5. BeeGFS

BeeGFS, formerly Fraunhofer General File System, is a popular parallel distributed file system for HPC systems. Since 2014 ThinkParQ has offered professional services for BeeGFS and an enterprise version. The enterprise version includes mirroring for data resiliency, quota enforcement, access control lists, and multiple storage pools. Even though the code is open-source, it is distributed under a proprietary license. Modifications of the source code may only be done for internal use, and users must be informed about all modifications. The following paragraphs are based on the BeeGFS documentation [84] if not stated otherwise.

Architecture A BeeGFS cluster consists of storage, metadata, management, and optional monitoring services. The storage services are responsible for storing the striped file data. Each storage service manages a set of storage targets, i.e., local storage devices of that node. Metadata services manage the file system's namespace and metadata for each file. Management services store settings for the cluster and monitor the health of other services. The optional monitoring service provides statistics using InfluxDB and Grafana. An overview of the components of a BeeGFS cluster is given in Figure 3.5.

Interfaces BeeGFS offers POSIX access via a Linux kernel module.

Metadata Handling Metadata is distributed among metadata services per directory. The metadata service that stores the root directory gets elected during the first startup of the cluster, and the management service stores its ID for later access. One metadata file is created on the respective metadata node for each logical file. The metadata is then stored to the extended attributes of that file, while the actual file data is left empty. In addition to POSIX file metadata, BeeGFS metadata, e.g., the stripe pattern, is stored.

Data Distribution Each file in BeeGFS is part of a storage pool created from a set of storage targets. The default pool consists of all available targets. The stripe pattern can be set per file or directory and is initially inherited by the parent directory. It consists of the number of storage targets and the stripe size. The choice of a storage target from the storage pool of the file depends on the capacity load of the targets so that the load will eventually balance. This load balancing is implemented via three capacity pools, *normal*, *low*, and *emergency*, in which the targets are categorized. Targets of a lower capacity pool are prioritized.

If mirroring is enabled, stripes will be distributed over buddy groups that consist of storage targets mirroring one another.

Synchronization and Caching BeeGFS supports client-side caching either via the Linux page cache, called *native* mode, or internal buffers of the kernel module, called *buffered* mode. The buffered mode activates write-back and read-ahead caching.

On the server-side, write caches are used [83].

Replication and Fault Tolerance Data and Metadata can be mirrored independently. Mirroring is realized using buddy groups instead of single targets while distributing stripes or metadata. Each buddy group consists of a primary and a secondary target and enables automatic fail-over. They can be created automatically or defined manually to reflect the cluster's topology, i.e., targets from different racks or server rooms can be chosen. Mirroring can then be activated on a file or directory by setting the stripe pattern to `buddymirror`.

If metadata mirroring is activated, the root directory will be mirrored automatically. For effective fail-over, care must be taken that all directories in a path to a mirrored directory are also mirrored. Otherwise, the lookup of the path might fail when metadata services crash because some components of the path can not be checked.

3.6. JULEA

JULEA [40] is a storage framework designed to provide a foundation for research and teaching on storage technologies and is distributed under the LGPLv3. Its key features are flexible storage backends and dynamically adaptable I/O semantics. Despite not being designed as a file system, this functionality is built on top of JULEA using FUSE. Currently, JULEA is under active development, and many features are work-in-progress.

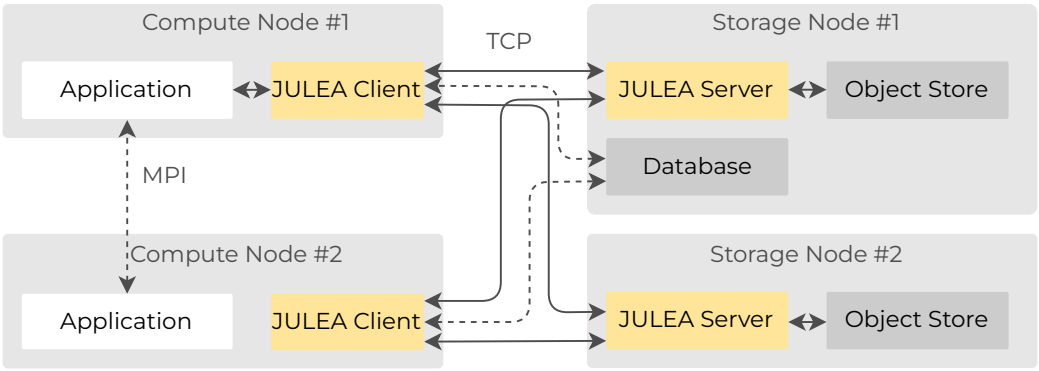


Figure 3.6.: Two compute nodes use two JULEA servers for I/O. Both JULEA servers provide object storage, while the first one also hosts a database server. The clients can directly load the database backend.

Architecture JULEA provides transparent access to several database, object store, and key-value store backends. Only a single type of server is used that dynamically loads backends. A single configuration file specifies which backends to use and which servers will provide access to them. Like MySQL and MongoDB, some backends can be loaded on the client-side. All backends are abstracted, and access is realized by a C interface provided for each backend type. Additionally, there are interfaces for distributed objects and data items. Distributed objects will stripe objects over all servers that provide an object storage backend. There are different distribution schemes. The item interface provides access to a lightweight, file system-like data model built by using the key-value and distributed object interfaces. Data items can be created in a hierarchy of data collections.

Figure 3.6 shows an example setup with two clients and two servers. The database backend is directly loaded on the clients to reduce overhead.

Interfaces Besides access via the library *libjulea*, HDF5 or POSIX access is provided. HDF5 version 1.12 is supported via a *Virtual Object Layer* (VOL) plugin. VOL is an internal abstraction layer of HDF5 that enables backends to implement callbacks for HDF5 functions. Currently, there are two implementations: One uses the key-value interface while the other uses the database interface. Both use the object interface to store the actual data. POSIX access is implemented on top of the key-value and object interfaces using FUSE. In addition, a simple command-line tool can be used to create, copy, delete, and get status information on objects, distributed objects, key-value pairs, items, and files.

Metadata Handling Metadata management depends on the interface used. Metadata is stored as key-value pairs for the item and POSIX interface. The HDF5 database implementation stores HDF5 metadata via the database interface, while the key-value implementation uses the key-value interface. As key-value and object distribution is done manually or by hashing, these interfaces do not need metadata management.

Data Distribution Objects and key-value pairs can either be placed on a specific server or pseudo-randomly distributed using a hash of their keys. Distributed objects can use three

different types of distribution functions: single server, round-robin starting at a random server, and a weighted approach that can be used to balance the storage load.

The POSIX and HDF5 interfaces use the object interface and inherit the pseudo-random approach.

Synchronization and Caching JULEA's semantics can be configured at run-time, and synchronization behavior is, as a consequence, very flexible. Configurable semantics include atomicity, concurrency, consistency, ordering, persistency, safety, and security of operations.

For relaxed persistency settings, operations of a batch are cached and will be executed asynchronously. Both objects and distributed objects can be synchronized manually.

Replication and Fault Tolerance Currently, there are no replication and fail-over features in JULEA. There are three types of data safety options: Network Safety ensures the target server receives data. Storage Safety ensures that data is written to disk on the target server. The last option is to make no safety guarantees.

Chapter 4.

Cluster Setup

This chapter describes the hardware and software setup of the ARM-based and the reference cluster. The theoretical peak performance of data and I/O operation throughput on both systems is determined based on a simple model.

4.1. Hardware

ARM The low-power cluster is built of six Odroid HC4 nodes featuring the Amlogic S905X3 SoC, with four cores at 1.8 GHz, 4 GiB DDR4 RAM, two SATA-3 ports, and a 1 Gbit/s *Network Interface Card* (NIC) [30]. All boards are equipped with a 32 GB SanDisk Extreme SD card [34].

Due to the form factor of the Odroid HC4's case, hostnames of the nodes are `toaster0i` where `i` is the number of the node.

While four nodes, `toaster02-toaster05`, are equipped two 1 TB WD Black HDDs [87], `toaster06` is reserved for monitoring and administrative services of the DFSs. The last one, `toaster01`, is equipped with two 512 GB Samsung V-NAND SSD 860 PRO SSDs [67]. This one is intended to be used for metadata storage.

All nodes are connected to a Netgear GS110EMX switch [56]. This switch has eight 1 Gbit/s ports and two 10 Gbit/s uplink ports. The cluster is connected to the network infrastructure of the Max Planck Institute for Dynamics of Complex Technical Systems via the uplink ports.

The complete cluster, including the switch, is powered by an MW HRP450-15 *Power Supply Unit* (PSU) [53] and consumes 56.36 W, measured over one hour with a standard deviation of 0.14W, in an idle state, with HDDs spun up. For a fairer comparison of the ARM-based cluster with the reference cluster, which does not include the switch in the power measurements, the idle power of the switch can be subtracted. It was measured to be 15.46 W, with a standard deviation of 1.13 W over one hour. Therefore, the adjusted idle power consumption of the ARM-based cluster is 40.9 W.

The ZES Zimmer LMG 450 [88] is used to measure the power consumption of the PSU for the whole cluster. The power meter is connected to a BananaPi M1 [10] via a serial-to-USB converter. The LMG 450 can measure power consumption with a time resolution of 20 Hz.

The clients used to perform the benchmarks are 4 Dell Precision 3650 Tower workstations [18] each with an Intel Core i7-11700 CPU with eight cores at 2.5 GHz, 8 GB RAM, a 512 GB Micron 2300 NVMe SSD [54] and a 1 Gbit/s NIC. Like the cluster, they are connected to the network

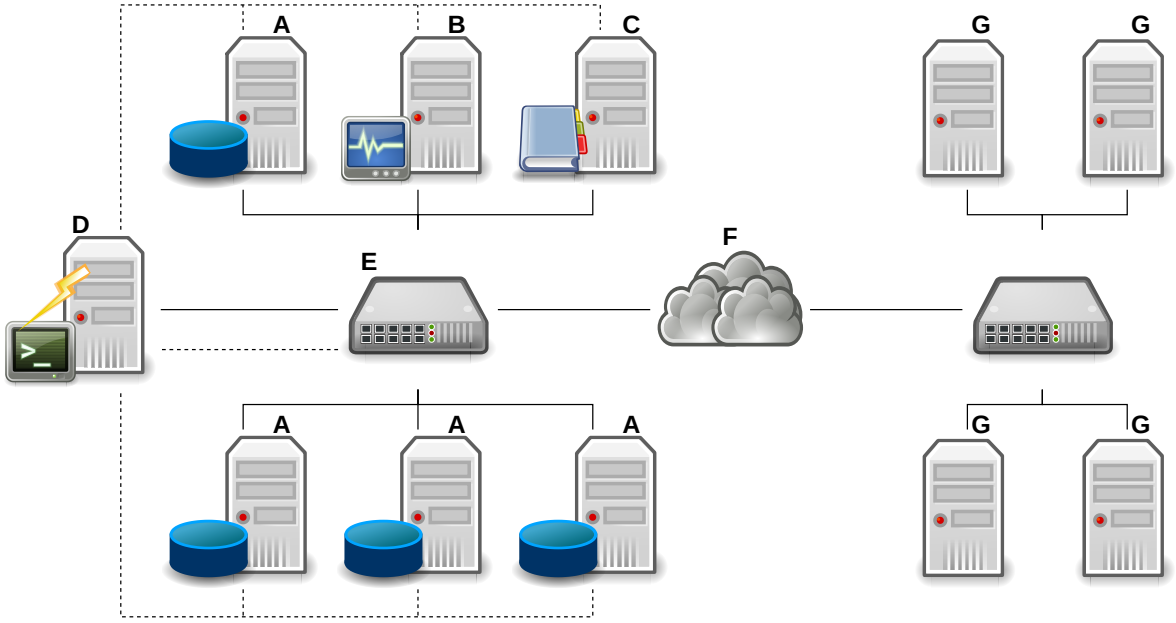


Figure 4.1.: This graphic shows the network topology of the ARM-based cluster. The storage nodes (A), the management node (B), the metadata node (C), and the BananaPi with the power meter (D) are connected to the Netgear switch (E). The dotted lines indicate which devices are included in power measurements. The storage cluster is connected to the clients (G) via the Max Planck Institute Magdeburg network infrastructure (F).

infrastructure of the Max Planck Institute for Dynamics of Complex Technical Systems. The ARM-based cluster setup is visualized in Figure 4.1.

Reference Cluster The cluster used as a reference for measurements is a five node subset of the `west` cluster partition at the University of Hamburg. Each node has two Intel Xeon X5650 CPUs, featuring six cores at 2.67 GHz, 11 GB RAM, and two Intel 82574L Gigabit NICs. The first node, `west1`, is equipped with a 250 GB Western Digital WD2502ABYS HDD [16], while the other nodes are equipped with a 250 GB Seagate ST3250318AS HDD [49]. Like in the ARM-based cluster setup, a ZES Zimmer LMG 450 power meter was used to measure the power consumption of this setup. The five nodes consumed 460.21 W on average with a standard deviation of 18.43 W, measured over one hour.

The clients used to benchmark the reference cluster were four servers of the same specification.

The first and second NICs of all storage and client nodes are likewise connected to the same switch. The first NIC is used per default for communication between the DFS clients and servers.

4.2. Software

ARM The ARM-based cluster nodes run on Armbian Buster 21.08.8, which uses Linux 5.10.81-meson64 as of the date of writing. Armbian [5] is a Linux distribution based on Debian, which

type	client memory	network	server disk
read - ARM	$4 \times (5695 \pm 40.08)$	$4 \times (112.176 \pm 0)$	$8 \times (142.48 \pm 5.59)$
read - x86	-	$4 \times (112.22 \pm 0.06)$	$4 \times (126.5 \pm 6.49)$
write - ARM	$4 \times (6927.12 \pm 23)$	$4 \times (111.34 \pm 0)$	$8 \times (140.09 \pm 8.11)$
write - x86	-	$4 \times (112.22 \pm 0.06)$	$4 \times (115.35 \pm 14.5)$

Table 4.1.: Maximum throughput in MiB/s that can be achieved per component of the data path and the standard deviation of the respective measurement.

is modified and optimized for use on SBCs. The pre-installed *Petitboot* was erased from the HC4's flash memory to use the *uboot* bootloader, which is part of the Armbian image.

The clients run Debian Bullseye 5.10.46-5, which uses Linux 5.10.0-8 as of the date of writing. OpenMPI 4.1.0 with the included MPI-I/O implementation OMPIO was installed from the Debian Buster Repository for parallel benchmarks. In some configurations, other MPI versions and configurations are used to leverage different interfaces of the DFSs. Those deviations from the usual software stack will then be listed.

All storage nodes and clients use NTP to synchronize their clocks with `toaster06`.

Reference All nodes of the reference cluster run Ubuntu 18.04.5 with Linux 4.15.0-147. Two partitions are created on the HDDs. The first is used as the root device, while the second is mounted under `/tmp`. Both partitions are formatted with ext4 and mounted with the `noatime` option. OrangeFS is used for the reference benchmarks. Like the ARM-based cluster setup, one node is used for metadata while the remaining four provide data storage. The HPC package manager Spack [24] was used to install the dependencies of OrangeFS that were not already available. OrangeFS version 2.9.8 was built using *gcc* 11.1.0 and linked against *libattr* version 2.4.48 and LMDB version 0.9.22. On the client, the upstream kernel module was used.

4.3. Theoretical Peak Performance

An estimation of the *Theoretical Peak Performance* (TPP) is needed to assess the performance of the respective systems. For this estimation, some assumptions are required. General assumptions are that no data is cached, and all overhead due to computations and client-server communication protocols is neglected. Servers of the same type, i.e., metadata or data nodes, are assumed to have the same configuration and, therefore, the same performance. At last, enough threads are used to saturate the respective target, e.g., memory or disks.

Let M_{perf} be the sum of the performances of the clients' memory, N_{perf} the aggregated network performance, and D_{perf} the aggregated disk performance of the servers. Then the TPP can be computed as the minimum of these values, see Equation (4.1). This equation is valid for metadata performance and throughput.

$$\text{TPP} = \min(M_{\text{perf}}, N_{\text{perf}}, D_{\text{perf}}) \tag{4.1}$$

Data is assumed to be transferred between the client's memory and the storage cluster's disks via the network. Sequential access on the clients is assumed to result in sequential disk accesses

type	client memory	network	server disk
read - ARM	$4 \times (10751308.12 \pm 38492.56)$	$4 \times (2514.68 \pm 5.95)$	$2 \times (4742.2 \pm 190.07)$
read - x86	-	4×5804.12	$1 \times (233.46 \pm 23.59)$
write - ARM	$4 \times (11182548.5 \pm 56663.41)$	$4 \times (2514.68 \pm 5.95)$	$2 \times (9043.88 \pm 1467.56)$
write - x86	-	4×5804.12	$1 \times (372.94 \pm 75.01)$

Table 4.2.: Maximum of IOPS that can be achieved per component of the data path and the standard deviation of the respective measurement.

on `toaster02` - `toaster05`. Following the data path, the read and write throughput of the client's memory, the network throughput, and the server's disk throughput were measured. For the memory and disk measurements, *fio* was used. The individual parameter files can be seen in Appendix A. On the Odroid HC4, both disks were used in parallel to ensure that the SATA controller and buses could achieve the expected performance. The average results per disk and the standard deviation were pooled to get the values shown in the tables. The measurements of the memory were done using `tmpfs`. Even though this additional layer introduced overhead, it simplified the measurement, and results showed that the clients' memory does not limit I/O. The network measurement was done using `iperf` with a runtime of 180 seconds and three repetitions. The results are presented in Table 4.1 and show that the throughput TPP is limited by the network throughput and is, henceforth, **448.7 MiB/s** for reads and **445.36 MiB/s** for writes. Because both values are close to each other and the minor difference in network throughput is unlikely to be seen in measurements of the DFSs performance, **445 MiB/s** will be used as the baseline for further evaluations for reads and writes.

Based on the values measured on the Dell clients, it is safe to assume that the memory throughput of the server clients in the reference setup does not limit the data throughput TPP. The network achieved a throughput similar to the ARM-based cluster. Like the ARM-based cluster, the throughput TPP of the reference cluster is bound by the network. However, the gap is not as big as for the ARM-based cluster, especially for writes which had a relatively high standard deviation, likely because the disk is additionally used as the root device. The resulting TPP is **448.88 MiB/s** for reads and writes. This value is rounded to **445 MiB/s** to simplify further analysis.

