bachelor thesis

Efficient interaction between Lustre and ZFS for compression

submitted by

Sven Schmidt

Workgroup scientific computing
Department of informatics
Faculty for mathematics, informatics and natural sciences

Student-ID Number:	6647018
Degree program:	Informatics
First Reviewer:	Dr. Michael Kuhn
Second Reviewer:	Anna Fuchs
Advisors:	Anna Fuchs Dr. Michael Kuhn
Date of submission:	2017/08/28

Abstract

As predicted by Moore's law, computational power was increasing rapidly within the last years, roughly doubling every 14.5 months throughout the history of the TOP500 [KKL16, p. 75], whilst storage capacity and speed showed far less significant growing factors. This results in an increasing gap, and, especially for High Performance Computing, Input and Output became a performance bottleneck. Furthermore, for the storage of data with up to multiple petabytes using distributed file systems like Lustre, another bottleneck evolves from the need to transfer the data over the network.

For the compensation of this bottlenecks, investigating compression techniques is more urgent than ever, basically aiming to exploit computational power to reduce the amount of data transferred and stored. One approach for the Lustre file system was presented by Anna Fuchs in her thesis "Client-side Data Transformation in Lustre" [Fuc16]. For the efficient storage within the underlying ZFS file system, Niklas Behrmann extended ZFS to allow storing externally compressed data as if it was compressed by ZFS itself, which allows to make use of the already existing infrastructure [Beh17].

This thesis interconnects both works. First, modifications to the read and write path are made to handle compressed data, that is, receiving it from Lustre and handing it to ZFS and vice versa. Moreover, metadata regarding the compression (such as the used algorithm) is stored together with the data as a header and given back to the client for decompression on the read path. The ultimate goal is to tailor new functionality as tightly as possible to the existing structures for best performance. First benchmarks showed, that the amount of data transferred over the network could be reduced by a fair amount, while the new functionality did not introduce performance regressions. Rather, reading compressed data turns out to be indeed faster.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Thesis goals 3
1.3 Outline 3

2 Background **5**
2.1 ZFS Overview 5
2.1.1 Architecture 6
2.2 Lustre Overview 8
2.2.1 Architecture 8
2.3 Interaction 11
2.3.1 Important Layers 11
2.3.2 Read/Write path 14
2.3.3 Compression within the Lustre client 16
2.3.4 Support for external data compression in ZFS 17

3 Design **19**
3.1 Metadata for compressed chunks 19
3.1.1 Adaptive compression 19
3.1.2 `struct chunk_desc` 20
3.2 Preconsiderations 21
3.2.1 The metadata as a header 21
3.2.2 Adjusting blocksize and chunksize 22
3.2.3 Handling compressed, uncompressed and insufficiently compressed data . 23
3.2.4 Competent layers for writing the metadata 24
3.3 Storing pre-compressed data in ZFS 25
3.3.1 Parsing remote into local buffers 25
3.3.2 Committing data into ZFS 28
3.3.3 Writing the header 28
3.4 Reading compressed data from ZFS 30
3.4.1 The main reader `osd_bufs_get_read` 30
3.4.2 Extracting the chunk headers 31

4 Related work **33**
4.1 Compression of network traffic 33
4.2 Enhanced Lustre-ZFS interaction 34
4.2.1 End-to-End data integrity for Lustre and ZFS 34
4.2.2 Ongoing work 34

5 Implementation 35
5.1 Preparations 35
5.1.1 Indicating compressed I/O 35
5.1.2 Introduce chunk sizes to PTLRPC 36
5.1.3 Passing chunk descriptors from the target handler to OSD-ZFS 36
5.1.4 Identifying a chunk header 37
5.2 The modified write path 38
5.2.1 Determine which chunk is written 38
5.2.2 Extracting the largest psize 38
5.2.3 `osd_bufs_get_compressed_write` 38
5.2.4 Committing data into ZFS 40
5.3 The modified Read path 41
5.3.1 Preparing the reply 41
5.3.2 Allocating memory for the headers 41
5.3.3 `osd_bufs_get_compressed_read` 42

6 Evaluation 45
6.1 Correctness 45
6.1.1 Transmission of data 45
6.1.2 Transmission of the chunk descriptors 47
6.2 Benchmarks 47
6.2.1 Network transfer size 47
6.2.2 Read performance 48

7 Conclusion and future work 51

List of Acronyms 53

List of Figures 55

List of Listings 56

Bibliography 57

Appendices 61

A Infrastructure 62

B Sourcecodes 63
B.1 `cdesc_for_off` 63
B.2 `chunks_max_psize` 63
B.3 Modified `struct chunk_desc` 63
B.4 `osd_bufs_get_compressed_write` 64
B.5 Changing the blocksize 66
B.6 `osd_bufs_get_compressed_read` 67

1 Introduction

This chapter gives an introduction into the importance of High Performance Computing (HPC) for the problems of our time. Furthermore, the economical challenges in question of the increasing gap between computational speed and different storage factors for HPC are described. A prototype for client-side compression in Lustre as one possible solution is introduced. However, further work regarding the interaction with the underlying ZFS file system is required, being the goal of this thesis.

Nowadays, scientific research more and more relies on algorithms with a heavy demand for computational power. The ultimate goal is to solve the most important problems of our time by calculating and processing amounts of data on a scale so large it might have been unimaginable only a few years ago. For example, in bioinformatics, next generation sequencing technology allows to "produce enormous amounts of sequence data inexpensively, with up to a billion reads[1] produced per run" leading to petabytes of data representing genomic information of whole species [SM14]. The analysis of this data, for instance the calculation of phylogenetic trees (that is, tree-of-life relationships), can help us understand the origin of species. On the one hand, this satisfies human curiosity, but, more importantly, it enables us to cure diseases more efficiently by "predicting the natural ligands for cell surface receptors which are potential drug targets" [BM01]. However, for such analysis, even more computational power is needed. For instance, the *maximum likelyhood* method for calculating phylogenetic trees was proven to be NP-hard [CT05].

1.1 Motivation

Besides this example, modern scientific computations almost always involve heavy computation and Input and Output (I/O) needs. Following Moore's law, the number of transistors per square-inch on integrated circuits doubles roughly every 18 months, making computation cheaper, hence more powerful over time. When looking at the TOP500 list, the total computational power increased from 567.4 PetaFLOPS[2] in 2016 to 748.4 PetaFLOPS as of this writing [Edi]. Throughout the entire history of the TOP500, computational power nearly doubled every 14.5 months [KKL16, p. 75]. However, increasing computational power also involves larger storage capacity needs; following the initial example, in order to process genomic data, it must either be kept in the computers main memory, or it must be stored on secondary storage. Due to the limitations in size regarding a computers fast main memory, this mostly involves devices like traditional Hard Drive Disks (HDDs) or Solid State Drives (SSDs). This points out a second

[1] A read is a short genomic sequence of up to a few hundred base pairs
[2] $567.4 \cdot 10^{15}$ Floating Point Operations per Second

need for modern HPC: *Fast* storage devices.
As of today, many computers in the HPC area rely on distributed as well as parallel file systems like Lustre which allow to access data on multiple storage devices in parallel. For such file systems, another important factor must be considered, namely the network throughput. Similar to Moore's law, Jakob Nielsen made observations according to network throughput, predicting a growth rate of 50% per year ("Nielsen's law") [Nie98].

Ultimately, this results in different growth rates for the most important factors when it comes to HPC: Per 10 years, computational speed, storage capacity, storage speed and internet throughput show growing factors of 300, 100, 20 and 57, respectively [KKL16; Nie98]. The different growth factors are illustrated in Figure 1.1.

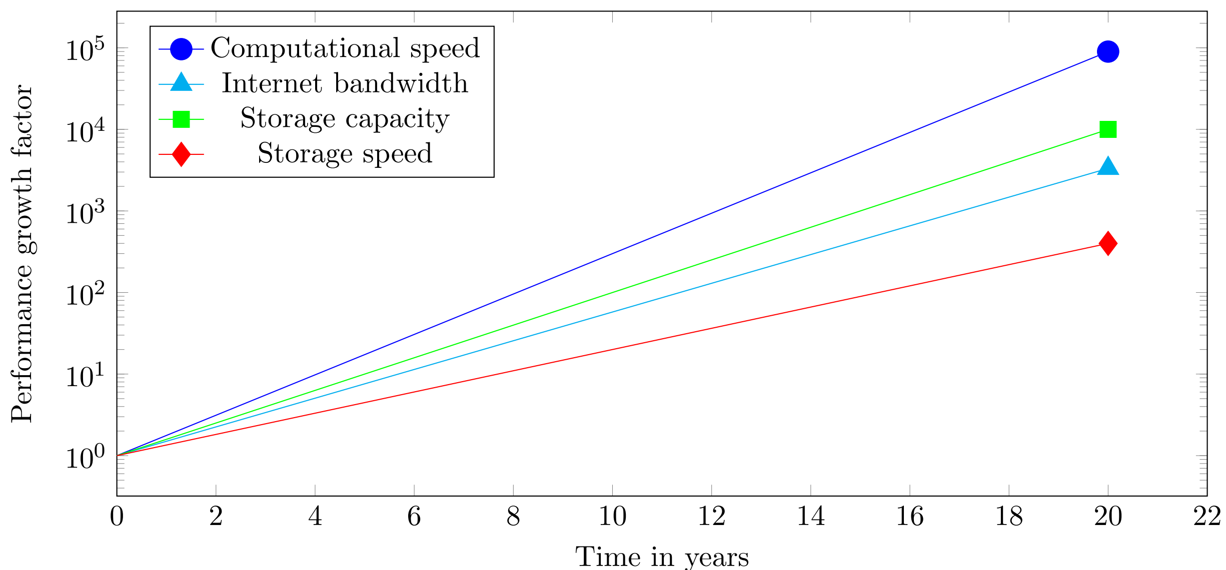

Figure 1.1: Development of computational speed, internet bandwidth, storage capacity and storage speed (based on [KKL16] and [Nie98])

While in the short-term it is possible to compensate for the increasing gap, for example by buying more storage hardware, in the long term this would not be possible. "Overall, the storage subsystems can be responsible for a significant portion of a system's total cost of ownership. In the case of Mistral[3], it accounts for roughly 20 % of the overall costs." [KKL16, p. 76] Hence, strategies for data reduction are increasingly being investigated.