For metadata performance, assumptions about caches and compute and communication protocol overhead remain valid. However, the network connection is only measured from client to server. This is because the client initiates all interactions, and data transfer ends after the server's response. Therefore, the maximum IOPS that can be achieved via the network depend on the number of message pairs processed per second. The IOPS reachable via the network as shown in Table 4.2 depict the limit for a single process on a single client as metadata accesses are always synchronous and serial for a single process. The Odroid HC4 can process more messages if they are sent in parallel. The number of pings sent without waiting for a response can be set using the `-l` option of ping. In other words, pings will then be processed in a pipeline. With a pipeline size of 450 and one client, 100,000 packages could be processed in 1.486 seconds resulting in approximately 67k IOPS. The downside of this load is a high round-trip time which rose from 0.397 ms for serial pings to 6.674 ms for a pipeline size of 450. For pipeline sizes greater than 450, the package loss rose, and communication could not be expected to be reliable anymore. Because metadata access is done serially by each process, four clients are unlikely to hit this limit as it would require 450 processes to perform metadata accesses simultaneously. Starting ping on all four clients in parallel without `-l` reveals that the Odroid

HC4 can process the 2.5k IOPS per client shown in Table 4.2 in parallel. Another assumption that is made for modeling the maximum achievable IOPS is that metadata accesses result in small random accesses on the server's disk. The disk IOPS were measured using *fio* with random 4 KiB accesses that correspond to loading pages from the disk. A list of all parameters is again available in the Appendix A. In contrast to throughput performance, metadata read performance is not limited by the network, which can achieve approximately 10k IOPS under the conditions described above, but by the IOPS achievable on the SSD. Despite the high standard deviation for write IOPS, metadata writes are likely bound by the network. This results in approximately **9.4 k-IOPS** for reads and **10 k-IOPS** for writes.

As for the throughput TPP, the reference servers' memory can safely be assumed not to impair performance. Even when measured serially, far more IOPS could be achieved via the network compared to the ARM-based cluster. However, that benefit can not be leveraged because the HDD of `west1` limits the metadata performance, which results in approximately **233 IOPS** for reads and **373 IOPS** for writes.

Chapter 5.

Setup and Configuration of Distributed File Systems

This chapter describes difficulties during the setup of the respective systems on the unusual hardware. As the system's behavior highly depends on its configuration, small experiments were done to receive proper installations. At the end of the chapter, the comparability of the setups is discussed.

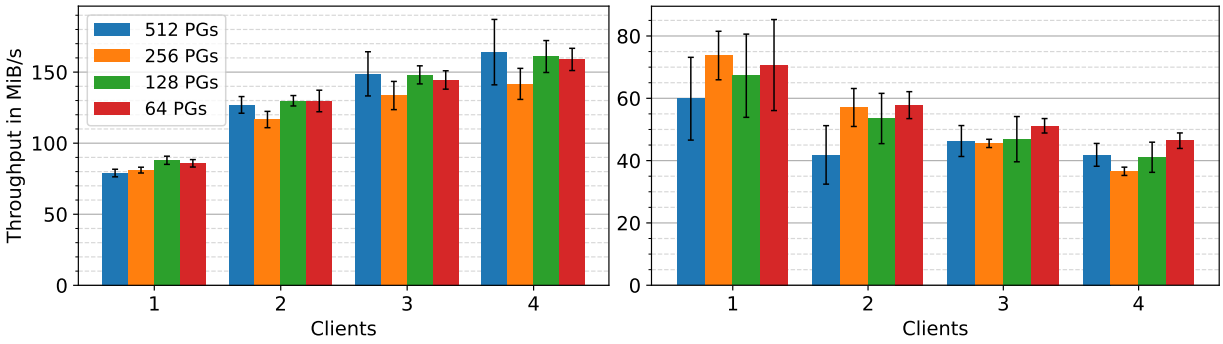
Each system has a large number of parameters that can be modified. Despite that no detailed optimization of each system could be done due to time restrictions, different settings were tested for most systems so they could provide acceptable performance on the cluster. Workloads that focus on parallel data access are more important for most HPC applications than metadata access. Therefore, only the throughput of the systems for this workload is measured in this chapter. If not stated otherwise, the measurements were done using *IOR* either with the POSIX interface for a single file per process or the MPI-I/O interface with one shared file. 36 GiB were written and read with one to four clients, a transfer size of 4 MiB, and a declining block size so that the aggregated size stayed constant. In order to avoid client-side caching, blocks were read by another process than they were written. Additionally, 85% of the clients' memory was allocated. Each measurement was repeated five times, and error bars in the plots show the standard deviation. The same benchmark will be used for comparison in Chapter 6. The *IOR* config file, which describes the benchmark, can be seen in the Appendix A.

5.1. CephFS

The recommended way to set up recent versions of Ceph makes use of containers. Several deployment tools such as Ansible, Rook, Salt, and the Ceph-own orchestrator may be used to deploy the containers. Because virtualization would introduce further overhead on the weak hardware, version 14.2.21, available in the buster-backports repository, was installed. For this version, the recommended approach is to use the *ceph-deploy* tool.

The different components of Ceph were installed as follows: One OSD was deployed for each storage device. The MDS was installed on `toaster01`. The Ceph monitor and management daemons are located on `toaster06`.

The two storage pools needed for CephFS use different CRUSH rules to distribute objects. While the data pool uses all HDDs and manages replicas on the node level, the metadata pool uses the two SSDs and manages replicas on the OSD level. The metadata pool needed the change of



(a) reads (b) writes

Figure 5.1.: Influence of the number of placement Groups used in Ceph on data throughput using POSIX access with *IOR*, a transfer size of 4 MiB, one file per client, and an aggregated file size of 36 GiB.

replication management level so that the goal of two replicas could be fulfilled without adding another metadata node. Ceph was configured to produce two replicas and return immediately after one replica is written, allowing for a fairer comparison with the other systems.

Because of the low RAM of the nodes, cache sizes and memory limits needed to be changed. The monitor's cache size was set to 2 GiB, and OSD memory targets were reduced to 1.5 GiB.

Based on the fact that the actual data is stored inside storage pools, their configuration was of particular interest. Ceph is built to run directly on storage devices without software or hardware RAID. Instead, replication is used to ensure data safety and high reliability. For each storage pool, the replication goal and the number of needed replicas can be configured. The standard is to produce three replicas and return control to the user after two are written. Because no private network for replication is available, both values were reduced by one. Returning after only one replica is written should improve the latency of I/O calls.

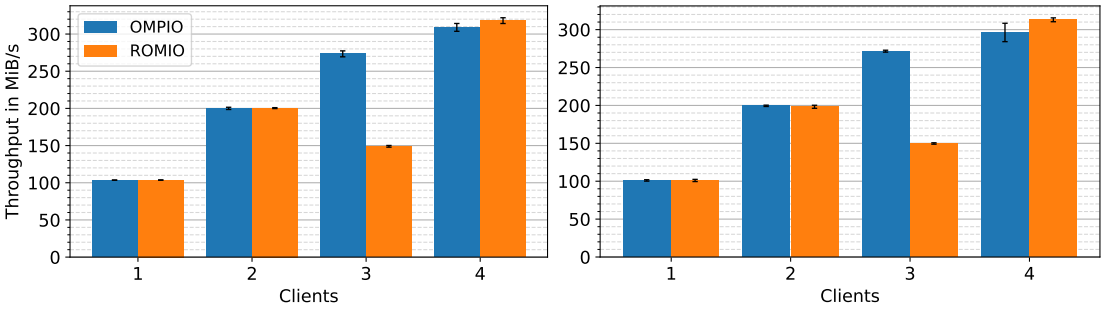
The Ceph documentation states that the number of *placement groups* (PG) for a storage pool greatly influences its behavior. Because the Ceph Docs recommend low PG counts for metadata pools, it was set to 64 for all measurements. The influence of the placement groups on the data pool was measured using 2^i PGs for the file system's data pool with $6 \leq i \leq 9$ and access via the POSIX interface. Figure 5.1 shows that lower counts of placement groups performed slightly better while writing than higher counts. For reads, no apparent effect can be seen. Nevertheless, the influence on performance is not as significant as expected. It is possible that only bigger clusters with more OSDs reveal noteworthy differences in performance. In general, the throughput is below expectations regarding the theoretical peak performance. Possible reasons for that will be discussed in Chapter 6. The rest of the benchmarks are done with 64 placement groups for the data pool.

5.2. OrangeFS

OrangeFS version 2.9.8 was built with *gcc* version 8.3.0 and LMDB 0.9.22 from the Buster repository using the configuration shown in Figure 5.2.

```
./configure --prefix=${HOME}/ofs --enable-shared --disable-static
   ↪ --enable-racache --with-db-backend=lmdb
   ↪ --enable-external-lmdb
```

Figure 5.2.: Build-configuration for OrangeFS.



(a) reads (b) writes

Figure 5.3.: Throughput of OrangeFS using MPI-I/O via OMPIO and ROMIO with *IOR*, a transfer size of 4 MiB, and an aggregated file size of 36 GiB.

As explained in Section 3.2, OrangeFS features only a single type of daemon, which is running on all nodes equipped with disks. Metadata is stored exclusively by `toaster01`, while the other nodes store the data.

Because OrangeFS offers no redundancy for data that is not read-only, the two disks of each node were mirrored. Instead of introducing additional storage layers, like mdadm or LVM2, ZFS was used. ZFS combines software RAID and volume management with a copy-on-write file system. Storage Pools in ZFS combine physical storage devices using different RAID schemes. Within a ZFS pool, multiple ZFS datasets, each providing a file system, can be created. Three datasets were created within a mirrored pool on top of the two disks. Two of the datasets, `storage/data` and `storage/logs`, used the default settings except that the modification of file access times was turned off. In addition to deactivating updates of access times, the `storage/meta` datasets' record size was reduced from the default of 128 KiB to 4 KiB because DFS metadata operations are expected to create many small random accesses. As ZFS uses copy-on-write, this access pattern would result in inefficient read-modify-write cycles for large record sizes.

OrangeFS has many possible interfaces. First measurements of the POSIX interface via the kernel module showed irregular behavior for reading. Four clients reading a 2 GiB file each achieved only 12.41 MiB/s. This behavior is not restricted to the ARM-based setup and could be reproduced using one client machine as an OrangeFS server for another client. The problem could not be observed on the reference cluster, which also runs on OrangeFS. Therefore, this problem could be related to the kernel module version running on the clients of the ARM-based cluster. If the VFS page cache is not used, i.e., files are opened with the `O_DIRECT` flag, the same benchmark described above achieved 291.45 MiB/s, which is closer to expectations. Consequently, further measurements of OrangeFS on the ARM-based cluster were done using direct I/O.

Because the defaults of MPI-I/O implementations will use POSIX to implement the actual I/O access, the same applies here. However, OpenMPI's implementation of the MPI-I/O standard, OMPIO, directly supports OrangeFS if linked against *libpvfs2*. Another option is to use the

Abstract I/O Device (ADIO) layer implementation provided by OrangeFS. ADIO is an internal abstraction layer of ROMIO, another implementation of MPI-I/O, that file systems can use to provide a native MPI-I/O interface [81].

To test both interfaces, *IOR* was compiled in two versions. The first version was linked against MPICH version 3.4.3, which uses ROMIO, while the second one was linked against OpenMPI version 4.1.0, which uses OMPIO. Both MPI versions were built with support for OrangeFS. The results are presented in Figure 5.3 and show that both interfaces achieved comparable performance. However, ROMIO had a reproducible performance drop for reads and writes with three clients for no obvious reason. As a consequence, OMPIO was used for further evaluation.

5.3. GlusterFS

Most possible configurations of GlusterFS are not suitable for HPC workloads, e.g., highly parallel, throughput oriented workloads:

Distributed volumes place whole files on storage servers. The result would be an unbalanced load for large files, and access to a single file would be limited by the throughput of disks and the network of the specific server. Therefore, this is not a viable option.

Though replicated volumes theoretically enable parallel reads, writes do not benefit. Based on the experiences with MooseFS, replication is likely to reduce the throughput of multiple users.

Dispersed volumes use striping over multiple servers in combination with erasure coding. Erasure coding ensures the reliability of stored data by adding redundant data blocks so that data can be restored when blocks are lost. There are different techniques for erasure coding [7]. This approach enables both parallel reads and writes. The downside is that the stripe size depends on the number of servers used in the volume and the redundancy count, see Equation (3.1). GlusterFS considers configurations that result in stripe sizes equal to a power of two as optimal. Other configurations are disadvantageous for most workloads that align storage access to powers of two [77]. No optimal configuration can be achieved with four GlusterFS bricks unless the redundancy count is set to two, which leads erasure coding to absurdity. In addition, bricks should not be placed on the same servers to ensure high data availability. The only way to achieve an optimal configuration with minimal changes to the cluster was to exchange the SSDs of `toaster01` with the identical HDDs used for the other nodes. This configuration also solves the problem that GlusterFS can not benefit from multiple storage tiers in a single volume because metadata is not separated. As recommended by the GlusterFS documentation [77], bricks were formatted using XFS with a maximum inode size of 512 B. The dispersed volume was created using `toaster01` - `toaster05` and a redundancy of one, resulting in a stripe size of 2 KiB.

Scaling up the capacity of dispersed volumes without changing the stripe size can be achieved by distributing files across several dispersed volumes. GlusterFS supports this configuration via distributed dispersed volumes. Nevertheless, data throughput for a single shared file would not scale with capacity in this case.

Figure 5.4 shows that the resulting configuration achieves a reasonable throughput that matches expectations for access via the POSIX interface. The higher write throughput compared to the

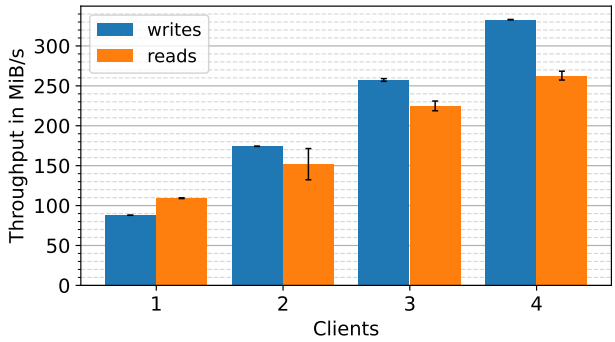


Figure 5.4.: Throughput of GlusterFS using POSIX access with *IOR*, a transfer size of 4 MiB, one file per client, and an aggregated file size of 36 GiB.

read throughput suggests that the servers store write accesses in a cache and return the call before the caches are flushed. However, no apparent bottlenecks or problems can be seen, so this configuration will be used for further comparison.

5.4. MooseFS

MooseFS version 3.0.115 was available in the Buster backports repository.

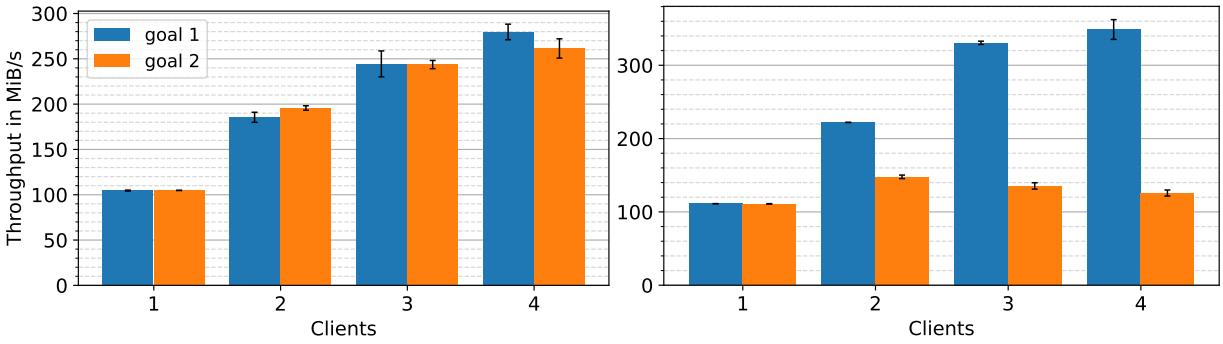
The master server was set up on `toaster01` and configured to ignore changes to access times. Though metadata in MooseFS is kept in memory, the master server can be expected to benefit from the SSDs because of the on-disk journal of metadata modifications. According to the documentation [39] one metadata entry consumes 300-350 B. Assuming 1 GiB of main memory is used by the operating system, page caching, and the master server (without metadata entries), 3 GiB are left for metadata. Therefore, approximately 9.2 million files can be stored without overcommitting memory. Although the system is limited in the number of files and directories, it is still viable to use MooseFS on the ARM-based cluster, depending on the expected workload.

MooseFS, like most file systems surveyed, relies on local file systems to provide persistent storage. Replication is optional and can be set per file or directory and is, like other MooseFS specific settings, inherited from the parent directory. Therefore, the same ZFS configuration used for OrangeFS was used again to ensure data safety even without replication.

The MooseFS chunkservers were installed on `toaster02` - `toaster05`. The monitoring server was installed on `toaster06`.

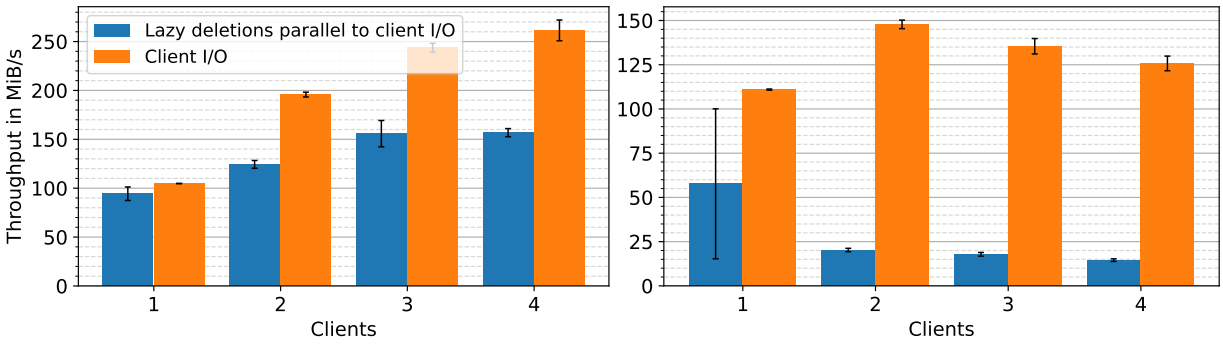
MooseFS client utilities provide tools to configure MooseFS-specific settings. Settings of interest for the performance are replication and trash times. Replication multiplies the amount of data transferred via network and written to disk. Therefore, its influence on performance is intuitive. The trash time in MooseFS describes how long files and their metadata are kept in the hidden companion file system of MooseFS. After this time is exceeded, files will be deleted according to the chunk loop of the master server described in Chapter 3. The deletion of chunks may overlap with client I/O and competes for disk usage on the chunk servers.