One approach for Lustre was presented by Anna Fuchs in her thesis "Client-Side Data Transformation in Lustre" [Fuc16], ultimately aiming to perform adaptive client-side data compression before sending the data through the network. Benchmarks "[...] showed, compression can increase throughput by up to a factor of 1.2 while decreasing the required storage space by half" [Fuc16, p. 3]. For an efficient permanent storage within the underlying file system (which is either Lustre's own ldiskfs or ZFS), the internal structures of that system must be considered as well. However, the current work focuses on ZFS due to its existing compression pipeline, which can be exploited to save pre-compressed data as if it was compressed within ZFS itself. Such support for pre-compressed data was introduced by Niklas Behrmann in his thesis "Support for

[3]The HPC system of the Deutsches Klimarechenzentrum (https://www.dkrz.de/Klimarechner/hpc), placing 38th on the TOP500 June 2017

external data transformation in ZFS" [Beh17].

1.2 Thesis goals

An efficient interaction between the involved systems, Lustre and ZFS, regarding the client-side compression is subject to this thesis. This involves the fact that Lustre is only aware of one size, while compression and the involved functions within the Lustre client and the ZFS backend make a distinction between the physical (compressed) and the logical (uncompressed) size. Hence, new functionality must be introduced to Lustre's backend to interconnect both systems. Most importantly, when compressing the data, metadata regarding the nature of the compression must be stored along with the data. Without, for instance, the knowledge of the specific algorithm, decompression can not be done. The primary directive, however, should be not to cause additional costs which could nullify the benefit gained from compression, particularly when reading data. Hence, new features should be connected as tightly as possible with the available structure.

1.3 Outline

The remainder of the thesis is structured as follows:

1. In the **Background**, the technical fundamentals will be presented. This includes an overview of ZFS as well as Lustre. Therefore, the architecture of both systems is illustrated and the most important layers concerning the Lustre-ZFS interaction are described. Also, the basic idea for client-side compression within the Lustre client and the changes regarding pre-compressed data made to the ZFS backend are briefly depicted.

2. In the chapter **Design**, theoretical considerations regarding the interconnection of Lustre and ZFS for pre-compressed data are made. This includes the requirements evolving from the previous work as well as limitations and edge-cases. Eventually, the idea for implementing the neccessary changes is presented.

3. The chapter **Implementation** describes how the designed solution was integrated into the existing Lustre source code. In this chapter, detailed insights in the code are given and the most important changes are presented in all details.

4. In the end, an **Evaluation** of the work is given, especially aiming to show that the transfer of the data is correct and no performance regressions compared to the unmodified Lustre exist.

5. Eventually, a **Conclusion** is given and ideas for **Future Work** are described.

2 | Background

This chapter describes the abstract internal details of ZFS and Lustre. It is divided into three sections: First, an overview about the basics and architecture of ZFS is given. Then, an overview about Lustre is provided accordingly. Finally, the interaction between those systems within the Lustre I/O stack is discussed in terms of the most important layers involved. Also, the read and write paths are briefly demonstrated and the previous work is presented.

2.1 ZFS Overview

ZFS is a transactional file system, whose development was started in 2001 by Matthew Ahrens, Bill Moore and Jeff Bonwick, before the source code was eventually opened as a part of OpenSolaris in 2005 [Ope14]. While *ZFS* does not stand for anything since 2006 [Bon06], it originally was an abbreviation for *Zettabyte file system* because *Zetta* was "the largest SI prefix we [the developers] liked" [Ope11] and the name was a reference to the fact that ZFS "can store 256 quadrillion zettabytes" [Ope11][1]. Since 2013, ZFS is natively supported in Linux [Ope14].

Besides the large theoretical storage capacity of ZFS, according to Matthew Ahrens [Ahr16], its key features most notably include:

- **Pooled Storage**: Traditionally, a file system is designed to reside on a single physical device, thus when it is neccessary to create storage spanning multiple storage devices a *volume manager* is needed. ZFS, however, is a file system as well as a volume manager, being able to present a single *storage pool* from multiple devices and to manage the free space within pools.

- **End-to-end data integrity**: ZFS is able to detect and correct silent data corruption throughout the entire storage stack using checksums.

- **Copy-on-write**: Whenever ZFS writes data to the disk, it does write that data to an area of the disk not currently in use first and makes sure the data is consistent before eventually changing pointers to make the data persistent.

- **Native compression**: ZFS features compressing data before writing it to the disk. Among others, featured algorithms are GZip, LZ4 (default) and LZE.

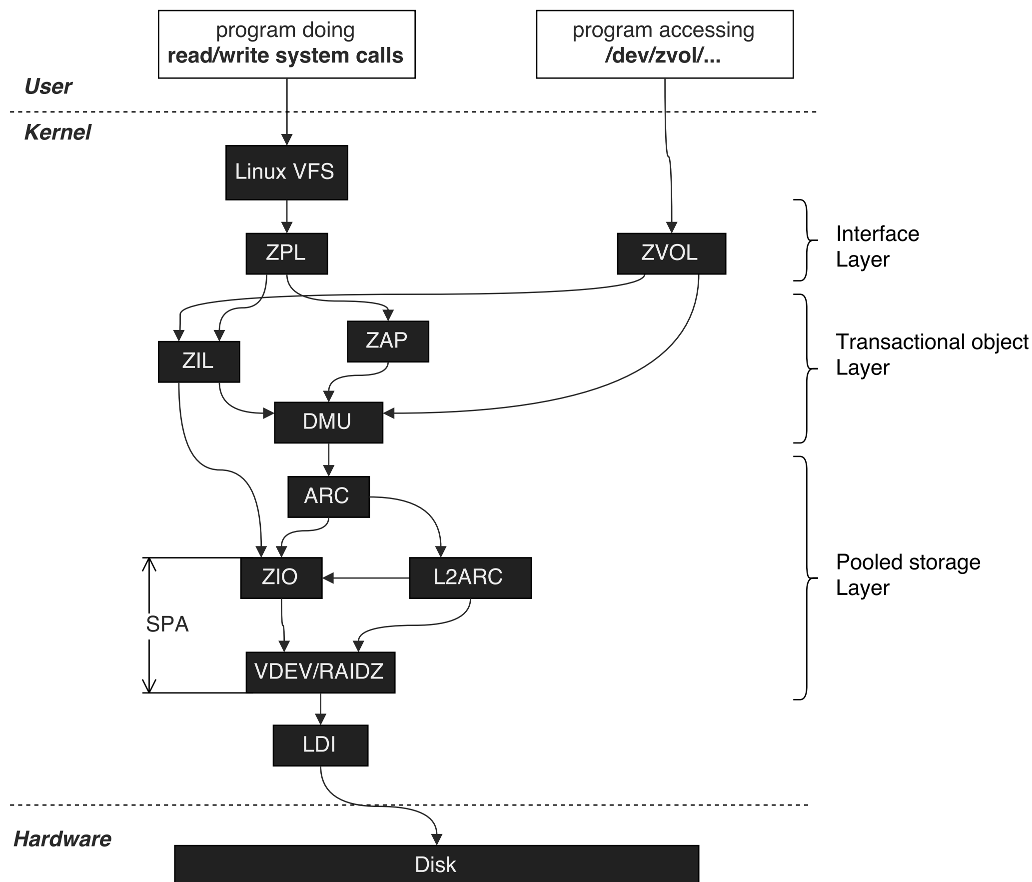

Figure 2.1: Relationship between ZFS' modules (based on graphics from [McK+14] (original) and [Beh17])

2.1.1 Architecture

Figure 2.1 shows the different modules of ZFS and the relationship to each other. ZFS itself is completely residing within kernel space. Its modules can be grouped into three layers: The *interface layer*, the *transactional object layer* and the *pooled storage layer*.

The former, the *interface layer*, can be seen as the *surface* of ZFS, which is the only part of ZFS userspace can directly interact with. ZFS is a file system compliant to the Portable Operating System Interface (POSIX), which means it presents a unified interface to the user in order to perform storage operations by calling functions such as `read`, `write` or `open` whose names are consistent among all file systems implementing POSIX. In consequence, storage operations can be performed without requiring further knowledge of the underlying structures or physical devices. In Linux, this unified interface is achieved through the Virtual File System (VFS). Within ZFS, the ZFS POSIX Layer (ZPL) is the layer implementing the VFS. Thus, when, for instance, the `write` syscall is used, ZPL performs adequate operations on the underlying layers to write the data to the disk.

Besides ZPL, a second interface exists in ZFS, namely ZFS Volumes (ZVOLs), which are logical volumes appearing as traditional, fixed-size disk partitions (e.g. `/dev/zvol/`) and can be used like regular block devices.

[1] One Zettabyte equals $1 \cdot 10^{21}$ bytes

On the other end of ZFS, there is one or multiple physical disks abstracted as a *pool* of storage. With each single device, it is communicated through a Linux Device Interface (LDI), which represents a special file in unix-like Operating Systems (OSes) implementing an interface for a device driver.

The in-between layers will be named and described in the following in the order they appear in Figure 2.1.

2.1.1.1 ZFS Intent Log (ZIL)

The ZIL stores per-dataset transaction logs to replay operations in case of a system crash. Although ZFS never leaves a consistent state and there's no immanent need to store information about how to restore the system to such a state, it is still required to store log information, for example in order "to guarantee data is on disk when the write, read, fsync syscall returns" [Dil10].

2.1.1.2 ZFS Attribute Processor (ZAP)

The ZAP is a property storage for the whole pool, allowing to store arbitrary {key, value} pairs associated with an object. For example, this includes metadata like {owner, sven}, {mode, 0777} or {eof, 131072}. The second important use-case is the implementation of directories.

2.1.1.3 Data Management Unit (DMU)

The DMU is "responsible for presenting a transactional object model" [Dil10] to its consumers. Interacting with the DMU involves objects, object sets and transactions (the term *transactional object layer* for DMU, ZAP and ZIL is derived from that fact). The main task of the DMU is to receive blocks of storage from the *pooled storage layer* and to export those blocks to its consumers in the form of abstract objects consisting of one or multiple blocks. In the reverse direction, it commits transactions from the above layers to the underlying storage. Transactions are "series of operations that must be commited to disk as a group; it is central to the on-disk consistency" [Dil10]. The importance of the DMU must be emphasized: Whenever data is read, written or modified, the caller performs operations on objects received from the DMU and gives those objects back to the DMU for persistent storage. Hence, data access always involves the DMU.

2.1.1.4 Storage Pool Allocator (SPA)

As implied by the name, the modules referred to as SPA are responsible for allocating storage from the pool, thus to manage free space on the underlying disks.

2.1.1.5 Adaptive Replacement Cache (ARC) and L2ARC

In general, an ARC is a "self-tuning cache [which] will adjust based on I/O workload" [Dil10]. ZFS in particular uses a modified ARC.
According to Matthew Ahrens, the ARC used by ZFS "is a caching layer [...] that's separate from the page cache which traditional file systems interact with" [Ahr16]. Following of that, besides being a caching layer, it also is the central instance for memory management within ZFS itself "which can be accessed with sub microsecond latency" [Gre08].

For reading and writing, the main tasks for the ARC differ:

- For **writing** data, the aspect of the *memory manager* is more important. Data which is subject to be written to the disk, is first copied into memory pages owned by the file system, which are requested from the ARC. Also, when such pages are given back to ZFS, the ARC involves the underlying layers for persistent storage.

- For **reading** data, the ARC is primarily a *caching layer*. Therefore, the ARC tries to allocate as much of the computers main memory as possible for caching objects. Usually, this includes all memory available to ZFS except for 1 GiB[2] [zfs15]. The large cache size is essential for good read performance since a cache miss almost always results in the involvement of the SPA, which must fetch the data from one or multiple disks first. Because this eventually implies the movement of physical parts, especially for traditional spinning discs, a cache miss can be very expensive.

The importance of the ARC as a caching layer for good read performance is emphasized by the existence of one or multiple additional second level ARCs (L2ARC). While the ARC caches data within the very limited main memory, a L2ARC "sits in-between [the ARC and the disk], extending the main memory cache using faster storage devices - such as flash memory based SSDs" [Gre08].

2.1.1.6 ZFS I/O (ZIO)

ZIO is the centralized I/O pipeline in ZFS. All physical disk I/O goes through ZIO. The pipeline includes, among others, compression, checksum calculation and data redundancy.

2.1.1.7 Virtual Device (VDEV)

A VDEV, only mentioned for the sake of completeness here, is an abstraction of physical devices, providing data replication and mirroring (RAID-Z, RAID-Z2) [Dil10].

2.2 Lustre Overview

Lustre is an "open-source distributed parallel file system" with an "extremly scalable architecture [...] popular in scientific supercomputing, as well as in the oil and gas, manufacturing, rich media and finance sectors" [Wan+09, p. 6]. As of this writing, it is the leading file system used in the list of the TOP500 by running on 90% of the top 10 and 75% of the top 100 [Gor17].

2.2.1 Architecture

In traditional network file systems there exists a central server providing a single network path between clients and server. This way, the network path turns out to be the performance bottleneck. Moreover, in case this server crashes, so does the entire system. Mostly, it is also not possible to add additional servers as a fallback. Both is not true for Lustre. It is distributed in the sense that there exist multiple storage servers providing the clients with access to the data. It is highly scalable in the sense that Lustre deployments with 4,000 storage systems were

[2] Here and in the following, when not noted otherwise, 1 KiB is 1024 bytes, 1 MiB is 1024 KiB etc.

successfuly tested[3] [Lus], so that in case of the failure of one server others can take over and new servers can be added into a running system. The storage capacity of the whole file system is the sum of the capacity of each single storage system. Finally, Lustre allows a true parallel access to the data with clients being able to access the same data at the same time and to write data in independent parts to multiple storage systems in parallel.

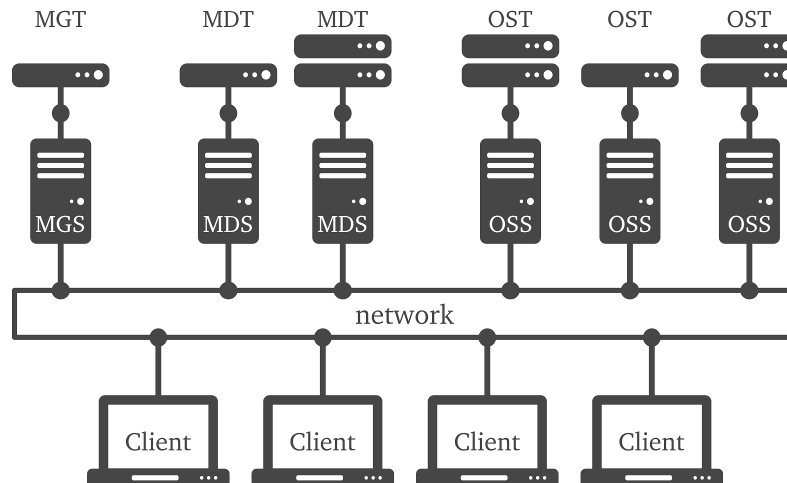

Figure 2.2: Lustre architecture [Fuc16]

Therefore, the architecture of Lustre is significantly different from that of a traditional network file system. First of all, Lustre consists of four main components: Object Storage Servers (OSSes), Meta Data Servers (MDSes) and Clients as well as not more than one Management Server (MGS). These components consist of several smaller layers with different, distinct functions. The most important of those layers are presented in Subsection 2.3.1. The major components are shown in Figure 2.2 and explained in the following.

2.2.1.1 Management Server

The MGS stores global information about the file system. This foremost includes global configuration valid for all components. Moreover, the MGS is the central authority for communication within Lustre. It knows about every MDS, OSS and Client and stores network identifiers as well as related information for them. The MGS can be seen as a registration unit where new providers or consumers within the system must make themselves known to Lustre. Due to the low storage needs of an MGS, it is mostly paired on one machine with an MDS.

2.2.1.2 Object Storage Servers

The OSSes are nodes handling requests for storage operations on actual data. Therefore, at least one Object Storage Target (OST) is connected to it. An OST eventually is the layer capable of storing the data on physical hardware, which, for instance, consists of HDDs with an own local file system. Currently, ldiskfs (a patched ext4) and ZFS are supported. For such devices, another interface exists, called an Object Storage Device (OSD). Each local file system implements this interface and provides unified operations such as allocating buffers and committing data into the file system. Existing implementations are `OSD-ldiskfs` and `OSD-ZFS`. The latter is the

[3] Which turns out to be the reasonable limit

most important layer in the course of this thesis because it is the direct link between Lustre and ZFS.

2.2.1.3 Metadata Servers

Metadata requests in Lustre are managed by MDSes. MDSes are nodes that, among other things, keep track about where data was stored and about the directory structure. Following of that, when considering reading and writing data, two aspects are especially important:

- When writing data, the client does not communicate directly with a storage device. Rather, first communication is done between the client and an MDS. The client requests storage space for an amount of data. The MDS then assigns space on one or multiple storage devices and grants permissions. Additionally, it tells the client with which OSSes it must further communicate and tells the involved OSSes about the details agreed upon.
- When reading, the client does not know where the requested data resides. As for writing, it is first communicated with an MDS which keeps track of the OSTs the data is stored.

2.2.1.4 Clients

Clients are the consumers of the file system, consisting of conventional personal computers like notebooks. For mounting the actual file system, the utility `mount.lustre` is used. For example, `mount -t lustre mgs@tcp0:/lustre /mnt/lustre/client` mounts the whole file system consisting of all OSSes and MDSes to `/mnt/lustre/client`. The data can then be accessed through the network like it were stored locally. Nevertheless, access to the data is only granted through the Lustre I/O stack. For that, a POSIX interface is provided.

2.2.1.5 Servers and Targets

MGSes, MDSes and OSSes represent nodes within the network. That is, servers providing services for metadata or data respectively. Each server holds at least one *target* as seen in Figure 2.2. A target represents a specific, physical device capable of actually storing data.

2.3 Interaction

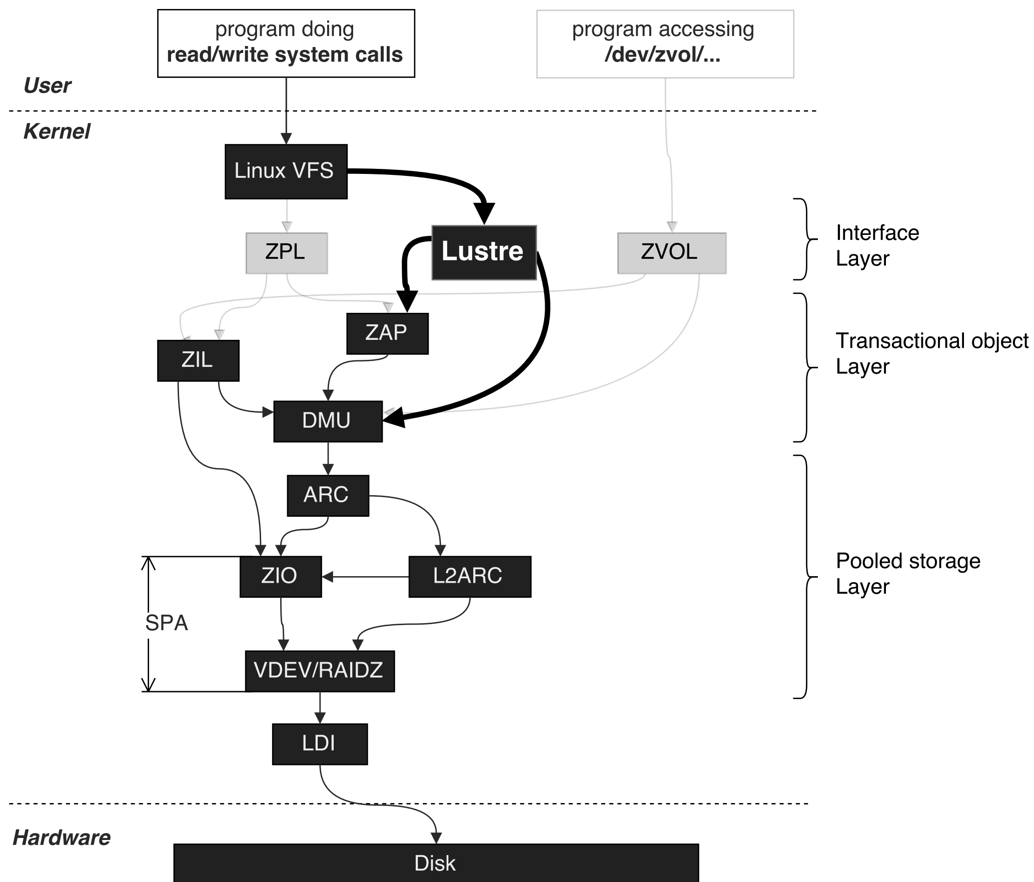

Figure 2.3: Lustre entering the ZFS I/O stack (based on graphics from [McK+14] (original) and [Beh17])

In the following, the interaction between Lustre and ZFS is illustrated. This includes an overview of the whole path of the data from userspace until it is eventually stored on disk. When it comes to the common Lustre-ZFS I/O stack, compared to the unmodified ZFS module relationship in Figure 2.1, Lustre is introduced within the *interface layer*. Here, it replaces ZPL as well as ZVOL (shown grayed) since neither is required by Lustre.