Figure 5.5 confirms that the doubled amount of data that is produced by replication impact the performance of the system. This measurement was done with the default trash time of 24 hours, so no asynchronous delete operations were overlapping with the following operations.



(a) reads (b) writes

Figure 5.5.: Influence of Replication in MooseFS on the performance for access via the POSIX interface with *IOR*, a transfer size of 4 MiB, one file per client, and an aggregated file size of 36 GiB.



(a) reads (b) writes

Figure 5.6.: Influence of deletion of old chunks in MooseFS parallel to client I/O via the POSIX interface with *IOR*, a transfer size of 4 MiB, one file per client, and an aggregated file size of 36 GiB.

While reads show no meaningful differences, the throughput for writes achieved its maximum with two clients. Less than half of the throughput could be achieved for four clients when data was replicated. In addition to the doubled amount of data, throughput could be impaired by the resulting accesses patterns on the local disks. Nevertheless, to confirm this, detailed I/O traces would need to be analyzed, which is out of the scope of this thesis.

Figure 5.6 shows the result of a measurement that collided with the asynchronous deletion of old data. The measurement was done using POSIX access with files that had a replication goal of two and were configured for a trash time of 24 hours. For reference, the result of the replicated measurement described above is plotted again. The high resource contention between the ongoing delete operations and the replicated I/O results in poor performance for writes and reads. As a result, such a configuration is not useful in practice. For trash times of zero, however, deletes would always overlap with the currently running benchmark. Therefore, the default trash time of 24 hours was kept for further benchmarks, and care was taken that measurements were not started parallel to ongoing delete operations.

Another critical aspect to keep in mind is that the files' metadata in the trash needs to be kept. For the benchmark described above, this imposes no problem. However, metadata benchmarks will create and delete many directories and small files in multiple iterations and for multiple

clients. Consequently, metadata benchmarks were done with a trash time of 0 to avoid memory problems on the master node. However, as small files will not even fill one MooseFS chunk, the overlapping deletes are expected to have less impact on metadata access. Based on the results of the replication analysis, the replication goal for metadata benchmarks was set to one.

5.5. BeeGFS

Up to version 7.2.6, several issues complicated BeeGFS's setup on the ARM-based cluster. The source code needed a patch to compile on different architectures than x86. The reason was that the BeeGFS-own `Atomics` template class was implemented using the *gcc* built-in `__sync_*` functions that are not defined for the `boolean` type. Nevertheless, the class was used with this type as a parameter, and the compilation failed. This problem did not appear for the x86 architecture, which used a different implementation activated by pre-processor directives. As a hotfix the C++ built-in atomics were used to implement `Atomics`. The modified server was able to run without visible errors. However, the BeeGFS client kernel module until version 7.3.0 was incompatible with Linux 5.10.0-08, which was used by the Debian Bullseye client machines. Fixing this issue was out of the scope of this thesis.

The most recent version, 7.3.0, released on April 13, 2022, fixed both problems. The client module, however, could not be loaded due to the *kernel_lockdown* feature, which was activated on the clients. As the new release came close to the deadline of this thesis, no benchmarks were done on BeeGFS.

5.6. JULEA

JULEA had a straightforward setup on the cluster. The database backend was configured to use SQLite, while LMDB was chosen as the key-value backend. Both were set up on `toaster01` to benefit from the high IOPS of the SDDs. The POSIX backend was chosen as the object storage backend, and `toaster01-05` were set as object stores.

However, all connection attempts via FUSE or *julea-cli* were refused by the servers. The JULEA test suite ran on `toaster06` and failed for all tests that would have required network connections to the cluster. Thus, the problems are not related to the different architectures of clients and servers. Further investigations using the tracing capabilities of JULEA's debug build showed that the message handling function of the server, `jd_on_run`, was not called for connection attempts via JULEA client tools. However, connection to the port that JULEA listened on via *netcat* could trigger the call of `jd_on_run`. General network issues can therefore be excluded.

Due to the restricted time, no further debugging could be done, and JULEA was not benchmarked.

5.7. Comparability of Configurations

As most systems have many parameters, only a small subset of combinations could be evaluated. The resulting configurations are close to the defaults and aim to achieve comparability between the systems as far as possible. Each system was designed and built with specific use cases in mind. As a result, each system has its own set of features and shows different behavior. GlusterFS, for instance, is not explicitly designed as an HPC file system but as a storage solution for enterprise file sharing and as a storage platform for virtualization [66]. Swapping the SSDs for HDDs in the GlusterFS setup does not change the data throughput TPP, which is still bound by the clients' network throughput, but enables a better load balancing between the disks while writing large amounts of data. That is an advantage that the other systems do not have. The TTP for metadata access does change because of the hardware modification and the fact that GlusterFS manages metadata on each node using the inodes of the local file systems. SSDs can achieve far more IOPS than HDDs which was confirmed by the measurements in Chapter 4. Consequently, the metadata TPP is lower for GlusterFS when using the same model as in Chapter 4. However, the GlusterFS setup would not have been reasonable without this modification.

Another aspect is that the stripe size of each system is different. While OrangeFS and CephFS have configurable stripe sizes, the stripe size of GlusterFS depends on the setup, and MooseFS has a hard-coded stripe size of 64 MiB. Changing the sizes, where possible, was not done because the systems are shipped with defaults tested over the years.

As pointed out in [71, 80] the interaction between a DFS and a local system are not trivial, and the choice will impact the overall performance. However, optimizing the storage stack for each DFS was not possible due to time restrictions. Therefore, ZFS was used for OrangeFS and MooseFS because they do not have data safety features, or these features are optional.

From the perspective of power measurements, comparability is also not trivial. Again, one example is that the SSDs were swapped for HDDs for GlusterFS. This change in hardware raises the cluster's idle power consumption and peak power consumption. In addition, `toaster06` was unused for OrangeFS and Gluster, apart from the NTP server, which could, in theory, be placed on each server. However, this node was active the whole time and therefore consumed energy. That can be justified by the fact that those systems would also need monitoring solutions in real-world setups. Even if the comparison based on average power consumption alone may not be justified, other metrics will be discussed in Chapter 6 and allow for a fairer comparison.

Chapter 6.

Performance and Energy Efficiency Analysis

In this chapter, possible metrics for analyzing the performance and energy efficiency of the ARM storage cluster are discussed. The systems are compared using benchmarks for different workloads and different metrics. Based on the results, implications for the efficient usage of the current ARM storage cluster and design recommendations for future clusters are derived.

6.1. Workloads

The behavior of file systems strongly depends on the I/O workload. Not all systems will be suitable for every workload. Different kinds of benchmarks need to be done to get an overview of the capabilities of each system under analysis. Because the goal is to build energy-efficient HPC storage, the focus is on workloads common for scientific applications. Such workloads include sequential, and parallel writes and reads of large files, e.g., checkpoints or results of simulations. Another workload is generated by using the file system as a shared home directory. That will result in interactions with many small files, e.g., software compilation. All benchmarks are done with one to four clients to analyze scaling behavior.

Parallel sequential access to large files One workload of interest comprises coordinated, parallel, sequential access to a single or a few open files. That, for example, is the case for simulations writing results or checkpoints to disk. *IOR* was used to simulate this behavior. Because high-level interfaces most likely use either POSIX or MPI-I/O as an underlying storage layer, these interfaces were used for measurements. Using these interfaces 36 GiB were written and read using one *IOR* segment and declining block sizes so that the aggregated size stayed constant. The transfer size was set to 4 MiB, which is the default stripe size of CephFS and aligns with the stripe sizes of the other file systems. Client-side caching would bias measurements because most parts of the DFSs would not be involved at all. Therefore, reads were shifted by one rank, i.e., if a block was written by the process of rank n out of m processes, rank $n + 1 \mod m$ was used to read it. On top of that, 85% of the clients' memory was allocated before the measurement started. Besides minimizing the influence of the local caches, this simulated the memory consumption of a running HPC application. Each process created its own file using POSIX, while all processes shared a single file with MPI-I/O. All parameters can be reviewed in Appendix A.

Metadata performance on many small files In addition to throughput-oriented workloads, metadata-focused workloads are of interest. Some applications, unfortunately, synchronize using files in a shared directory. One example is the ORCA simulation tool [55] for quantum-chemical simulations. Working with many small files will also generate focused metadata workloads. Therefore, two benchmarks were done using *mdtest*. The first one was designed to measure the performance of large directories. All participating clients accessed two shared directories and performed create, `stat` and delete operations on 25000 directories and 25000 files. 1 KiB is written and read to and from each file. Writing the data happened directly after creating the file and is included in the IOPS measured for file creation. That ensures that the respective file systems need to allocate space for the files. The second metadata benchmark used a shared binary tree of directories with a depth of four. The same operations as previously were done on 600 directories and 600 files in each directory.

Source code compilation as a mixed workload Another typical workload found in the home directories of HPC clusters is the compilation of source code. In contrast to the synthetic benchmarks described above, this is a real-world benchmark and is expected to put the most stress on the systems. Compiling the Linux kernel version 5.16 with the default configuration was used as an exemplary workload. The complete repository had an aggregated size of 5.5 GB and consisted of 4902 directories and 74291 files. However, 4.3 GB alone were consumed by the `.git` directory, which stored the history and metadata of the repository and contained 26 files in 16 directories. The average file size, excluding the `.git` directory, was 14.37 KiB. The benchmark was split into three phases. First, the *git* repository was cloned. Then the *git* head was set to the tagged release 5.16 and distributed to the local disks of each of the four clients. Therefore, the clients' internet connection could not introduce random errors to the measurement. In the first phase of the benchmark, each participating client copied the Linux repository to a directory on the distributed file system. Each client worked in its own directory. In the second phase, the default kernel configuration got generated, and the kernel was compiled using four threads per client. During this phase, many files were generated from the input. The exact count of generated files was not measured. However, one object file was created for each C translation unit. In the last phase, the complete directory was deleted. The time measured per phase was the maximum of the times the clients took to complete it. The benchmark was implemented as a *bash* script using *ssh* and the bash built-in command *wait*. An excerpt of the implementation can be seen in Appendix A. As for the other benchmarks, the workload was scaled from one to four clients, and each measurement was repeated five times. Because the compile-time also depends on the computing power of the client machines, this benchmark could only be used to compare systems using the same client configuration. Therefore, no comparison with the reference cluster was made. Instead, the same benchmark was done on the local file systems of the clients to have a reference value.

6.2. Data Collection and Discussion of Errors

During the benchmarks, performance and power values were collected. Performance measurements were done with *IOR* and *mdtest* for synthetic benchmarks and with a simple bash script for a real-world workload. The power samples were collected with 20 Hz like described in Chapter 4.

The samples were collected by the respective computers connected to the power meter using a Python script that interacts with the serial port via *pyserial*. Timestamps for the samples were generated on the computer after the value was read from the serial terminal. That introduces a systematic error when synchronizing power samples with I/O benchmarks. Another error is introduced by the clocks of the I/O-performing clients and the clock of the computer connected to the power meter. Although they are synchronized using NTP, errors of a few microseconds can not be avoided. However, the errors are minor and only influence the beginning and end of the benchmark. Because of the high sampling rate, both effects can, in practice, be neglected. Because the reference clusters measurement does not include the switch connecting the nodes, the idle power of the switch of the ARM-based cluster was subtracted from all measurements. The switch was included in power measurements in the first place to show the low energy consumption of the complete cluster, including the network architecture. Nevertheless, the results of the computed metrics are slightly impaired by this because the switch can be expected to consume more energy under a high load.

Like in all measurements, random errors can not be avoided. After all, the performance of the file systems depends, among other things, on the scheduling done by the operating systems. Additionally, physical measurements, like the power measurement, always involve random errors. All measurements were repeated five times to reduce their influence. The results show the mean of these five measurements, and the error bars depict their standard deviation.

6.3. Performance and Energy Metrics

Performance Multiple metrics can be used to measure the performance of a storage system. The most simple performance metric measures the time needed to perform a fixed task. Some other examples are the I/O rate in *I/O operations per second* (IOPS), the data rate or throughput as the amount of data transferred per second, for example MiB/s, and the response time or latency of the system measured in milliseconds [14]. Other metrics can be derived from these. One more complicated metric is to measure the maximum data rate for which a given percentage of the requests is completed below a given latency threshold [14].

In this work, the performance of metadata-focused workloads was measured in IOPS, while workloads focused on data throughput were evaluated using the data rate. The compile benchmark, which created a mixed workload and depended on the clients' compute performance, was measured in the time needed to complete the tasks.

Energy Efficiency Several energy efficiency metrics can be derived from the collected data. However, choosing a specific metric is not trivial, as there is no single optimal metric indicating energy efficiency [64].

The most straightforward energy efficiency metric that comes to mind is the average power consumption while performing a task. Though it is simple and easy to measure, it can lead to wrong conclusions when analyzing energy efficiency as no information about the run-time is given. A system consuming 10 Watt while performing a task that takes 10 seconds is less efficient than another system that consumes 15 Watt but takes only 6 seconds to complete the same task. Therefore, a useful metric has to be related to the system's performance to obtain insight into the systems' energy efficiency.

One viable possibility is to measure the energy efficiency using the total energy consumed for completing a task. For the example above, the first system would consume 100 Joule while the second one would consume 90 Joule. So this metric does indeed indicate which system is more energy-efficient for this task. In fact, it could be regarded as a special case of the Energy-Delay Product that will be discussed later. A disadvantage of this approach is that both systems need to be able to perform the same task in a reasonable amount of time. Depending on the scale of the systems, this could be a problem.

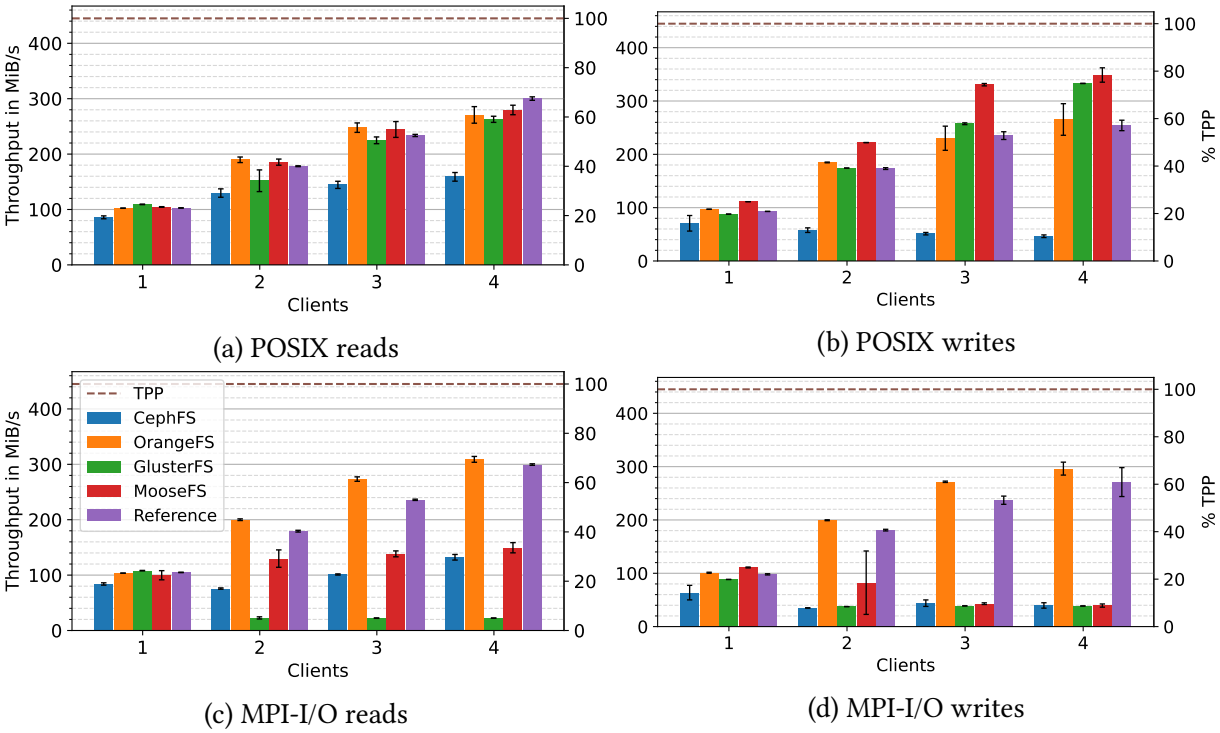
One metric that solves the problem of fixed benchmark task sizes is data or I/O throughput per Watt [13], which is a commonly used metric for evaluating and comparing storage energy efficiency. In addition to abstracting from the task size, this metric has a unit that can be interpreted well as it describes the transferred amount of data or, respectively, the I/O operations Joule. Therefore, this metric will be used for comparison in this thesis.

Another energy metric considered measures the capacity of the storage system per Watt [13]. Because of growing storage demands and therefore growing storage systems, optimizing systems regarding this metric is critical for the cost-efficient and environmentally friendly operation of data centers. However, as the reference cluster is not explicitly built as a storage cluster and has only one 250 GB disk per server, a comparison using this metric would not be fair in this case.