As illustrated in Figure 2.3, Lustre also implements the VFS interface. Hence, in the common software stack, userspace directly interacts with Lustre instead of ZFS.
On the other end, Lustre is set up atop the *transactional object layer* of ZFS. To perform file operations, Lustre therefore communicates with the ZAP and the DMU, respectively. While the Figure implies that Lustre is only a small part of the stack, rather the opposite is true as seen in the following.

2.3.1 Important Layers

Figure 2.4 provides an extended view on the Lustre-ZFS I/O stack by emphasizing the most important layers involved. However, it must be noted that several layers are missing. In particular, this involves metadata communication and locking, which are responsible for a large share of Lustre's overall communication. For the interaction, "direct" means the layers are connected

by means of functions which the upper layer calls on the lower one, while “indirect” includes either functions called on structs or a connection via omitted in-between layers.

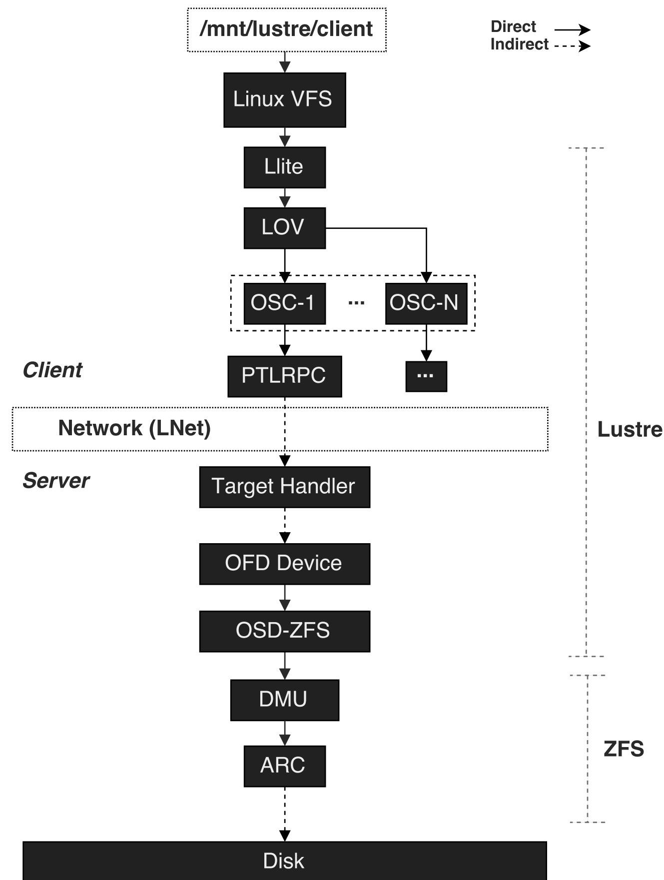

Figure 2.4: I/O stack including Lustre client, Lustre server and ZFS backend

2.3.1.1 llite

Lustre also being a POSIX-compliant file system, implements the unified interface described by the VFS as well as described before. llite is “a shim layer [...] that is hooked with VFS to present that interface. The file operation requests that reach llite will then go through the whole Lustre software stack to access the Lustre file system.” [Wan+09] Thus, llite is the top-most layer of Lustre handling all calls made and represents the actual file system logic. Moreover, llite is the only layer directly visible to userspace through mountpoints.

2.3.1.2 Clients

In Subsection 2.2.1 the *Client* was introduced as one major component of Lustre. However, *client* rather summarizes different components. Internally, Lustre distinguishes between clients as consumers for its other components by providing individual implementations of a common interface called *client_obdo*. Foremost, the implementations include Object Storage Clients (OSCs), Metadata Clients (MDCs) and Management Clients (MGCs) as consumers of OSTs, Meta Data Targets (MDTs) and Management Targets (MGTs), respectively.

The OSC is the most important client layer in the course of this thesis. Whenever a client computer wants to communicate with an OST, an instance of an OSC is created accordingly. Thus, there is an one-to-one relationship between those components for each request. Within the OSC layer, the request is prepared and eventually sent over the wire. Also, the introduced compression feature resides in the OSC.

2.3.1.3 Lustre Object Volume (LOV) and data striping

The parallelism is one of Lustre's key features. It is achieved, amongst others, by *file striping.* That is, a linear byte sequence, the data, is split into independent parts and then distributed between multiple OSTs, allowing to parallelize operations. For instance, let *stripe size* be 1 MiB and *stripe count* (number of OSTs to stripe over) be 3. Then, offsets (in MiB) $[0, 1), [4, 5), ...$ are stored as an object on OST-1, $[1, 2), [5, 6), ...$ on OST-2 and $[2, 3), [6, 7), ...$ on OST-3, respectively [Wan+09, p. 8]. Ultimately, the MDS keeps track of the mappings between stripe offsets and OSTs.

The striping is performed by the LOV layer, which is "responsible to assign the data pages to the corresponding OSCs. It determines the OSTs to talk with and presents them as a single volume to the client." [Fuc16, p. 15].

2.3.1.4 Remote Procedural Calls (RPCs), PTLRPC and bulk transfer

RPCs implement the basic request and reply mechanisms within Lustre. It "is a concept which allows local operations or calls to be executed on remote components" [Fuc16]. The basic structure can have multiple forms, consisting of different fields for each type of request. For instance, an RPC mostly contains a body with the actual request, the operation to perform (read / write), as well as flags and checksums.

When it comes to data transfer, most data is sent using *bulk transport*, used for "data of a variable amount which is too large to be sent within an RPC request itself" [Fuc16, p. 19]. Bulk transports are tightly connected to RPCs, nevertheless being a separate concept. In order to perform bulk transfer, there's two basic structures of importance: A *bulk descriptor* and at least one *Network I/O Buffer (niobuf).* The former describes the actual data to transfer regarding number of pages, size and offsets, the latter is a specialized structure for transferring data over the network.

A *Portal* can be compared to a port within the Transmission Control Protocol (TCP). There exist different portals for different operations, with one portal dedicated specifically to bulk transfer. The basic tasks performed by PTLRPC are sending requests and receiving replies, performing bulk data transfer and doing error recovery.

2.3.1.5 LNET

LNET is an abstraction layer allowing to communicate in a unified way independent of the underlying networking hardware. It has support for different networking types such as Infiniband or TCP/IP.

2.3.1.6 Target Handler

In Lustre's backend, the first layer of importance is the *Target Handler* which handles all incoming requests and offers another layer of indirection atop the actual target. Consequently, its first notable task is to determine with which kind of target actually is communicated with (MDT, MGT or OST). Furthermore, it receives all incoming RPCs and performs pre-processing. For example, the Target Handler checks the availability of all fields needed to handle a request and rejects those requests which are invalid. Hence, it is also a guard protecting the underlying local file system.
Besides preprocessing, the Target Handler also takes care of the remaining request from the moment the first RPC arrives until the data was handled by the target. In that timespan, the Target Handler is never left; it breaks an operation like writing data down into smaller problems, e.g. allocating buffers, for which the underlying layers provide solutions and calls the appropriate functions.

2.3.1.7 Object Filter Devices (OFDs)

When the *Target Handler* traverses, for example, through its `write` or `read` pipeline, it will eventually call operations on OFD Devices with some layers of indirection. OFDs are software layers handling actual file I/O by providing an unified interface for the needed operations. For instance, when preparing the write operation, it invokes `dbo_bufs_get`, which is implemented in OSD-ZFS as well as OSD-ldiskfs and takes care of allocating buffers from the targets main memory.

2.3.1.8 OSD-ZFS, DMU and ARC

OSD-ZFS eventually is the layer which actually communicates with both, Lustre as well as ZFS. OSD-ZFS acts as a translation unit, mapping functions owned by Lustre to operations known to ZFS and vice versa with the objective of interconnecting both systems. Therefore it directly invokes functions owned by the DMU. Since the DMU works by means of objects and transactions, OSD-ZFS owns an implementation tightly connected to the object structures known to the DMU. When it comes to involving the lower layers of ZFS, the DMU will interact with the ARC.

2.3.2 Read/Write path

In the following, the read and write path involving the important layers in Lustre and ZFS, respectively, are described briefly. However, metadata communication is omitted.

2.3.2.1 Writing data

1. The userspace application invokes, for instance, the syscall `write(struct file fd, const char __user *buf, size_t count)`, the POSIX function to write *count* bytes pointed to by *buf* into the file specified by the file descriptor *fd*. Here, `buf` points to memory owned by a userspace process.

2. The Kernel then invokes the according function implemented from the VFS. For that it looks into `struct file_operations` wrapped in `fd->f_op` and calls `fd->f_ops->write`, which is `ll_file_write` within **llite**.

3. Depending on the settings, LOV splits the data into stripes.
4. Eventually, the memory pages are inserted into the associated **page cache(s)** of one or many (if striping was performed) **OSCs** and flagged as *dirty*.
5. In order to send the data to a server, suitable space must be available on the targets. Upon each OSC initialization, an **OSS** grants the client some space. “A grant is just a number promised by the server that this client can flush out this much data and knowing the server can deal with it.” [Wan+09, p. 18]
6. On a regular basis, depending on how many pages are contained within the page cache and how large the actual grant is, pages are flushed out. By invoking `osc_check_rpcs`, an **RPC** is build. This involves preparing a bulk transfer by providing the neccessary data to a *bulk descriptor* and by assemblying *niobufs*.
7. Before the actual data is transferred, the RPC is sent to the server. Among other information, it contains the bulk descriptor.
8. The **target handler**, after pre-processing the incoming RPC and sending an acknowledgement back to the client, invokes `tgt_brw_write`. This function contains the whole processing of the write request. Among others, the most important tasks are:
   - Acquire locks on the involved file and return them later
   - Let the **OFD** layer prepare the write. The object is created if it does not exist and grant information is handled. Most importantly, the information provided by `struct niobuf_remote` wrapped in the RPC is processed. This is, parsing remote information (size and offsets) into local buffers, `struct niobuf_local`, containing enough memory for LNET to write the data into. Here, one `niobuf_local` equals one memory page of data.
   - For allocating memory, the OFD communicates with either, ZFS or ldiskfs through **OSD-ZFS** or **OSD-ldiskfs**.
   - In **ZFS**, sufficient memory is allocated through the **DMU**. The DMU, however, involves the **ARC**, since the ARC is the dedicated instance for memory management within ZFS.
   - The information about the local buffers is handed to **LNET** before eventually transferring the data.
   - During the whole process, the data is associated with an object and a transaction within ZFS. Thus, the transaction must be commited into the DMU, which then invokes the remaining ZFS I/O pipeline until the data is written persistently on a storage device.

2.3.2.2 Reading data

Reading data functions similar to writing data. After the appropriate system call is made, the call is handed over to the **OSC**. However, when reading, the read pages are cached within the page cache for subsequent readings. If the requested pages are still cached, they’re returned immediately without involving the server backend. Otherwise, a read RPC is assembled and sent over the network, handled again by the **Target Handler**.

- **OSD-ZFS** indirectly requests buffers associated with the object from the ARC by communicating with the DMU. At this point, when neither the ARC nor L2ARC have the data cached and can immediately return it, the ARC has to involve the underlying layers to read the data from disk. In either case, the buffers are larger in size than requested in order to efficiently perform read-aheads. Those buffers are then parsed as page-sized chunks into niobufs.

- In contrast to the write path, which does not involve sending a reply to the client, the read data must be transferred back to the client. An RPC always contains two areas, one being dedicated to the client and the other to the server, respectively. In this case, the request from the client regarding file information, offsets and sizes, are contained in the client's section. The server now fills its reply into the server area, again describing offsets and lengths to make the client prepare for the bulk transfer.

- Eventually, the data is received by the **OSC** and runs through the whole Lustre software stack again until it is passed to userspace.

2.3.3 Compression within the Lustre client

In her thesis "Client-Side Data Transformation in Lustre" [Fuc16] Anna Fuchs presented a prototype to introduce client-side compression to Lustre by adapting the I/O paths on the client[4]. In remembrance of the Introduction, the ultimate goal is to reduce the increasing gap between computational power on the one side and internet bandwidth, storage capacity as well as storage speed on the other side (compare Figure 1.1). The basic idea is to exploit computational power for compressing data on the client before sending it over the wire. Although operations related to compression such as copying memory between buffers come at a price, transferring large files over the network is far more expensive.

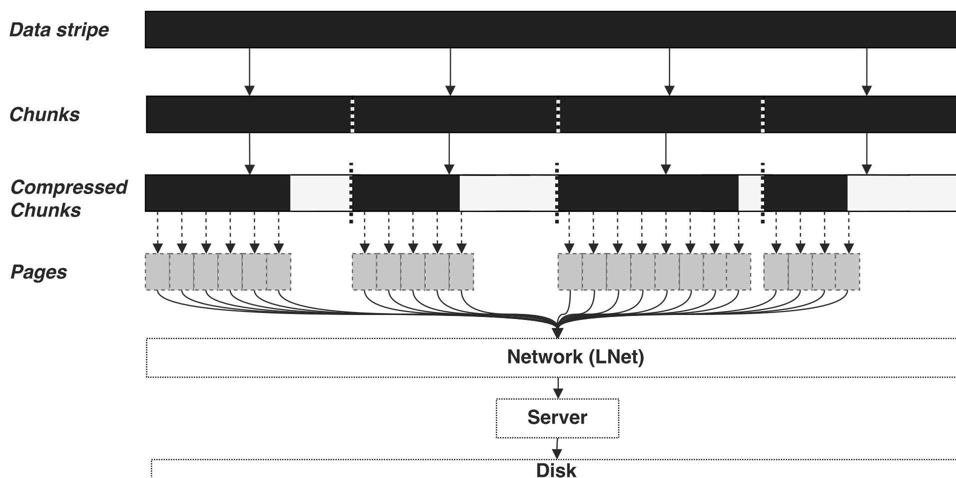

Figure 2.5: Basic idea of the currently implemented compression feature within Lustre's OSC layer

The principle of client-side compression is illustrated in Figure 2.5. The compression feature resides in the OSC layer and is done before the RPC is built. Thus, the RPC is filled with the correct (compressed) lengths and offsets within the file before sending it over the network.

[4]The compression feature is enabled via an autoconf parameter, `--enable-compression`

The data is passed as a pointer to page-aligned memory in userspace, `void __user *`. Consequently, the data must first be copyied from pages into large buffers for compression. Instead of using one buffer large enough for the whole data, smaller (but same-sized) buffers are used and, hence, the data is indirectly split into *chunks*. A chunk is a newly introduced unit for the client-side compression in Lustre. Basically, it is smaller than a file or a stripe, whilst still being larger than a page. The chunk size is a trade-off between different factors involved. Especially, the following points are important:

- **Parallel processing**: Each chunk is associated with an offset within the file, so that compressing and sending the data over the wire can be done in parallel for multiple chunks.
- **Adaption to the file systems record size**: The chunksize is chosen according to the record size of the underlying file system. This allows the file system to perform further optimizations if applicable.
- **Smaller chunks for read-modify-write**: The smaller the chunksize, the lower the costs for read-modify-write. This results from the fact that for modifying data within compressed data, read-modify-write becomes a read-decompress-modify-compress-write. Chunking the data allows to only modify the affected chunk and not the whole data stripe.
- **Larger chunks for good compression ratios**: Especially for dictionary-based compression algorithms like LZ4, too small chunks might result in bad compression ratios due to the accordingly smaller size of the dictionary. Research shows, that a chunk size equal to the ZFS record size (128 KiB) provides a ratio still suitable for efficient compression [Fuc+16].

For each compressed chunk, the data is then copied into pre-allocated `struct page`. This is neccessary since network transfer works on a per-page basis. Eventually, the data is sent over the wire and stored on a physical device by ZFS.

2.3.4 Support for external data compression in ZFS

As mentioned in Section 2.1, ZFS supports compression on a per-pool basis, which is ultimately processed as a part of the ZIO pipeline. Following of that, ZFS already provides an internal structure to store information about the compressed data, that is, the used algorithm, and, most importantly, `psize` as well as `lsize`. However, when compression is not turned off, ZFS will decompress the data while reading. When it comes to storing data passed from Lustre, it is usually desired to compress the data when written into ZFS to save storage space. This is not necessarily true for for pre-compressed data. Especially, two problems arise:

1. For already compressed data, the benefit from compressing it again are negligible. Rather, in particular for reading data, the costs of ZFS decompressing such double-compressed data before handing it to Lustre, might have a negative impact on the overall read performance.
2. ZFS compression is not done on a per-file basis but on each record. Since the compression in Lustre is based on stripes, when the involved units (especially chunks and records) are not coherent, Lustre might be forced to wait for ZFS to decompress a single record missing within a stripe. Following of that, it is intended to keep those values aligned.

Clearly, a conflict evolves when considering that in future Lustre setups clients with and without client-side compression might access the same storage servers on the same time. For the former, compression within ZFS is desired, while this is not true for the latter. Because of the significance of any decision for a good read performance, relying on the particular ZFS setup is not wanted.

Following of that, the most suitable solution is to enable Lustre to commit already compressed data to ZFS as if it was compressed by ZFS. For this task, Niklas Behrmann made changes to ZFS, introducing such support for pre-compressed data, as described in his thesis "Support for external data transformation in ZFS" [Beh17]. Basically, the functions called on the link between Lustre and ZFS (OSD-ZFS) were supplemented by similar functions performing reads and writes on the raw data in ZFS, hence skipping the compression and decompression, respectively. The specific functions will be presented in the following chapters as applicable.

3 | Design

This chapter explains how storing and reading compressed data along with their metadata is meant to be integrated into Lustre and how the modified ZFS functions are intended to make use of. The first section explains the detailed structure of the metadata. In the second section, preconsiderations are taken regarding edge cases and the most competent layer for handling the metadata. In the third section, different approaches for storing the data are illustrated with their respective limitations and benefits. Finally, a decision is made for the write path. The opposite direction, extracting the metadata when reading, is predetermined by this decision. A basic idea for reading data is presented in the fourth section accordingly.

3.1 Metadata for compressed chunks

3.1.1 Adaptive compression

Although ZFS only supports the algorithms LZJB, GZIP(1 - 9), ZLE and LZ4 for lossless compression (see `enum zio_compress`[1]), the introduction of the value `ZIO_COMPRESS_EXTERNAL` in [Beh17] does not limit Lustre to specific algorithms since it is identified as *externally compressed*. Nevertheless, following that Lustre resides in kernel space, it might be most practicable to foremost rely on those algorithms available in the Linux Kernel. Among others, these are LZ4, LZ4HC[2], Lempel-Ziv-Oberhumer (LZO) and Deflate. Since Kernel 4.11, there's also an implementation of LZ4Fast [Sch17]. The very same code was integrated into Lustre and is built when it is missing within the Kernel[3]. This approach is also conceivable for other algorithms. For Kernel 4.13, patches for the introduction of zstd are currently discussed [Lar17].

The variety of possible algorithms is important because the compression is meant to eventually be adaptive, which adds up "to somehow automatically and dynamically decide what is the best approach depending on the current situation and demands" [Fuc16, p. 57]. In particular, the decision for the best-suited algorithm and its parameters are based on factors like available memory and the utilization of the Central Processing Unit (CPU) as well as the nature of the data. For example, there exist algorithms specifically tailored to compression of images [Lin] or floating point data [LI06]. For this, the user will be able to provide Lustre with hints via a command-line interface (CLI). Possible decisions are:

- Focus on compression ratio when enough resources are available

[1] `zfs/include/sys/zio_compress.h`

[2] LZ4 specifically tailored to **h**igh **c**ompression ratios

[3] For instance, Kernel 3.10 used in CentOS 7 lacks LZ4 support

- Use a faster algorithm when a higher compression ratio might nullify the benefits for network transfer
- Skip compression, for example when the client finds that the benefit gained from compression does not compensate its costs

From this it follows that the specific algorithm with its parameters must be remembered for decompression.