Energy-Delay Product All metrics previously discussed focus on either performance or energy efficiency. When evaluating storage solutions for large-scale deployments, like HPC clusters, energy consumption is already an essential factor that is expected to become even more relevant because of increasing demands on capacity and performance. In addition, Geveler et al. [26] found that for simulations, in some cases, energy savings might lead to performance drops. In such cases, they motivated the use of the *Energy-Delay Product* (EDP) [33] as a fused metric describing energy efficiency and performance at once. The EDP is computed as the product of the total energy E consumed while performing a task and the time t needed to complete the task (Equation (6.1)).

$$\mathrm{EDP} = E \cdot t^w, \quad w \in \mathbb{N} \tag{6.1}$$

Though the energy-delay product was initially developed for hardware design, it is also useful when evaluating software, as done by Georgiou et al. [25]. Depending on the performance requirements, the time may be weighted (Equation (6.1)) [44]. Because the focus of this thesis is on energy consumption, w was set to one. Another benefit of the EDP is its usage for tuning storage systems, which enforces that balanced configurations are found. Neither performance nor energy-saving efforts are neglected in favor of the other one. In practice, care must be taken in choosing w so that tuned systems meet the expectations on performance. One disadvantage is that the unit of the EDP is hard to interpret and even changes with different weights. Consequently, the EDP was normalized using the lowest value per comparison.



(a) POSIX reads

(b) POSIX writes

(c) MPI-I/O reads

(d) MPI-I/O writes

Figure 6.1.: Throughput of all systems under comparison. The left scale shows the throughput in MiB/s whereas the right one shows throughput as percentage of the theoretical peak performance.

6.4. Performance Comparison

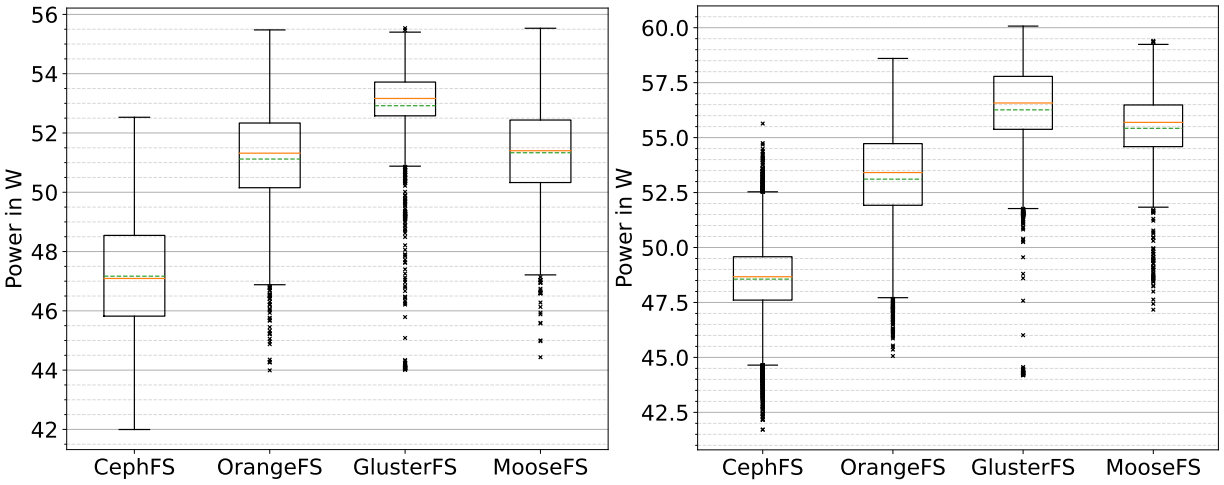
This section presents and discusses the performance aspects of the benchmarks' results.

6.4.1. Parallel Sequential Data Access

Figure 6.1 shows the result of the throughput-focused benchmarks. The discussion is split into the POSIX and the MPI-I/O benchmark.

POSIX Nearly all systems show good scaling behavior. The exception is CephFS which scales worse than all other systems for reads and even shows a declining throughput for writes. That is most likely related to data replication over the public network leading to a doubled network load and contention for disk accesses between ongoing replication and regular I/O requests. This idea is further supported by the logs of the OSDs that revealed slow operation warnings due to waiting times for sub-operations, i.e., replicating objects. As pointed out by Just [35], the Ceph OSD service utilizes many threads, leading to performance issues for a few cores as context switches introduce additional overhead. Another impacting factor might be introduced by CephFS' lazy deletes [12], which are done asynchronously by an MDS and probably overlapped with reads and writes, resulting in lower throughput as shown for MooseFS.

Figure 6.2 shows the distributions of the power samples measured during the last iteration of the throughput benchmark using POSIX access. CephFS, which performed overall at worst,



(a) POSIX reads

(b) POSIX writes

Figure 6.2.: Comparison of the distributions of the power samples measured during the last iteration with four clients for writing and reading via the POSIX interface. The orange lines denote the median and the dotted green ones the mean of the distributions.

had the lowest average power consumption, indicating that the hardware is probably not fully utilized, resulting in poor performance.

Except for CephFS, all systems had a similar read throughput. Differences can be seen for writes, where MooseFS and GlusterFS outperform OrangeFS, which reached approximately 60% of the TPP, while the other two systems achieved nearly 80% of the TPP. In contrast to CephFS, GlusterFS does not seem to be impaired by adding redundancy and writing 25% more data. It showed comparable or even better performance than OrangeFS, which uses no replication or erasure coding. That indicates that erasure coding is a viable possibility to enable resiliency against storage node failures on the ARM-based cluster without sacrificing performance.

The reference cluster performed similarly to the OrangeFS setup, which is not surprising as it also runs on OrangeFS. One difference, however, is that the standard deviation on the reference cluster was smaller. Though this is no clear evidence, given only five samples, it could be that the reference cluster is more stable in terms of performance.

Interestingly, writes are faster than reads for GlusterFS and MooseFS starting from two clients. That could be related to server-side write-back caching applied by the two systems. I/O requests could be aggregated, which would enable more efficient disk access. Read requests can not be cached efficiently for two reasons: The first one is that data is only read once, so there are no further accesses that benefit from caching. Second, even when writes are cached, the caches would be freed when the data is read due to the low memory of the nodes.

All systems could undoubtedly be tuned to achieve even better performance. The complicated part of this is understanding the interactions between the respective DFS and the local file systems. Taking OrangeFS as an example, the relation between the stripe size of OrangeFS and the record size of ZFS need to be taken into account. As shown by traces of MPI-I/O calls and OrangeFS' internal Trove layer, which does the actual disk I/O, single client-side write calls can result in multiple server-side Trove write calls [51]. Those should align to ZFS record sizes, if possible, to minimize read-modify-write cycles.

Overall the different DFSs, except CephFS, performed well compared to the reference cluster, which shows that DFSs indeed do not need the high compute power provided by typical servers. If this behavior scales to a far higher count of nodes, ARM-based SBC clusters are a viable alternative to traditional x86 storage clusters in terms of throughput.

MPI-I/O Access to a single shared file using MPI-I/O had a significant impact on the throughput of the respective systems. While access with one client showed no difference compared to POSIX access, no meaningful scaling can be seen for any system except for OrangeFS. OrangeFS benefits from the direct support of OMPIO and performs better than via the POSIX interface. That is likely to be related to the fact that the VFS is bypassed as OMPIO links to *libpvfs2*. Therefore, fewer storage layers are used compared to POSIX access. Even though, OrangeFS' maximum performance is still lower than for MooseFS or GlusterFS while writing.

MPI-I/O uses the POSIX interface to perform the actual I/O for all other systems. Because all systems, except OrangeFS, claim to be nearly POSIX compliant, the strict semantics for shared access is enforced using locking strategies that add overhead and impair performance. GlusterFS' low throughput, even for reads, hints that the servers implement more aggressive locking for shared file access compared to the other DFS. Competition for the lock results in serial I/O and introduces lots of overhead. Hence, only approximately 20 MiB/s could be reached for more than one client. For writes MooseFS, GlusterFS and CephFS achieved comparable performance.

OrangeFS reached a better performance on the ARM-based setup than on the x86-based setup. That is possibly related to the fact that the x86 setup used only a partition on a disk also used as the root device. Therefore, the disk bandwidth was shared with all I/O done by the operating system.

Overall, the measurement showed that MPI-I/O shared file access is not feasible in practical applications if the underlying file system exposes no native interface and enforces POSIX semantics.

6.4.2. Metadata Access

Many data sets were collected during the metadata measurements. In order to not go beyond the scope of this thesis a selection for the interpretation of the data needed to be made. The focus was on the benchmark that measured the IOPS for large directories. In general, the measurements done on the binary tree showed no utterly different behavior. The IOPS on the directory tree were higher than on the big directory, especially for MooseFS and GlusterFS. However, CephFS client-side caching capabilities distorted the measurement. For one client, the complete tree got cached while being created, and following `stat`-calls were served out of the cache resulting in 440 k-IOPS, which are not possible on the ARM-based cluster. Therefore the analysis is limited to the big-directory benchmark, though the results of the other benchmark can be seen in Appendix B.

Two errors happened in the design of the big-directory benchmark. Initially, it was planned to use only one directory. Due to the confusing naming of *mdtest*'s options, the first big directory contained a second one, and entries were split among them. The second error was that the number of entries is not divisible by three. Consequently, only 49,998 instead of 50,000 files and directories were created for three clients. Though the first error possibly resulted in a higher

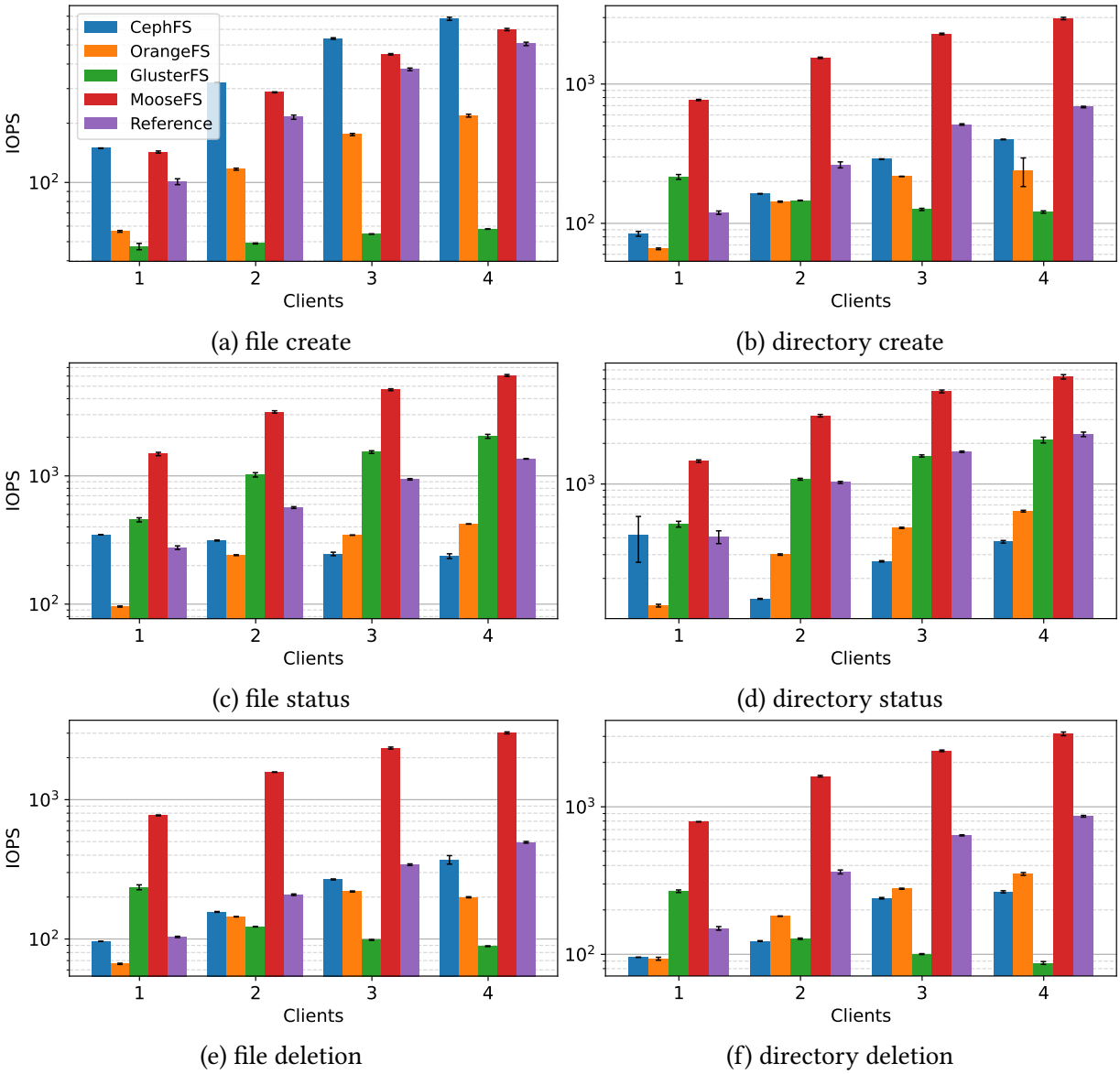


Figure 6.3.: Comparison of metadata performance measured in IOPS for different tasks.

performance, the second can be neglected as the influence of the two entries is minor. By the time both errors were detected, they could not be corrected anymore.

Figure 6.3 shows the result of the big-directory benchmark. Unlike in the data throughput plot, the systems' theoretical peak performance is not shown as a horizontal line. The reason is that most systems actually violate the assumptions of the analysis made in Chapter 4. MooseFS managing metadata in memory or GlusterFS distributing metadata over all bricks in a volume are just two examples. That shows that, in contrast to data throughput, metadata operations can not be covered by such a simple model for all DFS and that the TPP depends strongly on the respective DFS architecture. OrangeFS on the reference cluster, for example, exceeded its theoretical peak performance for most measurements. The reason could be related to the use of LMDB in OrangeFS, which uses memory mappings for I/O [31]. Memory mappings provide direct access to the Linux page cache [32]. One advantage of this approach is that reading or writing does not involve copies between the page cache in kernel address space and the process address space. Another advantage is that multiple updates could be aggregated before syncing

the mapping with the file on disk resulting in larger disk accesses. In addition, metadata could possibly be cached by the OrangeFS server. The use of caches and memory-mapped I/O violates the explicit assumption of no caching and implicit assumptions made in the parameters of *fio* which was used to measure disk performance.

While CephFS performed well for directory and especially file creations, its performance for the other workloads was only mediocre and showed even declining IOPS for more clients added during the file `stat`-call part of *mdtest*. The initial drop in performance for directory `stat`-calls happened due to the cache management of CephFS. While for one client, a certain number of requests could be served out of its cache, the cache is likely to be invalidated for shared directories leading to the drop for two clients. Nevertheless, adding more clients still shows an increase in IOPS. Although the CephFS documantation [12] states that deletes are performed asynchronously, the actual performance for deletes shows no big difference compared to OrangeFS on both architectures. This inconsistency is further supported by the high delete IOPS of MooseFS, which actually implements asynchronous deletes.

Interestingly, OrangeFS performed better on the x86-based reference than on the ARM-based setup, even though the reference cluster only uses HDDs. The reason is unlikely to be related to the lower compute capabilities of the Ordoid HC4 because MooseFS on the ARM-based cluster outperforms OrangeFS on the x86-based cluster. Limits due to the memory bandwidth can be excluded for the same reason. One possible reason could be the larger memory capacity on the reference storage nodes, which could reduce page faults on the memory-mapped LMDB file. On the other side, metadata is small, and it is unclear if the 4 GiB memory of the ARM nodes were fully utilized. Further investigation is needed to fully explain the difference in performance between the architectures.

GlusterFS showed the worst overall performance for all accesses that require writes to files' metadata. That is related to the fact that changes need to be done on every brick in the volume resulting in many network messages and a high metadata access load on all nodes. Deletes on GlusterFS even decreased for more clients added. Nevertheless, `stat`-calls did benefit from the distributed and duplicated metadata management, and it achieved higher IOPS than CephFS and OrangeFS on both architectures.

MooseFS showed similar behavior for creating files compared to CephFS and OrangeFS on the reference architecture. Though all metadata is held in memory, writing 1 KiB to the file enforces that a chunk is allocated. Both the allocation and the client's communication to the chunk server are included in the measured IOPS restricting the performance of MooseFS. In addition, operations that change metadata are written to an on-disk journal for disaster recovery. Creating directories does not need communication with the chunk servers because they are only stored in the memory of the master node. The high IOPS values in this case indicate that appending operations to the journal does not cause considerable overhead. In contrast to the other DFS, MooseFS does not directly access the data storage nodes for file deletions which are completely asynchronous or even delayed for trash times greater than zero. In general, MooseFS achieved superior performance compared to all systems, including the reference system, due to its architecture. Nevertheless, the price for the high performance is that the system is limited in the number of files and directories.

In conclusion, all systems behaved differently for the metadata-focused workload. Unlike MooseFS the other systems are not designed for fast small-file access. From the user's perspective `stat`-calls are essential for the interactivity of the system because directory listing will cause one call per entry in the directory. From that point of view, MooseFS and GlusterFS

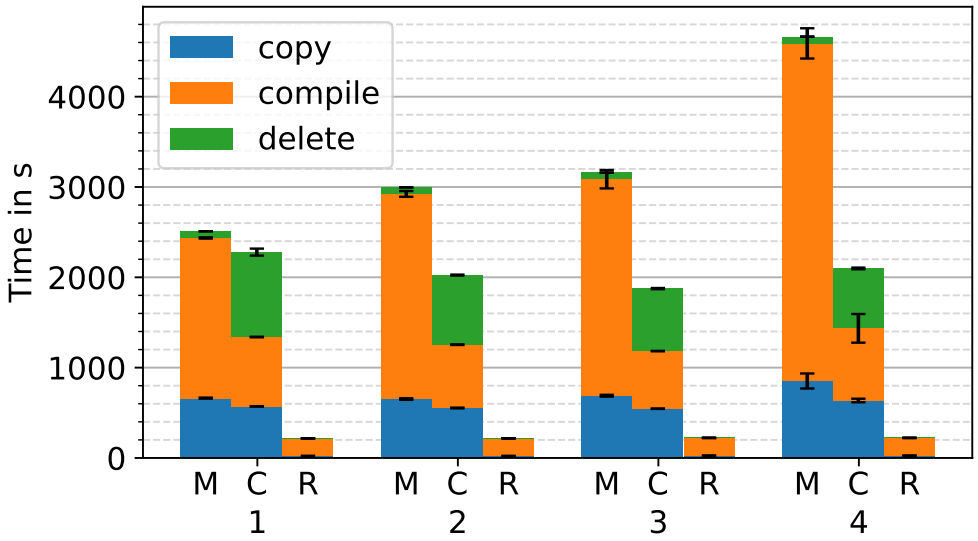


Figure 6.4.: Comparison of the duration of the different compile benchmark phases for MooseFS (M), CephFS (C), and on the local disks of the clients which are used as reference (R).

showed good performance comparable to or even exceeding the performance of the reference system.