3.1.2 `struct chunk_desc`

```
struct chunk_desc {
  __u32 ppages; /* number of pages (compressed) */
  __u32 psize;  /* size in bytes (compressed) */
  __u32 lpages; /* number of pages (uncompressed) */
  __u32 lsize;  /* size in byte (uncompressed) */
  __u32 cksum;  /* chunk checksum */
  __u32 algo;   /* algorithm, enum value */
};
```

Listing 3.1: The unmodified, but commented `struct chunk_desc` as introduced in [Fuc16]

Listing 3.1 shows the structure introduced to define metadata for compressed chunks in [Fuc16], `struct chunk_desc`. Its fields mainly result from the requirements of different algorithms regarding decompression on the one hand and the signatures of the functions within the DMU handling external compression on the other hand. Besides the used algorithm, the following fields exist:

- **cksum** is meant to ensure the chunk was transferred without corruption, therefore it is intended to complement the already available checksum for the RPC.
- **lsize** and **psize**: The former is the logical (uncompressed) size; the latter the physical (hence compressed) size. ZFS internally already distinguishes between lsize and psize because of ZFS' own compression within ZIO. For externally compressed data, both sizes must be passed from within Lustre. That is to the fact, that ZFS rounds the psize to the next power of two which is not suitable for using it for decompression.
  Moreover, some decompressors rely on the lsize, which, for instance, is true for LZ4Fast.
- **ppages** and **lpages**: The number of original pages and compressed pages is currently stored as well. However, this is likely to be omitted since, on the one hand, the information is useless when `PAGE_SIZE` differs for the decompressing machine. On the other hand, the same information results when dividing `lsize` and `psize` respectively through `PAGE_SIZE`.

Currently, as of this thesis, the stored metadata is meant to equal the chunk descriptor. However, this is not a requirement. In the future, it is conceivable to send more data with the chunk descriptor than actually stored, e.g. for providing additional information to the backend.

3.2 Preconsiderations

During compression, an array `struct chunk_desc *cdesc` is assembled within the client. To send it via the network, it must be introduced to PTLRPC. An RPC can take several forms, depending on the actual request, e.g. metadata, locking or, in this case, bulk read/write[4]. The generic structure for a bulk read or write request can be seen in Listing 3.2.

```
static const struct req_msg_field *ost_brw_client[] = {
        &RMF_PTLRPC_BODY,
        &RMF_OST_BODY,
        &RMF_OBD_IOOBJ,
        &RMF_NIOBUF_REMOTE,
#ifdef COMPRESSION_ENABLED
        &RMF_CHUNK_DESC, /* array of chunk descriptors */
#endif
        &RMF_CAPA1
};
```

Listing 3.2: Layout of an RPC for bulk read or writes (brw)

The layout of an RPC is defined as an array of pointers to message fields, that is `struct req_msg_field *`. A message field describes the size, type and flags as well as the offset within the message to directly address the field. `RMF_CHUNK_DESC` was introduced to hold the array of chunk descriptors, hence its type is `RMF_F_STRUCT_ARRAY`, a flag describing an array of struct, and its base size is defined as `sizeof(struct chunk_desc)`. The actual size of the field is set separately as soon as it's known (that is, when creating the request) and is required to be a multiple of the base size[5]. PTLRPC provides functions to set the size and to read as well as modify the contents of a message field, for example: `cdesc = req_capsule_client_get(pill, &RMF_CHUNK_DESC)`, where `pill` is a `struct req_capsule`, which is a wrapper to hold, among others, the actual request (`struct ptlrpc_request`) and the format of the RPC (`const struct req_format`).

3.2.1 The metadata as a header

Since this struct ultimately represents metadata, it would be consequential to store it accordingly, that is, communicating with an MDT, eventually involving the ZAP. However, a key requirement for the adaptive compression in this context is not only to involve Lustre, but to eventually give ZFS access to this metadata in the future to be able to apply optimizations if applicable. With that in mind, storing the data on an MDT is not sufficient since ZFS does not have access to the internal structures of Lustre.

Besides Lustre, ZFS also stores metadata . For instance, within the blockpointer, some required fields are already stored by ZFS. This includes the (rounded) psize as well as the lsize. Following of that, it would be possible to extend the blockpointer to store the chunk descriptor. When reading the data, ZFS would provide Lustre with the metadata (for instance, by

[4] All available layouts are defined in `lustre/lustre/ptlrpc/layout.c`

[5] When this is not true, it won't be capable of being used as an array

filling pointers). However, this would affect the on-disk layout of the data and break backwards-compatibility. Since ZFS is not only used together with Lustre, this is not appropriate.

In conclusion, the best-suited approach is to make the description of the data a part of the data itself, which comes down to adding a header. The data can be accessed from both, Lustre and ZFS without requiring extensive, structural changes in neither. Consequently, the only requirement for reading or modifying the header is to be aware of its presence. For writing, it must be known where it is expected to be (that is, preceding a chunk). In Figure 3.1, this idea is illustrated.

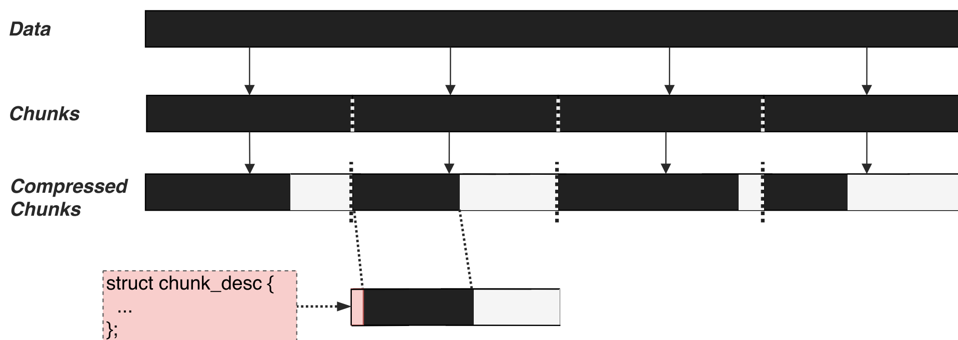

Figure 3.1: Basic idea for making the metadata a part of the data

This approach ultimately involves to ensure there is enough space for the header. When compressing the data, this results in space being freed in the end of each chunk (assuming `lsize != psize`). However, this does not mean sufficient space remains when dividing the chunk into pages. As an example, let a file be 12 KiB of size and assume it takes 8,190 bytes compressed, then it would be divided into two pages[6]. Following that, this results in $2 \cdot 4096 = 8192$ leaving 2 bytes for the header, which is not sufficient. Consequently, reserving additional space must be taken into concern.

3.2.2 Adjusting blocksize and chunksize

An important point for the write and read path respectively is the blocksize within Lustre. Initially, when an object is created from within Lustre, the blocksize is set to `PAGE_SIZE` and later grown according to the filesize. However, for the handling of pre-compressed data in ZFS, this is not appropriate. For the best integration within the existing structures of ZFS, it was concluded that "for pre-compressed blocks the logical size should always equal the recordsize" [Beh17, p. 62], which is 128 KiB. This follows from the fact, that the write path for pre-compressed data is meant to store compressed data as if it was compressed by ZFS, which is done in the means of records and, hence, a pre-compressed block must have the size of such. This is especially important for reading data, since when a chunk is tied to a record in ZFS, a more efficient read-ahead is possible.
Currently, the chunksize is already set to 128 KiB using a magic number. However, in the course of this thesis, a common constant for server and client must be added and the blocksize will be set accordingly.

[6] Assuming `PAGE_SIZE == 4096`

3.2.3 Handling compressed, uncompressed and insufficiently compressed data

When the compression feature is enabled, the client may provide the server with data fitting in one of the following three categories, assuming the chunksize is set to the ZFS recordsize as described in the previous subsection:

- `psize == lsize == chunksize`: As described before, the *adaptive* behavior includes the possibility to skip compression per chunk if no benefits are expected. If this condition holds, given that blocksize, ZFS recordsize and chunksize are set equal, the header can not be stored on the same block as the data.
- `chunksize - psize` $<$ `sizeof(struct chunk_desc)`: Compression was performed, but less than the header size was saved. For example, assume `psize = 131030`. Then, with a chunksize of 128 KiB, this leads to a reduction factor of $1 - \frac{131030}{131072} \approx 0.00032$. Due to the insignificant savings, compression might as well be omitted. Nevertheless, this always leads to the uncompressed case.
- `chunksize - psize` $\geq$ `sizeof(struct chunk_desc)`: This is the desired[7] case. The data was compressed and a fair amount of space was saved.

Apart from the third case, for which storing the header is always possible without further effort, the following solutions are considerable for the other two:

1. Not storing the header
2. Adding additional blocks
3. Adjusting the chunksize

The first case, however, results in problems when a header is assumed to be available during read and it is missing. This is in particular true when the flag `db_compressed` of the dnode for indicating pre-compressed data is set (compare [Beh17, p. 64]). Eventually, one can not be sure why the header is missing:

- By intent, that is, it was not stored due to insufficient space
- Due to errors, e.g. data corruption.

Nevertheless, following the fact that the client sent a chunk descriptor on the write path, the same descriptor is expected when reading. When it was not stored, it must be assembled.

For the second solution, the chunk with header is larger than 128 KiB, therefore the data at offsets $(128KiB, 128KiB + sizeof(structchunk_desc)]$ is stored on a second block within ZFS. For subsequent data, two possibilities follow:

1. Store the next chunk *dense*, that is on the same block the other chunk ends. This requires re-calculations of offsets and re-alignments of the whole data.
2. Start the next chunk on the next offset dividable by the chunksize, ultimately wasting `131072 - sizeof(struct chunk_desc)` bytes for each such uncompressed block.

[7]It is meant to give the administrator the possibility to define a threshold for the right-hand side of the expression, that is, when compression is beneficiary

In conclusion, while both previous approaches require several, additional changes and assumptions, which are error-prone, this is not true for adjusting the chunksize. Ultimately, this adds up to setting the chunksize to `131072 - sizeof(struct chunk_desc)`. Hence, this approach was followed. However, it should be noted that the other solutions can be implemented as well but they require additional assumptions (for instance, that a header is only missing when it was not stored).

3.2.4 Competent layers for writing the metadata

For adding the header to the data, several layers are competent: The client[8], the target handler, OFD, OSD-ZFS or an in-between layer (compare Figure 2.4). However, especially the layers succeeding the target handler, before involving OSD-ZFS, add several abstractions for a generic interface. Given that, adding the header should either be performed as *early* or as *late* as possible, ultimately reducing the space of feasible solutions to the following:

1. Within the client, while doing the actual compression without involving the server backend
2. On the server, before the data is committed
3. On the server, before the data is actually transferred

3.2.4.1 The client approach

Writing the chunk descriptors within the client is the earliest moment possible. Consequently, no extensive changes in the backend seem to be needed. At the same time, not involving the backend has several disadvantages:

- The functions within ZFS handling pre-compressed data require `psize` as well as `lsize` since they are stored by ZFS. Therefore, the server backend must either extract the descriptors or the sizes must be sent with the RPC. However, this requires the backend to check whether the fields are available in the RPC and an additional flag must be sent indicating if the functions for pre-compressed data must be chosen.
- When modifying the data, this would always require to communicate in the matter of chunks, e.g. sending the file to the client, modifying the data and sending the modified, re-compressed chunk back. Another possibility would be to only send the modified data back, let the backend decompress the appropriate chunk, change the data and re-compress. This can only be done efficiently if the server is involved in handling the metadata.
- Accordingly, assuming a client which does not support or use the compression feature, requests compressed data. In that case, when the backend has appropriate functions to handle compressed data, it might process decompression and send uncompressed data to that client.
- When introducing *compressed chunks* to the backend building an appropriate, generic interface, an implementation within ldiskfs or future local filesystems can be done with less effort.

Altogether, no further efforts were made to write the metadata within the client layers.

[8] here and in the following, when not noted otherwise, *client* refers to the OSC

3.3 Storing pre-compressed data in ZFS

As described in Subsection 2.3.1, the data transfer follows two steps: First, the RPC containing metadata such as the chunk descriptors is preceeding the actual data transfer. After preparations are done, bulk I/O is performed. For the server's write path, `tgt_brw_write` is the function responsible. For either task. The most important steps are shown in Listing 3.3.

```
1  rc = obd_preprw(tsi->tsi_env, OBD_BRW_WRITE, exp, &repbody->oa,
2      objcount, ioo, remote_nb, &npages, local_nb); /* parsing from
     remote to local buffers */
3  ...
4  for (i = 0; i < npages; i++)  /* provide local buffers to PTLRPC */
5    desc->bd_frag_ops->add_kiov_frag(desc,
6              local_nb[i].lnb_page,
7              local_nb[i].lnb_page_offset,
8              local_nb[i].lnb_len);
9  ...
10 rc = target_bulk_io(exp, desc, &lwi); /* perform bulk I/O */
11 ...
12 rc = obd_commitrw(tsi->tsi_env, OBD_BRW_WRITE, exp, &repbody->oa,
13       objcount, ioo, remote_nb, npages, local_nb, rc); /* commit */
```

Listing 3.3: `tgt_brw_write` (commented): Most important steps

Namely, these are:

1. Allocating local memory (`obd_prep_rw`, which invokes `osd_bufs_get_write`)
2. Providing local memory pointers to PTLRPC (`for` loop, lines 4 to 8)
3. Performing bulk I/O (`target_bulk_io`)
4. Commiting data into the local file system (`obd_commitrw`)

The first and the last step, that is before and after data transfer, are subject to be adapted due to the new functions introduced for handling pre-compressed data in [Beh17]. Besides that, for writing the header to the data, these are the only steps to be considered as well. This is to the fact that the logic of *transferring* data is handled by *LNet*, in particular *lib move*[9], whose interfaces are tailored to all kinds of networking within Lustre.
In the following, steps 1 and 4 are discussed in terms of the changes to be made for pre-compressed data on the one hand and their suitability for adding the header on the other hand.

3.3.1 Parsing remote into local buffers

The function `obd_preprw` (lines 1 and 2) handles the parsing of remote into local buffers (niobufs), that is remote niobufs (rnbs) into local niobufs (lnbs). The associated structs are shown in Listing 3.4[10].

An array of `struct niobuf_remote` is wrapped in the RPC (compare Listing 3.2, namely `RMF_NIOBUF_REMOTE`). Consequently, no data is contained. Instead, the set of rnbs *announce*

[9] `lnet/lnet/lib-move.c`

[10] It is to mention that both structs are defined in different headers, lustre_idl.h (rnb) and obd.h (lnb)

```
struct niobuf_remote {
  __u64 rnb_offset;
  __u32 rnb_len;
  __u32 rnb_flags;
};
struct niobuf_local {
  __u64   lnb_file_offset;
  __u32   lnb_page_offset;
  __u32   lnb_len;
  __u32   lnb_flags;
  int   lnb_rc;
  struct page *lnb_page;
  void    *lnb_data;
};
```

Listing 3.4: `struct niobuf_local` and `struct niobuf_remote`

the actual data and inform the server, how much space it must prepare according to the grant. Therefore, an rnb only consists of three fields: The offset within the file, the length of associated data and flags. It must be noted, however, that *file* in this context refers to the object actually stored on the precise target. With that in mind, when the data is striped, a distinction between the object stored locally and the global file it belongs to is immanent. Furthermore, an OST has no knowledge about global offsets since keeping track of that is subject to the MDSes.

The main task for `obd_preprw` is to parse remote information about the data (rnbs) into local memory to hold the data after transmission (lnbs). Therefore, each lnb is associated with a page-sized part of the data at a precise offset and owns a pointer to `struct page` for PTLRPC to copy data into. Within OSD-ZFS, `osd_bufs_get_write`[11] is invoked. The function can allocate memory for lnbs using either *zerocopy* or *anonymous pages*.

3.3.1.1 Anonymous pages

An *anonymous page* simply refers to a single memory page obtained from the Memory Management Unit (MMU).

3.3.1.2 Zerocopy

The basic idea of *zerocopy* is *loaning* buffers from the ARC instead of the Kernel. Requests are done by invoking `arc_buf_t *dmu_request_arcbuf(dmu_buf_t *handle, int size)`, which requests `size` bytes from the ARC, so that an arcbuf can be larger than a page. This approach has the advantage that an additional `memcpy` from buffers owned by Lustre into buffers owned by ZFS is omitted since Lustre is allowed to directly use buffers from ZFS (thus zerocopy). Consequently, zerocopy is preferred whenever feasible. However, assigning an arcbuf to an offset within the file is limited by two constraints: `offset == db->db.db_offset && blksz == db->db.db_size`. This sums up to being only able to assign arcbufs which have the blocksize and are aligned accordingly. As a matter of fact, `db->db.db_offset` is not increased for every write. Rather, it is set only to multiples of the blocksize.

[11] `osd_bufs_get_write(const struct lu_env *env, struct osd_object *obj, loff_t off, ssize_t len, struct niobuf_local *lnb)`

3.3.1.3 `osd_bufs_get_write`

```
1  i← 0 ;
2  while len >0 do
3      if off is aligned to blocksize then
4          arcbuf←dmu_request_arcbuf(obj, size) ;
5          buf_size←arcbuf->size ;
6          while buf_size >0 do
7              if this is the first page then
8                  lnb[i].lnb_data←arcbuf ;
9              lnb[i].lnb_page←kmem_to_page(arcbuf->b_data + off_in_buf) ;
10             off_in_block←off_in_block + min(PAGE_SIZE, len) ;
11             off←off + min(PAGE_SIZE, len) ;
12             i←i + 1
13     else
14         lnb[i].lnb_page←alloc_page() ;
15         off←off + min(PAGE_SIZE, len) ;
16         i←i + 1
```

Figure 3.2: Pseudocode for the main loop in `osd_bufs_get_write`

Figure 3.2 shows the most important parts of the main loop in `osd_bufs_get_write`. The function is called once for every rnb. Therefore, `off` as well as `len` refer to `rnb_offset` and `rnb_len` respectively. As long as there is information to parse, it is checked whether the current offset is aligned to the blocksize. If not, an anonymous page is allocated (line 14). Otherweise, an arcbuf of the appropriate size is requested for zerocopy (line 4) and then split into lnbs page-wise (loop lines 6 to 12). `kmem_to_page` ultimately invokes `vmalloc_to_page`, which "converts a kernel virtual address obtained from vmalloc to its corresponding struct page pointer" [CRK05, p. 460]. Eventually, the zerocopy case comes down to splitting one large buffer into several page-aligned pointers, which is required by PTLRPC. When assigning the arcbuf into ZFS, however, this is not done page-wise. Instead, the `arcbuf_t *` is remembered within the `lnb_data` field of the first lnb (line 8) and used for committment into ZFS.
Eventually, these methods do not exclude one another. For example, the last page normally relies on an anonymous page (assuming it is not entirely filled).

3.3.1.4 Allocating memory for pre-compressed data

Since for pre-compressed data the logical size always equals the record size (compare Subsection 3.2.2), the assignment constraints for zerocopy always hold. Hence, memory allocation will be handled entirely by the new function `dmu_request_compressed_arcbuf(dmu_buf_t *handle, int psize, int lsize)`. For the unmodified write path within ZFS, a record is compressed within ZIO and `psize` and `lsize` stored accordingly. When an already compressed block of size `psize` is passed from within Lustre, the `lsize` must be passed as well. The information is then associated with the arcbuf, which is the main task of the function compared to `dmu_request_arcbuf`, and later used when giving it back to ZFS (compare [Beh17, p. 65]).

3.3.2 Committing data into ZFS

After parsing remote into local buffers, each lnb is provided to the bulk descriptor (see Listing 3.3, lines 4 to 8) before bulk transfer is eventually initialized using `target_bulk_io` (ibid., line 10). The remaining step is actually committing the received data into ZFS. This is performed by `osd_write_commit`, whose main loop is seen in Figure 3.3.

```
1 grow_blocksize(first_lnb, last_lnb) ;
2 foreach lnb do
3     if lnb_data is not NULL then
4         dmu_assign_arcbuf(handle, lnb_file_offset, lnb_data, tx) ;
5         /* Skip other lnb for this arcbuf */
6     else
7         dmu_write(os, obj, lnb_file_offset, lnb_len, lnb_page, tx) ;
```

Figure 3.3: Pseudocode for the main loop in `osd_bufs_get_write`

The first notable aspect is, that the blocksize is grown according to the file size, which is calculated using the first and last lnb respectively. After that, for each lnb one of two tasks are performed:

1. If the **lnb_data** field contains an arcbuf, it is *given back* to the ARC using `dmu_assign_arcbuf(dmu_buf_t *handle, uint64_t offset, arc_buf_t *buf, dmu_tx_t *tx)`
2. Otherwise, `dmu_write(objset_t *os, uint64_t object, uint64_t offset, uint64_t size, const void *buf, dmu_tx_t *tx)` is used, which copies `size` bytes from `buf` into ZFS-internal buffers before assigning these to the ARC. The data is then written to `offset` within the file.