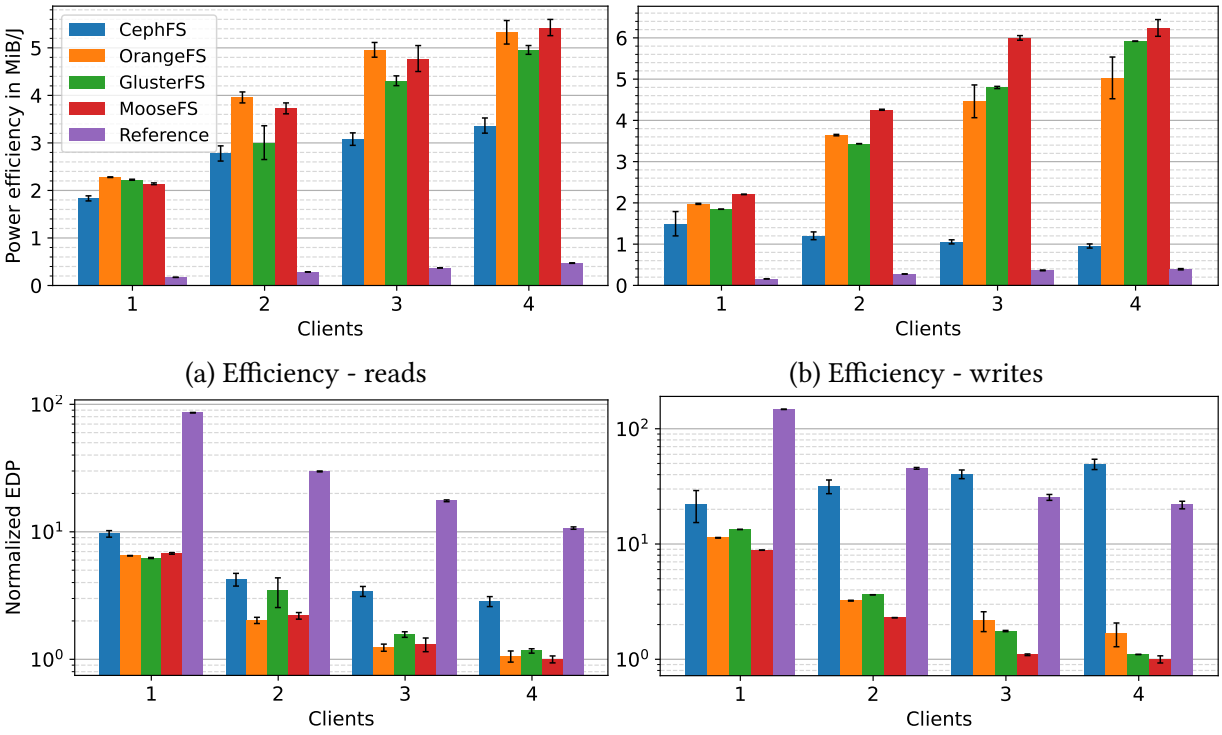
6.4.3. Source Code Compilation

The source code compilation benchmark was the most difficult one to complete on the systems. GlusterFS took about 3 hours, and OrangeFS even about 14 hours to complete the first iteration of the benchmark using one client. Therefore, these systems are not suitable for this type of workload and are excluded from further evaluation.

The results for MooseFS and CephFS are presented in Figure 6.4. As a reference, the local disks on the clients were used. In the best case, the duration of the different benchmark parts stays constant when adding more clients, which was approximately the case for CephFS. Most likely, CephFS benefits from its cache management via capabilities. Each client operated in its own directory that other clients did not access. Therefore, each newly created file could be cached on each node which is a considerable benefit for compiling code where each C translation unit results in one object file. As previously discussed for the metadata benchmark, lazy file deletions of CephFS can not be observed, and deleting the complete repository takes approximately the same time as the compile phase.

MooseFS, on the other side, shows an increasing compile time while the time needed to copy the repository stood nearly constant. Because deletes in MooseFS are asynchronous, the delete time remained constant, and deletes were fast.

The vast gap between the reference measurement and the DFS measurements shows that source code compilation should generally be done on the local storage of the nodes or should at least be aggressively cached.



(a) Efficiency - reads

(b) Efficiency - writes

(c) EDP - reads

(d) EDP - writes

Figure 6.5.: Energy-efficiency of the systems during the throughput oriented benchmark using the POSIX interface.

6.5. Energy Efficiency of the Parallel File Systems

This section will evaluate the energy efficiency for the benchmarks discussed above using the throughput and, respectively, I/O operation efficiency metric. This value was obtained by dividing the five samples of each data point by the mean power consumption of their run-time. The final value is the mean of these five efficiency samples, and the error bars show their standard deviation. The use of the energy-delay product as a metric for tuning storage systems will be explored using the throughput-oriented benchmark via the POSIX interface as an example. The EDP samples are obtained by integrating the power samples of each performance sample and multiplying them by the time that their iteration took. The final value is again obtained as the mean of these five samples, and the error bars show their standard deviation.

6.5.1. Parallel Sequential Data Access

The performance evaluation showed that access via MPI-I/O is not reasonable for its low throughput on all systems except for OrangeFS. Because of this, only the energy efficiency of the POSIX access will be evaluated in this section. Overall the plots of the energy efficiency shown in Figure 6.5 resemble the plots of the throughput, Figure 6.1, indicating that energy consumption is mainly determined by run-time. The exception is the reference system which consumes more than ten times the power of the ARM-based cluster even in an idle state. The performance, however, was comparable to the ARM-based cluster. Of course, this results in terrible energy efficiency on the reference cluster.

Even though the run-time mostly determines energy efficiency, there were differences in power consumption between the DFSs on the ARM-based cluster. That can be seen in Figure 6.2 which shows the distributions of the power samples during the last iteration for writing and reading with four clients.

As previously mentioned, CephFS had the lowest average power consumption of all systems. As discussed in Section 6.3 this does not implicate energy efficiency. CephFS also had the worst throughput performance resulting in long run times. Therefore, it was less efficient than the other systems on the ARM-based cluster, especially for writes. Nevertheless, due to the low power consumption of the hardware, it was still more efficient than the reference cluster.

OrangeFS and MooseFS had comparable energy efficiency for reads. MooseFS achieved the maximum read efficiency of all systems with four clients. The two systems were closely followed by GlusterFS, which reached comparable efficiency despite having the highest power consumption. The reason for the higher power consumption is most likely that the SSDs were exchanged for HDDs during the setup.

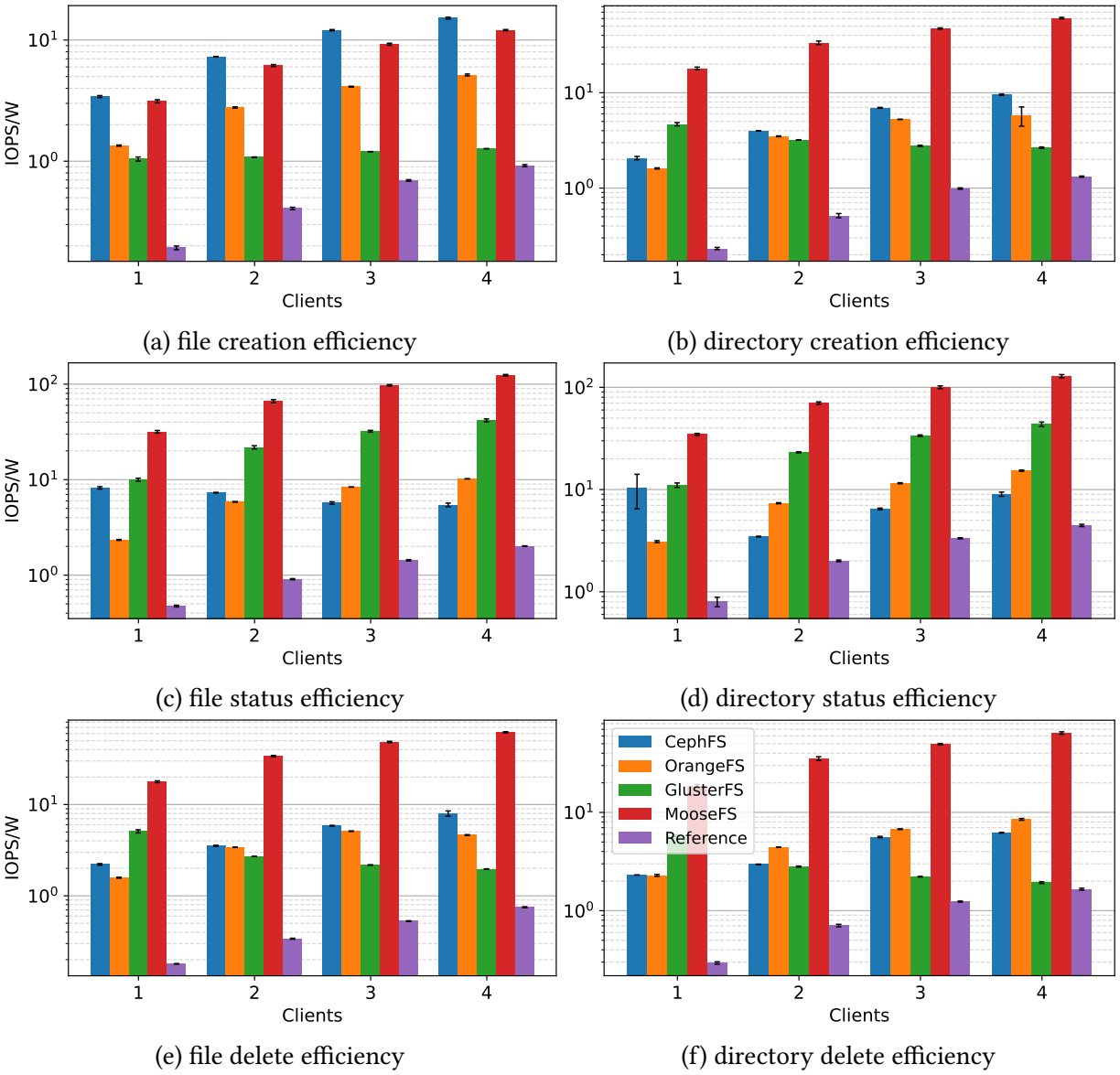
For writes, MooseFS had the best energy efficiency, followed by GlusterFS and OrangeFS. Surprisingly, GlusterFS came very close to the maximum achieved by MooseFS. However, the efficiency of GlusterFS is overestimated due to the measurement setup. This is because the erasure coding is done by the clients and its overhead on power consumption is therefore not measured.

The EDP is a fused metric that measures energy efficiency and performance at once. Because the weight of the time was set to one, the focus was put on efficiency. This property can be observed for writes, where CephFS reached a far better EDP than the reference system. Nevertheless, starting from two clients, the reference system got better than CephFS as the performance outweighed energy consumption. That shows precisely the property that would make the EDP a good metric for tuning storage systems. Changes that result in higher energy consumption are made less worthwhile. In practice, one would choose higher weights for the performance, especially for HPC systems.

The EDP favors systems that are both performant and energy-efficient. Therefore, GlusterFS, MooseFS, and OrangeFS achieved comparable values. MooseFS achieved the minimum with four clients for reading and writing.

One open problem of the EDP is that some results had a high standard deviation. The reason is that the time is taken into account multiple times. The total energy consumption is computed as the integral of the power consumption over time and henceforth is a function over the time bounds and the power function. By multiplying the time, the error introduced by it gets amplified. This effect is even worse when the time has a higher weight. In practice, that would result in many measurements needed to achieve a close guess of the true EDP.

Overall, the results show that the ARM-based cluster is an energy-efficient alternative to x86 clusters for data throughput-focused workloads. Against the first intuition, systems with a higher average power consumption had a better energy efficiency. The reason is that greater performance, most likely due to better hardware utilization, and henceforth shorter run-times outweighed the higher power consumption.



(a) file creation efficiency

(b) directory creation efficiency

(c) file status efficiency

(d) directory status efficiency

(e) file delete efficiency

(f) directory delete efficiency

Figure 6.6.: The energy-efficiency of file and directory creations.

6.5.2. Metadata Access

Only the energy efficiency in IOPS per Watt of the metadata benchmark is analyzed and compared to keep the scope of this thesis. Like the performance results of the metadata benchmark using the directory tree, the energy-efficiency results are also available in Appendix B.

Figure 6.6 shows the resulting energy efficiency. The relationships between the systems on the ARM-based cluster resemble their counterparts in the performance metric, which indicates that energy efficiency was again determined mainly by performance and resulting run times. The exception was the reference system which achieved below one I/O operation per Watt for most measurements on file and directory creations and deletes. MooseFS, in contrast, was able to achieve above ten file creations per Watt and approximately 60 directory creations per Watt. However, the reference system did not use an SSD for the metadata node. That would most likely have resulted in higher performance and slightly lower power consumption. The comparability is therefore limited in this case.

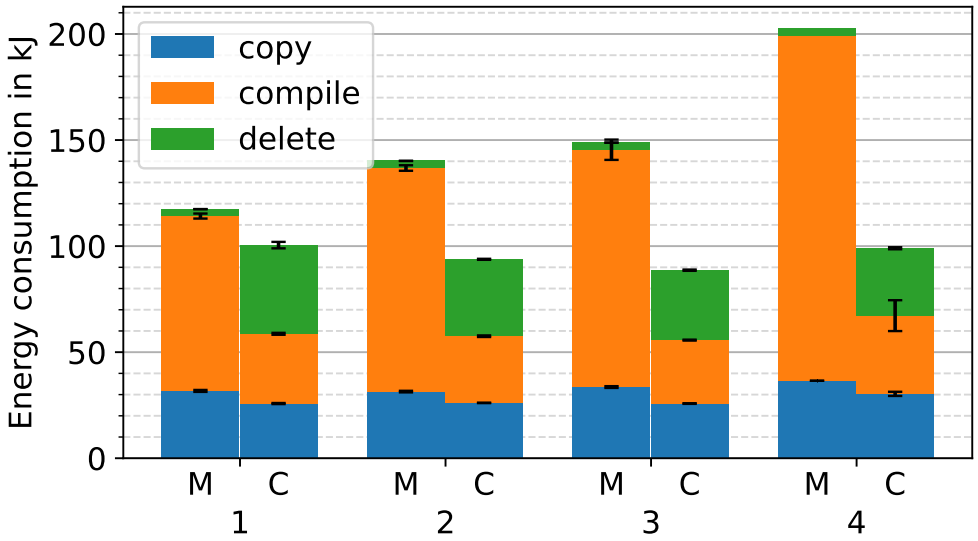


Figure 6.7.: Total energy consumption of the compile benchmark for MooseFS (M) and CephFS (C). Only one iteration could be measured for MooseFS with four clients due to an error with the power meter setup.

MooseFS had the highest energy efficiency in most cases. The exception was file creation, where CephFS was most efficient. Because file creations needed access to the data storage nodes for space allocation, they consumed up to one order of magnitude more energy than performing `stat`-calls. Even though directory creations on MooseFS also showed a lower energy efficiency compared to the `stat`-calls despite access to the chunk servers were not needed. The reason could be that the status of a file is only read from memory while creating a directory requires a write to the journal.

As deletes cause write access on the metadata node and release of allocated storage on the data nodes, they showed similar limitations as creations. The power efficiency of MooseFS was possibly under-estimated because allocated storage is asynchronously released, which means that even after the end of the measurement, deletes were still ongoing in the background.

6.5.3. Source Code Compilation

The energy efficiency of the compile benchmark was measured using the total energy consumption of the systems while performing the benchmark. The results are shown in Figure 6.7. Unfortunately, the BananaPi connected to the power meter had either a freeze or a clock reset during the second iteration of the benchmark for MooseFS with four clients resulting in a missing time span of about ten minutes. By the time this issue was detected, no repetition could be done. Because the exact error remained unclear, it is uncertain if the following iterations were still in synchronization with the timestamps generated by the benchmark script. Therefore, only the first iteration was used for MooseFS with four clients, and no error bar could be plotted.

As Figure 6.7 resembles Figure 6.4, the energy consumption is again determined mainly by the performance of the systems. Yet, the energy consumption of MooseFS is likely even higher than shown because it is not known if all asynchronous deletes are included in the following copy or compile phase. At least for the last iteration, this can not be the case.

As already concluded for this benchmark's performance, compiling source code is not a reasonable workload for the systems.

6.6. Design Recommendations for Future Single-Board Computer Storage Clusters

Overall, the performance and energy-efficiency results on the ARM-based cluster were comparable to or exceeded the reference cluster. All DFSs that were tested proved useful for some workloads. While CephFS performed the worst in the synthetic benchmarks, it performed best for the source code compilation. OrangeFS was the only system that reached a reasonable throughput using MPI-I/O. GlusterFS provided high data safety due to erasure coding while preserving high data throughput via the POSIX interface. In terms of metadata performance and data throughput via the POSIX interface, MooseFS showed the best result. Yet, no DFS performed best for all workloads, and the DFS for a productive storage cluster should always be chosen with respect to its intended use case. The negative impact of replication on CephFS or the overlapping delete operations on MooseFS shows that contention for disk access leads to reduced performance. Consequently, the ARM-based cluster should be accessed in a coordinated manner. More detailed studies on ARM-based clusters with more nodes should be done to evaluate the scaling behavior of the system. The hope is that more nodes and consequently more disks allow for more performance for overlapping different workloads.

The performance and the resulting run-time mostly determined energy efficiency in all cases. The low average power consumption of CephFS suggested low hardware utilization, probably due to idle disks. The low performance of CephFS further supported this. MooseFS or GlusterFS, on the other side, had the highest average power consumption and achieved both high performance and energy efficiency.

To conclude, the results indicate that saving energy by building storage clusters from single-board computers is possible. They are a viable alternative to traditional clusters based on x86 machines, especially when focusing on data throughput. The low-price SBCs also allow easy and fine-grained capacity scaling by adding a few nodes when needed. Nevertheless, for productive usage, some improvements to the setup are needed. Some setup recommendations for future storage clusters are discussed in the following paragraphs. They are based on the experiences made with the current cluster.

Memory While MooseFS could provide good metadata performance by keeping all metadata in memory, this approach also has disadvantages. First, the number of files and directories is limited by the memory capacity of the metadata node. An obvious solution is to equip the metadata node with more memory. But not only the metadata node would benefit from adding more memory. Data storage nodes with more memory could provide larger caches and, therefore, higher performance.

The second disadvantage of MooseFS holding all metadata in memory is that memory errors are common and will happen regularly during the lifetime of the storage cluster [70]. For performance reasons, MooseFS does no checks on metadata [39]. In the best case, the system will crash and can recover using the on-disk journal and backup files. However, in the worst case, data is silently corrupted, resulting in missing files or directories. Depending on the

use case of the DFS, e.g., the safety requirements for the data, this is not acceptable. The use of *Error Correction Code* (ECC) memory is therefore highly recommended by the MooseFS documentation [39]. Though this problem is severe for the MooseFS master server, the chunk servers and other DFSs are also affected. All storage nodes could read corrupted data from their memory and send it to a client resulting in further errors. Unfortunately, there are no SBCs that feature ECC memory, as far as known by the author.

Hybrid Clusters As explained above, the metadata in MooseFS is prone to silent corruption. One way to mitigate this would be to use a regular server with ECC memory as the MooseFS master server while the chunks servers could still reside on SBCs. As the master server has a single-threaded implementation, the system could even benefit from regular processors' higher single-core compute power. However, among the analyzed DFSs, this approach is only useful for MooseFS and does not solve the problem with memory errors in chunk servers.

Storage Devices Though approaches like erasure coding between nodes proved useful, not all systems implement such techniques. Regular replication imposed strict performance limitations that are not acceptable for using the SBC cluster in practice. Software RAID on the node level seems like the most reasonable approach to guaranteeing a certain data safety level. However, only mirroring can be done on the Odroid HC4 as only two disks can be connected. The result is that half of the storage capacity is lost, and there are high costs per usable tebibyte. More efficient RAID levels like RAID 5 or RAID 6 would require at least three respectively four storage devices attached to the machine. Additionally, those RAID levels would provide a higher data rate for writes compared to mirroring. Therefore, SBCs used for further storage clusters should feature more disk connectors, e.g., more SATA connectors. That leads directly to the next recommendation.

Network The network throughput needs to be improved to saturate the bandwidth provided by greater amounts of disks. Some SBCs already feature faster NICs. One example is the Clearfog A388 Base [50] which features multiple NICs, one of which offers 2.5 Gbit/s. However, this device has only up to 2 GB memory capacity and no native SATA connectors.

Chapter 7.

Related Work

Surveys of Distributed File Systems Early works developed a taxonomy on distributed file systems [69, 82] and used it to describe several distributed file systems, including NFS, AFS, and Apollo. While the developed taxonomy is still applicable, some surveyed file systems are no longer actively maintained.

Newer surveys [19] provide a good overview of currently popular file systems like MooseFS, CephFS, and Lustre. A comprehensive survey of storage solutions, in general, is given in [21], which is based on [52].

However, this thesis includes its own survey of the distributed file systems under analysis to provide a solid and up-to-date foundation for later discussions.

Energy-Efficiency of existing Storage Clusters There have been various efforts to reduce the energy consumption of existing storage clusters. One approach to reduce the energy consumption of storage clusters is to reduce the amount of data that needs to be stored in the first place. Therefore, Costa et al. [17] evaluated the influence of data deduplication on the energy consumption of storage systems. Data deduplication was described as a trade-off between increasing energy consumption due to computational overhead and decreasing energy consumption due to lower storage sizes and reduced I/O pressure. They found that deduplication reduced energy consumption while writing a 256 MB file starting from 18% of data similarity. However, from the performance perspective, more than 40% data similarity was needed to observe benefits.

Other techniques are based on the idea of sending idle peers to sleep [45]. Zhang et al. [89] proposed a system that is partitioned in a hot and cold storage node-set based on the assumption that 80% of the data accesses go to 20% of the storage space. Hot nodes store replicated data that is frequently accessed, whereas cold nodes store only occasionally accessed data. Cold nodes would therefore stay in a low-power sleep mode most of the time to save energy. Another approach is to assign subsets of storage clusters to specific users and only run them at full power when said user uses the attached compute cluster [36]. This method was validated using a simulation model with real-world workloads traced at the HPC cluster at the University of Connecticut as input. The conclusion was that this approach could be used on systems that only have a few active users.

In contrast to sending complete storage nodes to sleep, Ruan et al. [65] studied the effect of spinning down idle disks. To create larger idle times, they used buffer disks on each storage node of the cluster. I/O requests were directed to the buffer disk if possible so that the primary data disks could be spun down most of the time. Evaluation of their prototype of an *energy-efficient*

cluster storage system (ECOS) showed that depending on the workload and ECOS-specific parameters, significant energy-efficiency improvements could be achieved.

Considering that local file systems are often part of the storage stack, their influence on energy efficiency and performance were analyzed in [71] using simulated workloads of web, database, and file servers. It was found that the choice of file system and its configuration greatly influence performance and energy efficiency. However, no file system performed best for all workloads.

Energy-Efficiency due to Low-Power Hardware Apart from energy-saving efforts on existing storage systems, there also have been evaluations of different design approaches to storage systems. Kougkas et al. proposed to build storage clusters based on the *Open Ethernet Drive* (OED) [38] which pairs a storage device with a built-in computer system similar to a single-board computer. Though initially developed for cloud environments, they evaluated the OED architecture for usage in HPC. They ran benchmarks to measure the performance of the CPU, memory, disk, and network capabilities of the OED device by HGST. They found that the HGST OED provided high performance in I/O but lacked performance for computations.

Gudu and Hardt evaluated the use of an ARM-based Ceph cluster, made of Cubieboards, as a replacement for traditional NAS controllers [29]. They measured the throughput of their cluster at RADOS and *RADOS Block Device* (RBD) levels. They found that the Cubieboard cluster is a viable alternative to NAS controllers. However, the limited network capabilities were the bottleneck of the system.

Those approaches are similar to the approach pursued in this work. However, as far as the author knows, no OED devices have been released since the original release of the HGST OED and Seagate Kinect. Apart from the fact that no such hardware is currently available, this approach would most likely suffer from lacking data safety features that need to be implemented using replication or erasure coding. While Gudu and Hardt focused on ARM-based SBC clusters as a replacement for home NAS systems, this thesis analyzes their use for traditional HPC storage, and measurements were done on the file system level.

Chapter 8.

Conclusion and Future Work

Conclusion Storage clusters are steadily scaled up to satisfy increasing demands on capacity and performance. Like it is already the case for compute clusters, energy consumption can be expected to become a limiting factor. Increasing the energy efficiency of storage systems would reduce costs and enable further scaling of the systems. One possibility of solving this problem for compute clusters was using ARM-based processors instead of traditional x86-based processors, which have better energy-proportionality. Another possibility that was analyzed was to use single-board computers. In this thesis, this approach was applied to storage clusters as well. The goal was to answer the question of whether ARM-based single-board computers can be used to build storage clusters that achieve better energy efficiency than typical x86-based storage clusters while providing comparable performance.

An ARM-based test cluster was built and evaluated for its energy efficiency and performance to answer this question. Different distributed file systems were chosen for evaluation on the test cluster. While CephFS, OrangeFS, GlusterFS, and MooseFS had a straightforward setup on the unusual hardware BeeGFS and JULEA could not be evaluated due to technical issues. Different workloads were run on the four remaining file systems. As a reference, OrangeFS on a comparable x86-based setup was used. It was found that all systems were useful for some workloads. Using the POSIX interface, MooseFS showed overall the best results if care was taken that delayed delete operations did not overlap with ongoing I/O. For access via MPI-I/O, only OrangeFS was suitable. For metadata accesses, the systems showed diverse behavior. MooseFS had the best performance again, but scaling was limited to the memory capacity of the master node. The real-world workload generated by compiling the Linux kernel put the most stress on the systems, and only MooseFS and CephFS were able to complete the benchmark in a reasonable amount of time. Results showed that none of the evaluated systems was suitable for this workload.

The ARM-based cluster was superior to the x86-based cluster in terms of energy efficiency. Comparison between the different file systems on the ARM-based cluster showed that energy efficiency was determined mainly by the run-times of benchmarks and henceforth the achieved performance. Systems that showed a lower performance had a lower average power consumption, hinting that the hardware, especially the storage devices, were not fully utilized.

So to answer the research question of the thesis, the ARM-based storage cluster proved to be a viable alternative to traditional x86-based storage clusters, especially for workloads that focus on data throughput. However, based on the experiences gained with the Odroid HC4, future single-board computer storage nodes should have a higher network throughput, more memory, and should be able to provide access to more storage devices.

Future Work This thesis opens several opportunities for future work. First, using the energy-delay product for tuning a distributed file system should be further evaluated. Especially the choice of the performance weight is of interest.

Second, further work should be done on the different distributed file systems. Because erasure coding showed promising results on GlusterFS, the erasure coding feature of CephFS should be evaluated as well. This feature could improve the performance and energy efficiency of CephFS. OrangeFS had a lower metadata performance on the ARM-based cluster than on the x86-based cluster despite achieving higher IOPS on its disk. The reason for this could give insight into the differences between the two setups and should be investigated. The promising DFSs should be further evaluated using real-world scientific applications using different high-level interfaces like HDF5 or NetCDF.

At last, some more evaluations of single-board computers as storage nodes are needed for practical deployments. Of particular interest is if the computing power of the nodes is sufficient for features like compression, data deduplication, or encryption. Furthermore, the scaling behavior for more storage nodes should be analyzed. Evaluation of the different design recommendations proposed at the end of Chapter 6 is also needed.

Appendix A.

Benchmark Parameters

```
time_based=1
runtime=600
group_reporting=1
bs=4M
```

```
[mem_read]
new_group=0
include throughput.fio
size=256M
directory=/tmp
rw=read
numjobs=4
```

```
[mem_write]
new_group=0
include throughput.fio
size=256M
directory=/tmp
rw=write
numjobs=4
```

Figure A.1.: *fio* parameters that were used to measure the main memory data throughput of the clients. The first file is the one included in the two following ones.

```
runtime=600
time_based=1
group_reporting=1
directory=/tmp
size=128M
bs=1
numjobs=8
```

```
[iops_read_ram]
new_group=0
include iops_ram.fio
rw=randread
```

```
[iops_write_ram]
new_group=0
include iops_ram.fio
rw=randwrite
```

Figure A.2.: *fio* parameters that were used to measure the main memory I/O operation throughput of the clients. The first file is the one included in the two following ones.

```
time_based=1
runtime=600
group_reporting=1
bs=4M
```

```
[global]
direct=1
size=16G
ioengine=libaio
iodepth=32
include throughput.fio


[bw_write_1]
new_group=0
filename=/dev/sda
rw=read

[bw_write_2]
new_group=1
filename=/dev/sdb
rw=read
```

```
[global]
direct=1
size=16G
ioengine=libaio
iodepth=32
include throughput.fio

[bw_write_1]
new_group=0
filename=/dev/sda
rw=write

[bw_write_2]
new_group=1
filename=/dev/sdb
rw=write
```

Figure A.3.: *fio* parameters that were used to measure the disk data throughput on the Odroid HC4. The first file is the one included in the two following ones.

```
runtime=600
time_based=1
group_reporting=1
direct=1
ioengine=libaio
iodepth=32
size=1G
bs=4k
numjobs=2
```

```
[iops_read_1]
new_group=0
include iops_blockdevice.fio
filename=/dev/sda
rw=randread

[iops_read_2]
new_group=1
include iops_blockdevice.fio
filename=/dev/sdb
rw=randread
```

```
[iops_write_1]
new_group=0
include iops_blockdevice.fio
filename=/dev/sda
rw=randwrite

[iops_write_2]
new_group=1
include iops_blockdevice.fio
filename=/dev/sdb
rw=randwrite
```

Figure A.4.: *fio* parameters that were used to measure the disk I/O operation throughput on the Odroid HC4. The first file is the one included in the two following ones.

```
    ...
    let files_per_host_per_dir=1200/$i
    mdtestargs="-I ${files_per_host_per_dir} -b 2 -z 4 -i 5 -N 1
       ↪ -w 1024 -e 1024 -p 5"
    ...
```

```
    ...
    let files_per_host=50000/$i
    mdtestargs="-n ${files_per_host} -b 1 -z 1 -i 5 -N 1 -w 1024
       ↪ -e 1024 -p 5"
    ...
```

Figure A.5.: The parameters for *mdtest*. The upper file describes the directory tree benchmark while the second one describes the large directory benchmark.

```
IOR START
    testFile=IOR_throughput
    repetitions=5
    interTestDelay=5
    readFile=1
    writeFile=1
    #filePerProc=1
    memoryPerNode=85%
    verbose=1
    #api=POSIX
    #fsync=1
    api=MPIIO
    segmentCount=1
    transferSize=4M
    reorderTasksConstant=1
    blockSize=36G
    numTasks=1
RUN
    blockSize=18G
    numTasks=2
RUN
    blockSize=12G
    numTasks=3
RUN
    blockSize=9G
    numTasks=4
RUN
IOR STOP
```

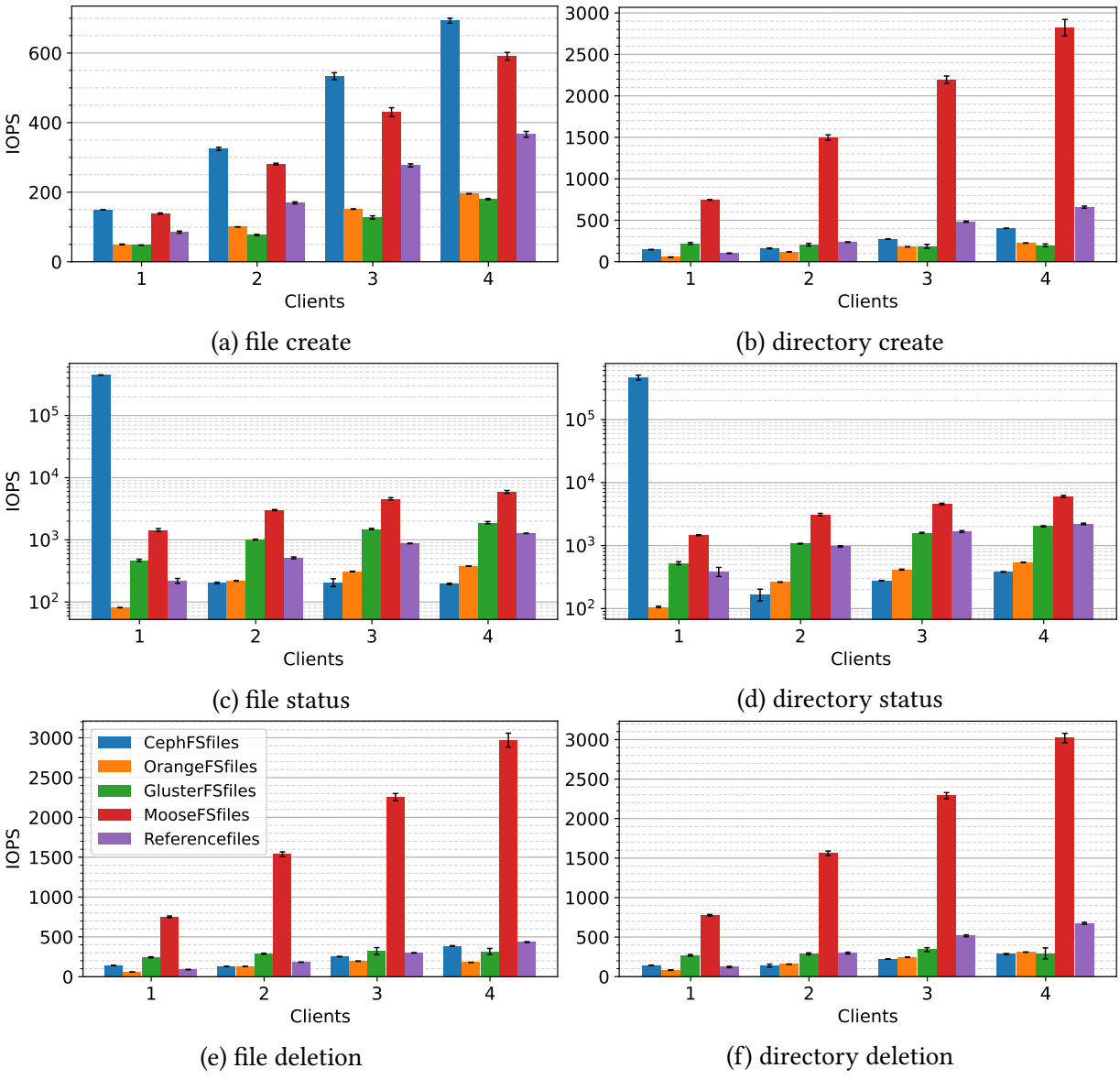
Figure A.6.: The parameters for *IOR*. While this is the MPI-I/O version, the gray comment lines show the additional parameters for the POSIX benchmark.

```
# [...] clone repository and install dependencies on each host
for clients in {1..4}; do
    for iteration in {1..5}; do
        # copy
        t1=$(bc -l <<< "$(date +%s%N)/(10^9)")
        procs=()
        for host in ${hosts[@]:0:${clients}}; do
            ssh ${host} "cp -r linux ${curr_dir}/linux-${host}" >
               ↪ /dev/null 2>&1 &
            procs+=("$!")
        done
        wait ${procs[@]}

        # compile
        t2=$(bc -l <<< "$(date +%s%N)/(10^9)")
        procs=()
        for host in ${hosts[@]:0:${clients}}; do
            ssh ${host} "cd ${curr_dir}/linux-${host}; make
               ↪ defconfig; make -j4" > /dev/null 2>&1 &
            procs+=("$!")
        done
        wait ${procs[@]}

        # delete
        t3=$(bc -l <<< "$(date +%s%N)/(10^9)")
        procs=()
        for host in ${hosts[@]:0:${clients}}; do
            ssh ${host} "rm -rf ${curr_dir}/linux-${host}" >
               ↪ /dev/null 2>&1 &
            procs+=("$!")
        done
        wait ${procs[@]}
        t4=$(bc -l <<< "$(date +%s%N)/(10^9)")
        # [...] copy=t2-t1 compile=t3-t2 rm=t4-t3 and save
           ↪ results to csv file
    done
done

```

Figure A.7.: A shortened version of the script for the source code compilation benchmark. The start of each phase and its duration is stored in CSV format.

Appendix B.

Additional Measurements



(a) file create

(b) directory create

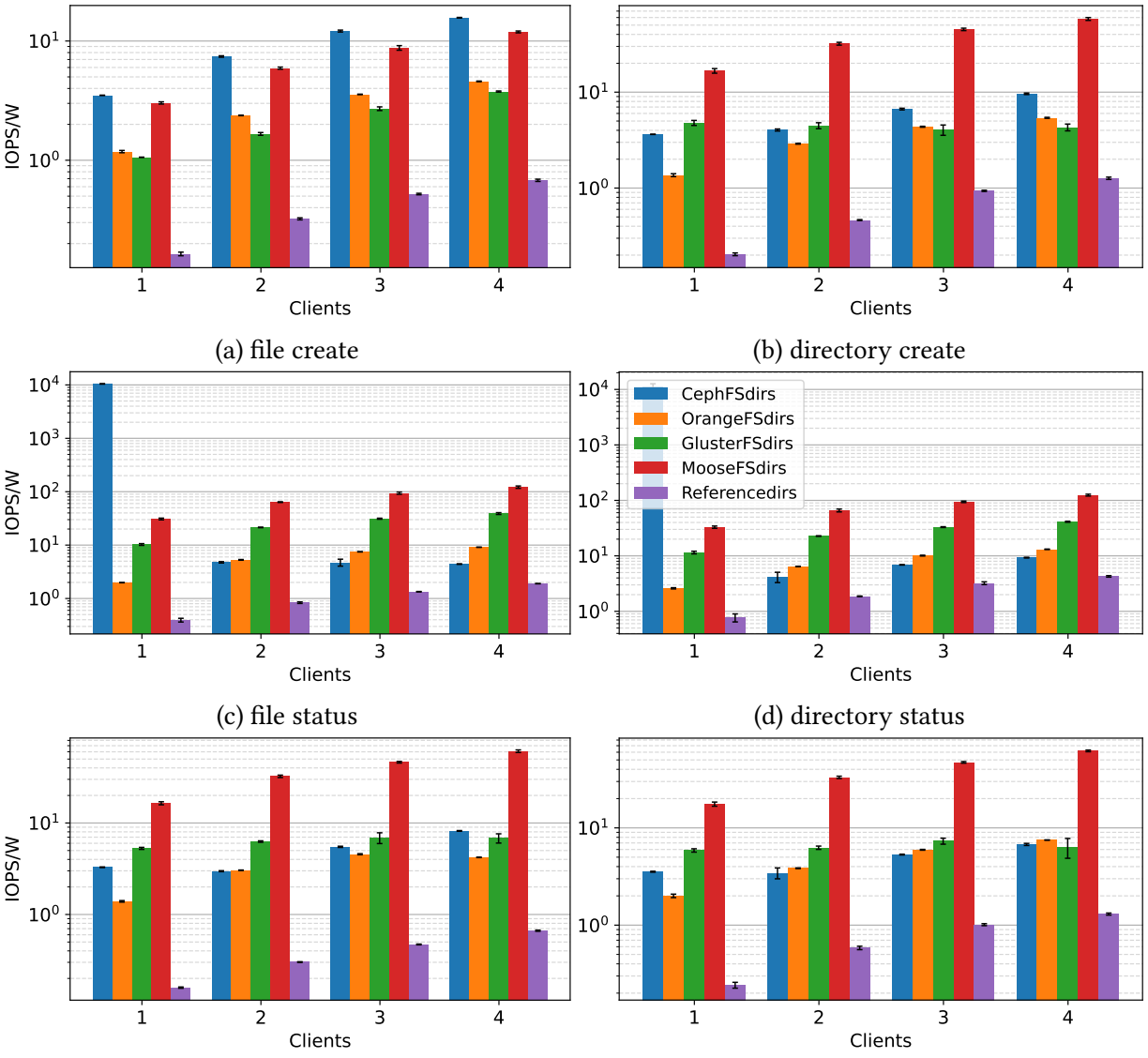
(c) file status

(d) directory status

(e) file deletion

(f) directory deletion

Figure B.1.: Comparison of the metadata performance measured in IOPS for different tasks on a directory tree as described in Chapter 6.

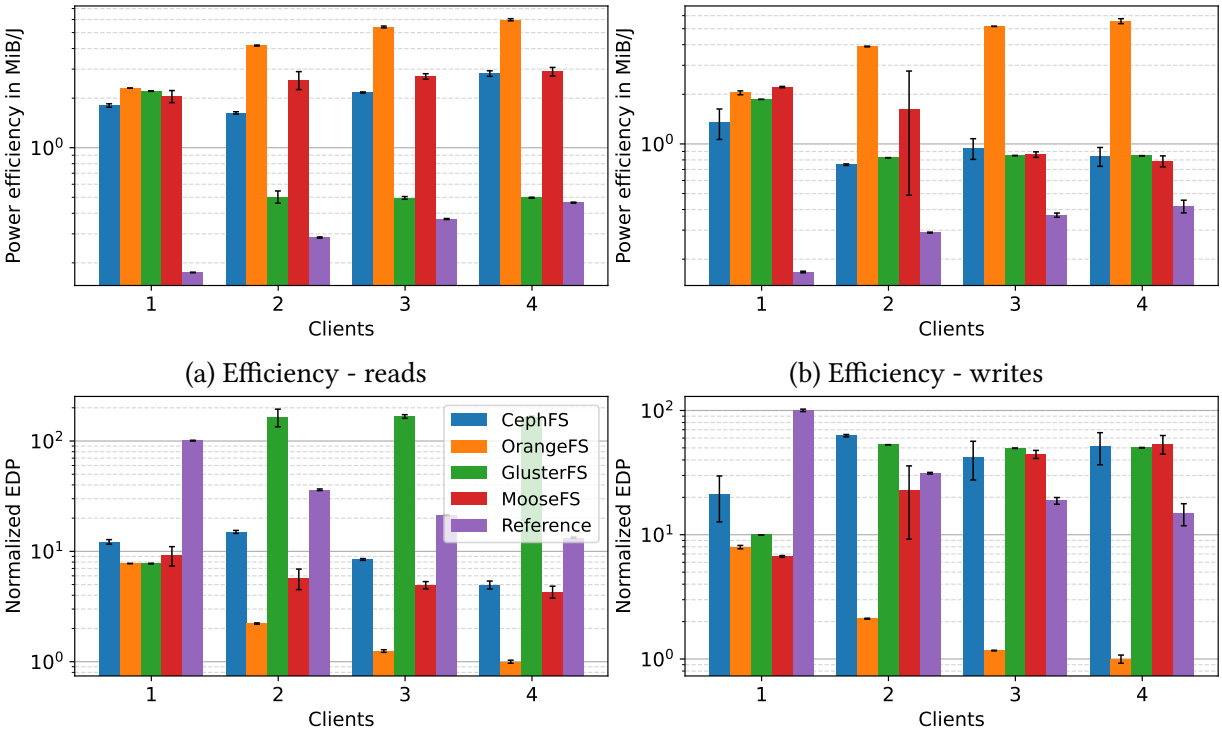


(a) file create (b) directory create

(c) file status (d) directory status

(e) file deletion (f) directory deletion

Figure B.2.: Comparison of the energy-efficiency of metadata operations measured in IOPS per Watt on a directory tree as described in Chapter 6.



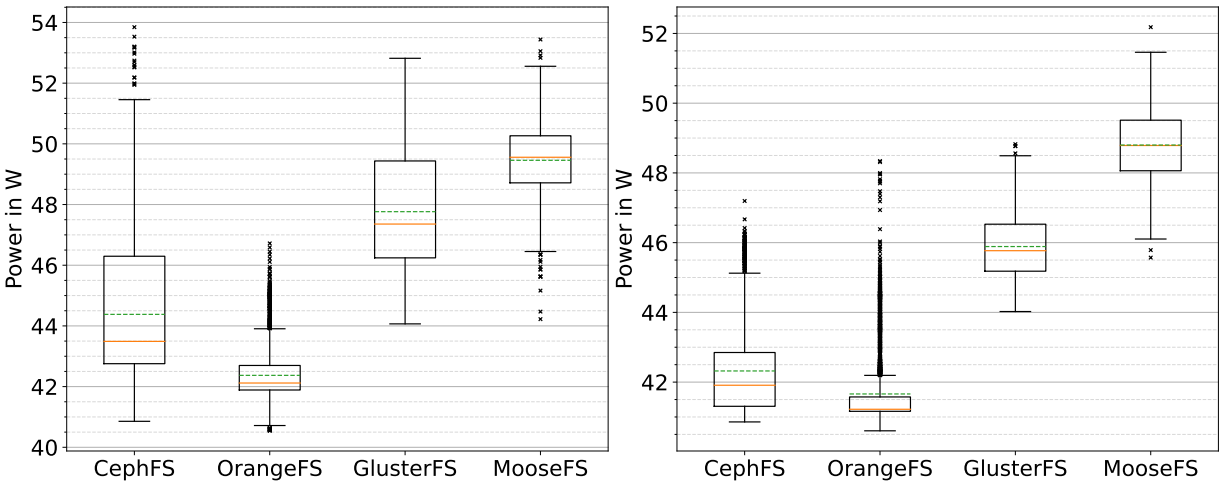
(a) Efficiency - reads

(b) Efficiency - writes

(c) EDP - reads

(d) EDP - writes

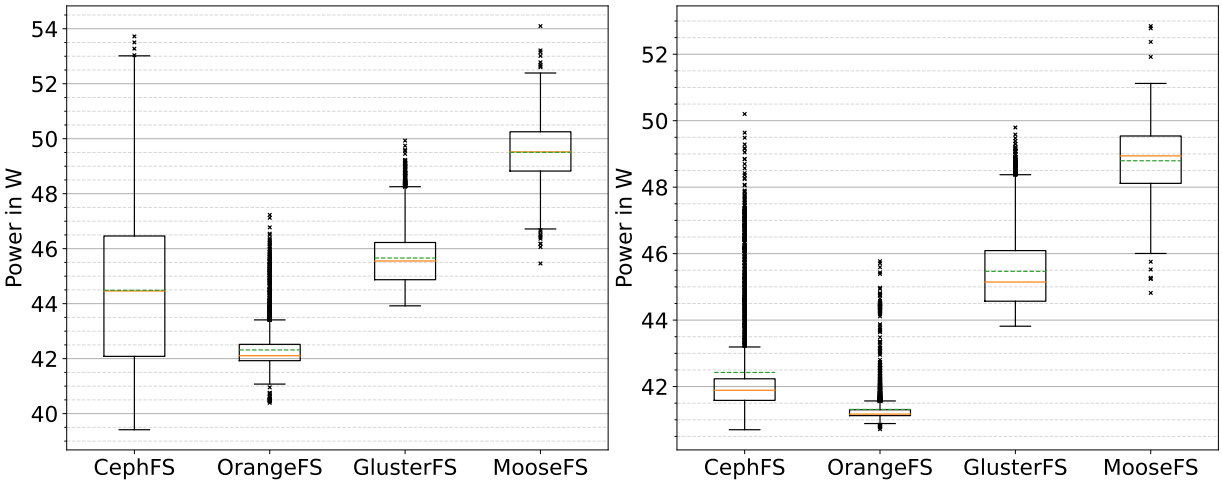
Figure B.3.: Energy-efficiency of the systems during the throughput oriented benchmark using the MPI-I/O interface.



(a) file creates

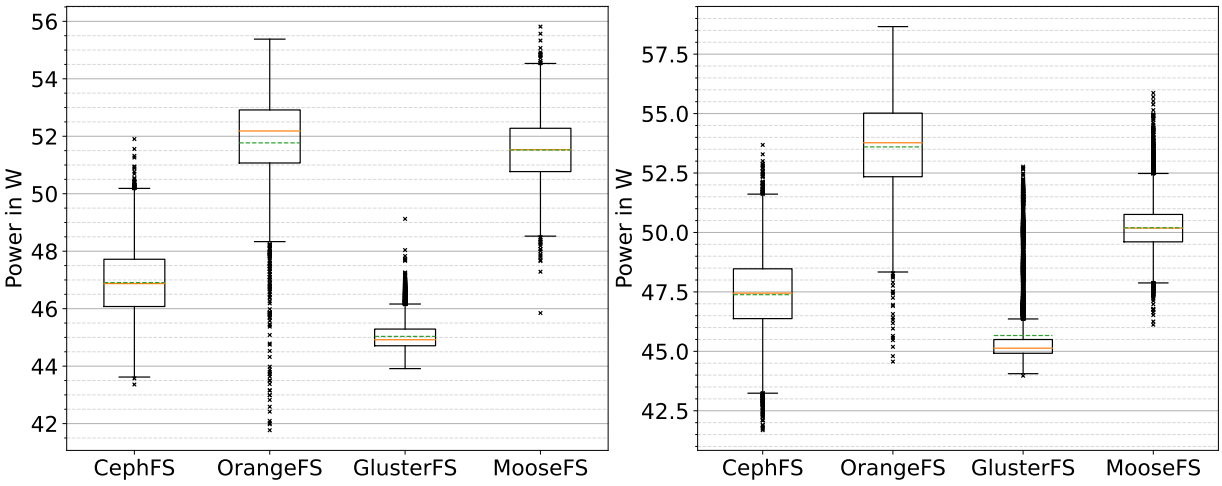
(b) directory creates

Figure B.4.: Comparison of the distributions of the power samples measured during the last iteration with four clients for creates of files and directories on the directory tree. The orange lines denote the median and the dotted green ones the mean of the distributions.



(a) file create (b) directory create

Figure B.5.: Comparison of the distributions of the power samples measured during the last iteration with four clients for creates of files and directories on the big directories. The orange lines denote the median and the dotted green ones the mean of the distributions.



(a) MPI-I/O reads (b) MPI-I/O writes

Figure B.6.: Comparison of the distributions of the power samples measured during the last iteration with four clients for writing and reading via the MPI-I/O interface. The orange lines denote the median and the dotted green ones the mean of the distributions.

Bibliography

[1] Ping(8) - Linux man page. https://linux.die.net/man/8/ping. [retrieved: 09.05.2022]. (Cited on page 13)

[2] Stat(2) - Linux manual page. https://www.man7.org/linux/man-pages/man2/stat.2.html. [retrieved: 03.05.2022]. (Cited on page 14)

[3] Pekka Abrahamsson, Sven Helmer, Nattakarn Phaphoom, Lorenzo Nicolodi, Nick Preda, Lorenzo Miori, Matteo Angriman, Juha Rikkila, Xiaofeng Wang, Karim Hamily, and Sara Bugoloni. Affordable and Energy-Efficient Cloud Computing Clusters: The Bolzano Raspberry Pi Cloud Cluster Experiment. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, pages 170–175, Bristol, United Kingdom, December 2013. IEEE. http://ieeexplore.ieee.org/document/6735414/. (Cited on pages 7 and 13)

[4] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. The case for custom storage backends in distributed storage systems. *ACM Transactions on Storage*, 16(2), May 2020. https://doi.org/10.1145/3386362. (Cited on page 15)

[5] Armbian. Armbian Odroid HC4. https://www.armbian.com/odroid-hc4/, 2022. [retrieved: 12.05.2022]. (Cited on page 28)

[6] Jens Axboe. Fio Documentation. https://fio.readthedocs.io/en/latest/fio_doc.html, 2017. [retrieved: 28.04.2022]. (Cited on page 13)

[7] S. B. Balaji, M. Nikhil Krishnan, Myna Vajha, Vinayak Ramkumar, Birenjith Sasidharan, and P. Vijay Kumar. Erasure coding for distributed storage: An overview. *Science China Information Sciences*, 61(10):100301, September 2018. https://doi.org/10.1007/s11432-018-9482-6. (Cited on pages 17 and 35)

[8] Zhang Baojun, Pan Ruifang, and Ye Fujun. Analyzing and improving load balancing algorithm of MooseFS. *International Journal of Grid and Distributed Computing*, 7(4):169–176, August 2014. https://doi.org/10.14257%2Fijgdc.2014.7.4.16. (Cited on page 22)

[9] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007. (Cited on page 7)

[10] BIPAI KEJI CPA.,LTD and Banana PI team Open source community. BananaPi M1 Hardware Specification. http://www.banana-pi.com/eacp_view.asp?id=35, 2014. [retrieved: 12.05.2022]. (Cited on page 27)

[11] Michael Moore David Bonnie, Becky Ligon, Mike Marshall, Walt Ligon, Nicholas Mills, Elaine Quarles Sam Sampson, Shuangyang Yang, and Boyd Wilson. OrangeFS: Advancing PVFS. In *USENIX Conference on File and Storage Technologies (FAST)*, 2011. (Cited on pages 10 and 17)

[12] Ceph authors and contributors. Ceph Documentation. https://docs.ceph.com/en/latest, 2021. [retrieved: 12.05.2022]. (Cited on pages 15, 16, 44, and 48)

[13] Doron Chen, Ealan Henis, Ronen I. Kat, Dmitry Sotnikov, Cinzia Cappiello, Alexandre Mello Ferreira, Barbara Pernici, Monica Vitali, Tao Jiang, Jia Liu, and Alexander Kipp. Usage centric green performance indicators. *SIGMETRICS Perform. Evaluation Rev.*, 39(3):92–96, 2011. https://doi.org/10.1145/2160803.2160868. (Cited on page 43)

[14] P.M. Chen and D.A. Patterson. Storage performance-metrics and benchmarks. *Proceedings of the IEEE*, 81(8):1151–1165, 1993. (Cited on page 42)

[15] Intel Corporation. DAOS Storage and High Performance Computing (HPC) Solutions. https://www.intel.com/content/www/us/en/high-performance-computing/daos.html. [retrieved: 09.05.2022]. (Cited on page 12)

[16] Western Digital Corporation. WD2502ABYS Datasheet. https://products.wdc.com/library/SpecSheet/ENG/2879-701281.pdf?_ga=2.204934934.742886585.1651008103-1759292557.1651008103, 2008. [retrieved: 26.04.2022]. (Cited on page 28)

[17] Lauro Beltrão Costa, Samer Al-Kiswany, Raquel Vigolvino Lopes, and Matei Ripeanu. Assessing data deduplication trade-offs from an energy and performance perspective. In *2011 International Green Computing Conference and Workshops*, pages 1–6, 2011. (Cited on page 56)

[18] Dell Inc. Dell Precision 3650 Tower Hardware Specification. https://www.delltechnologies.com/asset/en-us/products/workstations/technical-support/precision-3650-spec-sheet.pdf, 2021. [retrieved: 12.05.2022]. (Cited on page 27)

[19] Benjamin Depardon, Gaël Le Mahec, and Cyril Séguin. Analysis of Six Distributed File Systems. Research Report, February 2013. https://hal.inria.fr/hal-00789086. (Cited on page 56)

[20] Jon Dugan. Iperf(1) - Linux man page. https://linux.die.net/man/1/iperf. [retrieved: 09.05.2022]. (Cited on page 13)

[21] Kira Duwe and Michael Kuhn. Deliverable D1: Survey - Coupled Storage System for Efficient Management of Self-Describing Data Formats (CoSEMoS), 2021. (Cited on pages 12 and 56)

[22] Jake Edge. The OrangeFS distributed filesystem. https://lwn.net/Articles/643165/, 2015. [retrieved: 12.05.2022]. (Cited on page 19)

[23] Roy T. Fielding. *Architectural Styles and the Design of Network -Based Software Architectures*. PhD thesis, 2000. https://www.proquest.com/dissertations-theses/architectural-styles-design-network-based/docview/304591392/se-2?accountid=104761. (Cited on page 15)

[24] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. The Spack package manager: Bringing order to HPC software chaos. In *SC15: International Conference for High-Performance Computing, Networking, Storage and Analysis*, pages 1–12, Los Alamitos, CA, USA, November 2015. IEEE Computer Society. https://doi.ieeecomputersociety.org/10.1145/2807591.2807623. (Cited on page 29)

[25] Stefanos Georgiou, Maria Kechagia, Panos Louridas, and Diomidis Spinellis. What Are Your Programming Language's Energy-Delay Implications? In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 303–313, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3196398.3196414. (Cited on page 43)

[26] Markus Geveler, Balthasar Reuter, Vadym Aizinger, Dominik Göddeke, and Stefan Turek. Energy efficiency of the simulation of three-dimensional coastal ocean circulation on modern commodity and mobile processors. *Comput. Sci. Res. Dev.*, 31(4):225–234, 2016. https://doi.org/10.1007/s00450-016-0324-5. (Cited on page 43)

[27] J. Goodacre and A.N. Sloss. Parallelism and the ARM instruction set architecture. *Computer*, 38(7):42–50, 2005. (Cited on page 13)

[28] Xiaoyan Gu, Rui Hou, Ke Zhang, Lixin Zhang, and Weiping Wang. Application-driven energy-efficient architecture explorations for big data. In *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*, ASBD '11, pages 34–40, New York, NY, USA, 2011. Association for Computing Machinery. https://doi.org/10.1145/2377978.2377984. (Cited on page 7)

[29] Diana Gudu and Marcus Hardt. ARM Cluster for Performant and Energy-Efficient Storage. In Jörg Lässig, Kristian Kersting, and Katharina Morik, editors, *Computational Sustainability*, volume 645 of *Studies in Computational Intelligence*, pages 265–276. Springer, 2016. https://doi.org/10.1007/978-3-319-31858-5_12. (Cited on page 57)

[30] HARDKERNEL CO., LTD. Odroid HC4 Datasheet. https://wiki.odroid.com/odroid-hc4/hardware/hardware, 2021. [retrieved: 12.05.2022]. (Cited on pages 13 and 27)

[31] Martin Hedenfalk. LMDB: Lightning Memory-Mapped Database Manager (LMDB). http://www.lmdb.tech/doc/, 2015. [retrieved: 09.05.2022]. (Cited on page 47)

[32] Austin Hemmelgarn. Answer to "Understanding mmap". https://unix.stackexchange.com/a/389131, August 2017. (Cited on page 47)

[33] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Proceedings of 1994 IEEE Symposium on Low Power Electronics*, pages 8–11, 1994. (Cited on page 43)

[34] SanDisk International. SanDisk Extreme® microSD™ UHS-I Memory Card Datasheet. https://www.westerndigital.com/products/memory-cards/sandisk-extreme-uhs-i-for-mobile-gaming-microsd. [retrieved: 18.04.2022]. (Cited on page 27)

[35] Samuel Just. Crimson: A new ceph OSD for the age of persistent memory and fast NVMe storage, February 2020. (Cited on page 44)

[36] Cengiz Karakoyunlu and John A. Chandy. Techniques for an energy aware parallel file system. In *2012 International Green Computing Conference, IGCC 2012, San Jose, CA, USA, June 4-8, 2012*, pages 1–5. IEEE Computer Society, 2012. https://doi.org/10.1109/IGCC.2012.6322247. (Cited on page 56)

[37] Sandeep Koranne. Hierarchical data format 5 : HDF5. In *Handbook of Open Source Tools*, pages 191–200. Springer US, Boston, MA, 2011. https://doi.org/10.1007/978-1-4419-7719-9_10. (Cited on page 12)

[38] Anthony Kougkas, Anthony Fleck, and Xian-He Sun. Towards Energy Efficient Data Management in HPC: The Open Ethernet Drive Approach. In *2016 1st Joint International Workshop on Parallel Data Storage and Data Intensive Scalable Computing Systems (PDSW-DISCS)*, pages 43–48, Salt Lake, UT, USA, November 2016. IEEE. http://ieeexplore.ieee.org/document/7836567/. (Cited on page 57)

[39] Agata Kruszona-Zawadzka. MooseFS 3.0 User's Manual. https://moosefs.com/Content/Downloads/moosefs-3-0-users-manual.pdf, 2017. [retrieved: 26.04.2022]. (Cited on pages 21, 22, 36, 54, and 55)

[40] Michael Kuhn. JULEA: A flexible storage framework for HPC. In Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf, editors, *High Performance Computing*, pages 712–723, Cham, 2017. Springer International Publishing. (Cited on page 24)

[41] Michael Kuhn. OrangeFS - Hochleistungs-Ein-/Ausgabe, 2017. (Cited on page 18)

[42] Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf, editors. *High Performance Computing: ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers*, volume 10524 of *Lecture Notes in Computer Science*. Springer International Publishing, Cham, 2017. http://link.springer.com/10.1007/978-3-319-67630-2. (Cited on page 14)

[43] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/. (Cited on page 17)

[44] James H. Laros III, Kevin Pedretti, Suzanne M. Kelly, Wei Shu, Kurt Ferreira, John Vandyke, and Courtenay Vaughan. Energy Delay Product. In *Energy-Efficient High Performance Computing: Measurement and Tuning*, pages 51–55. Springer London, London, 2013. https://doi.org/10.1007/978-1-4471-4492-2_8. (Cited on page 43)

[45] Geoffrey Lefebvre and Michael J Feeley. Energy efficient peer-to-peer storage. Technical report, Technical Report TR-2003-17. Department of Computer Science, University of British Columbia, 2000. (Cited on page 56)

[46] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4):321–374, December 1990. https://dl.acm.org/doi/10.1145/98163.98169. (Cited on page 9)

[47] Guodong Li, Robert Palmer, Michael DeLisi, Ganesh Gopalakrishnan, and Robert M. Kirby. Formal specification of MPI 2.0: Case study in specifying a practical concurrent programming API. *Science of Computer Programming*, 76(2):65–81, 2011. https://www.sciencedirect.com/science/article/pii/S0167642310001164. (Cited on page 11)

[48] Walt Ligon. OrangeFS Overview Tutorial. https://storageconference.us/2012/Presentations/T05.Ligon.pdf, 2012. [retrieved: 12.05.2022]. (Cited on page 17)

[49] Seagate Technology LLC. Seagte ST3250318AS Product Manual. https://www.seagate.com/staticfiles/support/disc/manuals/desktop/Barracuda%207200.12/100529369c.pdf, 2009. [retrieved: 26.04.2022]. (Cited on page 28)

[50] SolidRun Ltd. ClearFog A388 Base SBC. https://www.solid-run.com/wp-content/uploads/2021/11/ClearFog-A388-Base-Datasheet-2021.pdf, 2021. [retrieved: 09.05.2022]. (Cited on page 55)

[51] Thomas Ludwig, Stephan Krempel, Julian Kunkel, Frank Panse, and Dulip Withanage. Tracing the MPI-IO Calls' Disk Accesses. In Bernd Mohr, Jesper Larsson Träff, Joachim Worringen, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 322–330, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. (Cited on page 45)

[52] Jakob Lüttgau, Michael Kuhn, Kira Duwe, Yevhen Alforov, Eugen Betke, Julian Kunkel, and Thomas Ludwig. Survey of storage systems for high-performance computing. *Supercomputing Frontiers and Innovations*, 5(1), 2018. https://centaur.reading.ac.uk/77664/. (Cited on page 56)

[53] MEAN WELL. MW HRP 450-15 Datasheet. https://www.meanwell.com/webapp/product/search.aspx?prod=HRP-450, 2021. [retrieved: 12.05.2022]. (Cited on page 27)

[54] Inc. Micron Technology. Micron 2300 SSD Datasheet. https://media-www.micron.com/-/media/client/global/documents/products/product-flyer/2300_client_nvme_ssd_product_brief.pdf?la=en&rev=b56df4d135384ca48c76e0fed9c1564b, May 2020. [retrieved: 09.05.2022]. (Cited on page 27)

[55] Frank Neese, Frank Wennmohs, Ute Becker, and Christoph Riplinger. The ORCA quantum chemistry program package. *The Journal of Chemical Physics*, 152(22):224108, 2020. https://doi.org/10.1063/5.0004608. (Cited on page 41)

[56] NETGEAR, Inc. Netgear GS110EMX Datasheet. https://www.netgear.com/images/datasheet/switches/webmanagedswitches/GS110EMX_GS110MX.pdf, 2021. [retrieved: 12.05.2022]. (Cited on page 27)

[57] Zhonghong Ou, Bo Pang, Yang Deng, Jukka K. Nurminen, Antti Ylä-Jääski, and Pan Hui. Energy- and Cost-Efficiency Analysis of ARM-Based Clusters. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, pages 115–123, 2012. (Cited on page 13)

[58] Edson L. Padoin, Daniel A. G. de Oliveira, Pedro Velho, and Philippe Olivier Alexandre Navaux. Evaluating Performance and Energy on ARM-based Clusters for High Performance Computing. In *41st International Conference on Parallel Processing Workshops, ICPPW 2012, Pittsburgh, PA, USA, September 10-13, 2012*, pages 165–172. IEEE Computer Society, 2012. https://doi.org/10.1109/ICPPW.2012.21. (Cited on pages 7 and 13)

[59] Ashwin Pajankar. Introduction to single board computers and Raspberry Pi. In *Raspberry Pi Image Processing Programming*, pages 1–24. Springer, 2017. (Cited on page 12)

[60] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. Association for Computing Machinery. https://doi.org/10.1145/50202.50214. (Cited on page 9)

[61] TK Priyambodo, AW Lisan, and M Riasetiawan. Inexpensive green mini supercomputer based on single board computer cluster. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, 10(1-6):141–145, 2018. (Cited on page 7)

[62] Nikola Rajovic, Alejandro Rico, Nikola Puzovic, Chris Adeniyi-Jones, and Alex Ramírez. Tibidabo: Making the case for an ARM-based HPC system. *Future Gener. Comput. Syst.*, 36:322–334, 2014. https://doi.org/10.1016/j.future.2013.07.013. (Cited on pages 7 and 13)

[63] R. Rew and G. Davis. NetCDF: An interface for scientific data access. *IEEE Computer Graphics and Applications*, 10(4):76–82, 1990. (Cited on page 12)

[64] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, Christos Kozyrakis, and Justin Meza. Models and Metrics to Enable Energy-Efficiency Optimizations. *Computer*, 40(12):39–48, 2007. (Cited on page 42)

[65] Xiaojun Ruan, Shu Yin, Adam Manzanares, Jiong Xie, Zhiyang Ding, James Majors, and Xiao Qin. ECOS: An energy-efficient cluster storage system. In *2009 IEEE 28th International Performance Computing and Communications Conference*, pages 79–86, Scottsdale, AZ, USA, December 2009. IEEE. http://ieeexplore.ieee.org/document/5403814/. (Cited on page 56)

[66] Sayan Saha and Jeff Darcy. Red Hat Gluster Storage: Direction, Roadmap and Use-Cases. https://videos.cdn.redhat.com/summit2015/presentations/12326_red-hat-gluster-storage-direction-roadmap-use-cases.pdf, 2015. (Cited on page 39)

[67] Samsung. Samsung V-NAND SSD 860 PRO Datasheet. https://www.samsung.com/semiconductor/global.semi.static/Samsung_SSD_860_PRO_Data_Sheet_Rev1_1.pdf, 2018. [retrieved: 12.05.2022]. (Cited on page 27)

[68] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, Kouichi Hirai, Atsushi Furuya, Akira Asato, Kuniki Morita, and Toshiyuki Shimizu. Co-Design for A64FX Manycore Processor and "Fugaku". In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020. (Cited on pages 7 and 13)

[69] M Satyanarayanan. A Survey of Distributed File Systems. *Annual Review of Computer Science*, 4(1):73–104, June 1990. http://www.annualreviews.org/doi/10.1146/annurev.cs.04.060190.000445. (Cited on page 56)

[70] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A large-scale field study. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 193–204, New York, NY, USA, 2009. Association for Computing Machinery. https://doi.org/10.1145/1555349.1555372. (Cited on page 54)

[71] Priya Sehgal, Vasily Tarasov, and Erez Zadok. Evaluating Performance and Energy in File System Server Workloads. In Randal C. Burns and Kimberly Keeton, editors, *8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 23-26, 2010*, pages 253–266. USENIX, 2010. http://www.usenix.org/events/fast10/tech/full_papers/sehgal.pdf. (Cited on pages 39 and 57)

[72] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In José M. Laginha M. Palma, Michel Daydé, Osni Marques, and João Correia Lopes, editors, *High Performance Computing for Computational Science – VECPAR 2010*, pages 1–25, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. (Cited on page 7)

[73] Hongzhang Shan and John Shalf. Using IOR to Analyze the I/O Performance for HPC Platforms. In *In: Cray User Group Conference (CUG'07)*, 2007. (Cited on page 14)

[74] IEEE Computer Society and The Open Group. IEEE standard for information Technology—Portable operating system interface (POSIX(TM)) base specifications, issue 7. *IEEE Std 1003.1, 2016 Edition (incorporates IEEE Std 1003.1-2008, IEEE Std 1003.1-2008/Cor 1-2013, and IEEE Std 1003.1-2008/Cor 2-2016)*, pages 1–3957, 2016. (Cited on page 10)

[75] SECO S.P.A. UDOO BOLT V8. `https://shop.udoo.org/en/udoo-bolt-v8.html`. [retrieved: 09.05.2022]. (Cited on page 13)

[76] Tappest sp. z o.o. MooseFS Website. `https://moosefs.com`, 2022. [retrieved: 12.05.2022]. (Cited on page 22)

[77] GlusterFS Development Team. GlusterFS Documentation. `http://docs.gluster.org/`. [retrieved: 12.05.2022]. (Cited on pages 19, 20, and 35)

[78] IOR Developer Team. IOR 3.3.1 Documentation. `https://ior.readthedocs.io/en/3.3/`. [retrieved: 03.05.2022]. (Cited on page 14)

[79] OrangeFS Development Team. OrangeFS Documentation. `http://docs.orangefs.com/`. [retrieved: 12.05.2022]. (Cited on pages 17 and 18)

[80] OrangeFS Development Team. OrangeFS Website. `http://www.orangefs.org/`. [retrieved: 12.05.2022]. (Cited on pages 18 and 39)

[81] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, IOPADS '99, pages 23–32, New York, NY, USA, 1999. Association for Computing Machinery. `https://doi.org/10.1145/301816.301826`. (Cited on page 35)

[82] Tran Doan Thanh, Subaji Mohan, Eunmi Choi, SangBum Kim, and Pilsung Kim. A Taxonomy and Survey on Distributed File Systems. In *2008 Fourth International Conference on Networked Computing and Advanced Information Management*, pages 144–149, Gyeongju, South Korea, September 2008. IEEE. `http://ieeexplore.ieee.org/document/4623994/`. (Cited on pages 9 and 56)

[83] ThinkParQ and Fraunhofer ITWM. BeeGFS Wiki: Frequently Asked Questions (FAQ). `https://www.beegfs.io/wiki/FAQ`, 2020. [retrieved: 12.05.2022]. (Cited on page 24)

[84] ThinkParQ and Fraunhofer ITWM. BeeGFS Documentation. `https://doc.beegfs.io/`, 2022. [retrieved: 12.05.2022]. (Cited on page 23)

[85] Chen Wang, Kathryn Mohror, and Marc Snir. File system semantics requirements of HPC applications. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '21, pages 19–30, New York, NY, USA, 2021. Association for Computing Machinery. `https://doi.org/10.1145/3431379.3460637`. (Cited on pages 10, 11, and 18)

[86] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In Brian N. Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 307–320. USENIX Association, 2006. `http://www.usenix.org/events/osdi06/tech/weil.html`. (Cited on page 15)

[87] Western Digital Corporation. WD Black WD10SPSX Datasheet. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/internal-drives/wd-black-hdd/product-brief-western-digital-wd-black-mobile-hdd.pdf, 2020. [retrieved: 12.05.2022]. (Cited on page 27)

[88] ZES ZIMMER Electronic Systems GmbH. ZES Zimmer LMG 450 Brochure. https://www.zes.com/en/content/download/286/2473/file/lmg450_prospekt_1002_e.pdf, 2010. [retrieved: 12.05.2022]. (Cited on page 27)

[89] Lingwei Zhang, Yuhui Deng, Weiheng Zhu, Jipeng Zhou, and Frank Wang. Skewly replicating hot data to construct a power-efficient storage cluster. *Journal of Network and Computer Applications*, 50:168–179, April 2015. https://linkinghub.elsevier.com/retrieve/pii/S1084804514001362. (Cited on page 56)

Statement of Authorship

I herewith assure that I wrote the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Magdeburg, May 13, 2022

_____

Signature