3.3.2.1 Committing pre-compressed data

For pre-compressed data, the blocksize will not be calculated but set to the chunksize plus the header size, which sums up to the ZFS record size of e.g. 128 KiB (compare Subsection 3.2.3). The task of committing pre-compressed blocks into ZFS then comes down to a call to `dmu_assign_compressed_arcbuf`. It was designed to give back an arcbuf to ZFS as its uncompressed counterpart would do, only it skips the compression within ZIO and just stores the block as is. The handling of pre-compressed data can ultimately be represented as a third branch of the `if` as seen in Figure 3.3.

3.3.3 Writing the header

For writing the header, two options follow from the preceding discussions: While allocating buffers (`osd_bufs_get_write`) or before commitment (`osd_write_commit`).

3.3.3.1 Before commitment

The most suitable solution for writing the header after receiving data is illustrated in Figure 3.4. However, only the first chunk is taken into concern here for simplicity; subsequent chunks have an offset according to the chunksize. First, the header is written using `dmu_write`, which allows

```
1 if lnb_data is not NULL then
2     dmu_write(os, obj, 0, sizeof(struct chunk_desc), cdesc) ;
3     dmu_assign_compressed_arcbuf(handle, offset + sizeof(struct chunk_desc), lnb_data,
       transaction) ;
4     /* Skip other lnb for this arcbuf */
```

Figure 3.4: Writing the `struct chunk_desc` before committing using `dmu_write`

writing data of arbitrary size to arbitrary offsets within the file. Following that, the arcbuf is given back with an offset beyond the header. For this solution, however, the constraints for assigning an arcbuf evaluate to `sizeof(struct chunk_desc) == 0 ...`, which is never true, and leads to misalignments for subsequent buffers, too, ultimately never being able to assign the arcbuf.
Other as the new function, the original implementation offers a fallback in such case, that is, copying the arcbufs content into new buffers first and then using `dmu_write`. However, since all pre-compressed blocks have the same size as the record, subsequent blocks are misaligned too, which means the fallback becomes the normal case, although it is far more expensive. Eventually, all conceivable solutions would require the data to be shifted, resulting in performance regressions compared to the original implementation.

3.3.3.2 Before data transfer

Hence and in conclusion, the approach to assign the header before data is received, was followed. Figure 3.5 provides a summarized illustration of this approach.

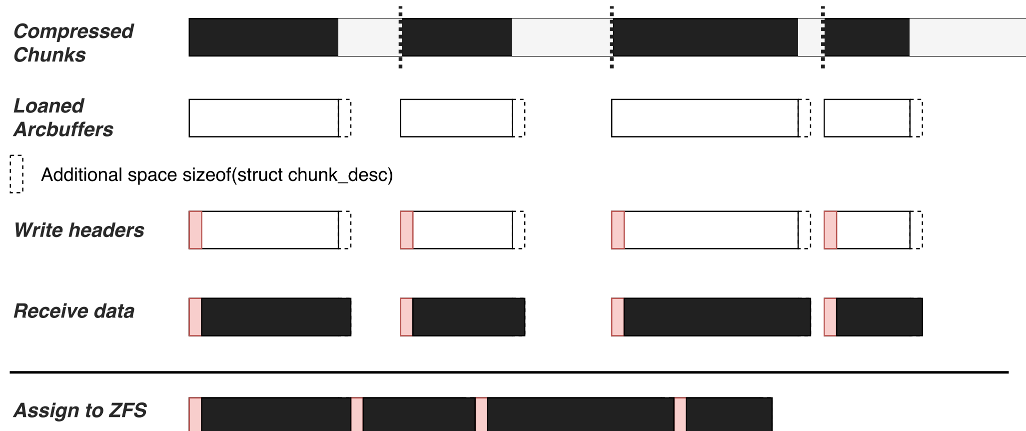

Figure 3.5: Assigning the header following the approach to modify arcbufs before receiving data

Ultimately, `osd_bufs_get_write` (compare Figure 3.2) is subject to be adapted, such that PTLRPC writes the data with an offset honoring the header size for each chunk into the buffers obtained using zerocopy. For that, the header size is added to the psize when requesting the buffer and, hence, a `memcpy` is sufficient for writing the header to offset 0 within the buffer. In `osd_write_commit`, the buffer including the header is committed as is.

3.4 Reading compressed data from ZFS

3.4.1 The main reader `osd_bufs_get_read`

Reading data is mostly handled by `tgt_brw_read` within the target handler. Compared to writing data, the committment step is missing, thus reading only involves parsing buffers in `obd_preprw`, which eventually calls `osd_bufs_get_read`. Its prototype can be seen in listing 3.5.

```
1 static int osd_bufs_get_read(const struct lu_env *env, struct osd_object
      *obj, loff_t off, ssize_t len, struct niobuf_local *lnb)
```

Listing 3.5: Prototype of `osd_bufs_get_read`

Basically, the function transforms `len` bytes from buffers associated with the object `obj`, starting at offset `off` into local niobuf. In contrast to the write path, the niobufs assembled during read, are clearly already associated with data via their respective `struct page` pointer. The data is not copied into Lustre-internal buffers, though. Similar to the equivalent write operation, a *zerocopy* solution exists. Moreover, reading always follows zerocopy. Therefore, `dmu_bufs_hold_array_by_bonus` is called, which "retrieves all dbufs directly that hold the wanted data directly from the DMU" [Beh17, p. 60].

The function's main task consists of a triple nested loop, which is illustrated in Figure 3.6. The three loops are explained briefly in the following:

```
1  i← 0 ;
2  while len >0 do
3      dmu_buf_hold_array_by_bonus(obj, off, len, ..., &dbp);
4      foreach buf in dbp do
5          tocopy←min(remaining in buf, len) ;
6          buffer_offset←off - buf->offset ;
7          while tocopy >0 do
8              thispage← min(PAGE_SIZE, remaining in buf);
9              local_niobuf[i].lnb_page ← kmem_to_page(buf + buffer_offset) ;
10             local_niobuf[i].lnb_page_offset = ... ;
11             local_niobuf[i].lnb_file_offset = ... ;
12             local_niobuf[i].lnb_len = ... ;
13             buffer_offset←buffer_offset + tocopy ;
14             tocopy←tocopy - thispage ;
15             i←i + 1 ;
```

Figure 3.6: Pseudocode for the main loop in `osd_bufs_get_read`

1. `while len > 0`: The outer loop iterates as long as there is data to read according to the requested size. For each iteration, dbufs are requested via `dmu_bufs_hold_array_by_bonus`. After the call returns, *dbp* points to an array of `struct dmu_buf_t`.
2. `foreaech buf in dbp`: The second loop makes sure every buffer is handled, that is, iterating over the array. Because it is not likely that always multiples of the blocksize are

read, the second loop initializes *tocopy* with the minimum of the length within the buffer and the remaining length, so that the whole function returns as soon as *len* bytes are read.

3. `while tocopy > 0`: The inner-most loop continues as long as `buf` contains data. Here, the main reading logic is handled. Line 8 as well as 13 and 14 make sure that no data is read across buffer boundaries. In line 9, the actual reading is performed. As for the write path, `lnb_page` is set to a page-aligned pointer to the data. Subsequently, the other fields of the lnb, which were already discussed for the write path, are set accordingly.

It is to mention that `osd_bufs_get_read` performs additional read-ahead. Per default, this means Lustre parses 1 MiB of buffers from ZFS into local niobufs. Later in the read pipeline, `osd_read_prep` rejects buffers whose `lnb_file_offset` is beyond `eof`[12].

3.4.1.1 Reading compressed data

Normally, ZIO would decompress the data obtained in the outer loop.
Therefore, `dmu_buf_hold_array_by_bonus_compressed` was introduced, which "works analogous to the original function except [...] a flag indicating that the dbufs should be returned in a compressed state." [Beh17, p. 67]. Hence, the respective function must be chosen depending if the data is compressed.

3.4.2 Extracting the chunk headers

When it comes to extracting the metadata from the compressed data, the principle results from the modified write path. Figure 3.7 shows the layout of the stored, compressed data with headers in ZFS.

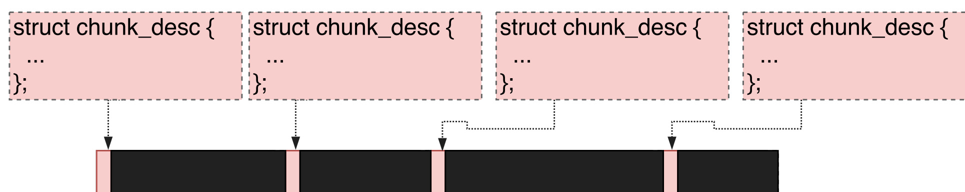

Figure 3.7: Layout of the compressed data with headers in ZFS

When considering the main loop as shown in Figure 3.6, the most suitable approach would be to increase offsets to skip the descriptors after receiving them. The basic principle is illustrated in the modified pseudocode in figure 3.8. However, only the inner loop is shown as the outer loops are not modified.

The basic idea is to change the reading of the data to do it chunk-wise. As shown in Figure 3.7, a chunk descriptor (*cdesc*) is expected at offset 0. The next chunk descriptor then must exist at offset `cdesc->psize` etc. Hence, another loop is introduced in the pseudocode to indicate that data is read as long as the end of the chunk is not reached. The chunk descriptor is eventually copied into a pre-allocated array and the offset within the buffer increased by the chunksize. However, two points must be noted:

[12]end of file; this offset is stored in the files extended attributes during write

```
1  struct chunk_desc *current_chunk←NULL ;
2  while tocopy >0 do
3      thispage← min(PAGE_SIZE, remaining in buf);
4      if cdesc is NULL then
5          current_chunk←pointer to current position ;
6          copy(cdesc_array, cdesc) ;
7          buffer_offset←buffer_offset + sizeof(struct chunk_desc) ;
8      while chunk has data do
9          local_niobuf[i]->lnb_page ← kmem_to_page(buf + buffer_offset) ;
10         buffer_offset←buffer_offset + tocopy ;
11         tocopy←tocopy - thispage i←i + 1 ;
12     cdesc←NULL ;
```

Figure 3.8: Pseudocode for the modified main loop for reading compressed data

1. It is not guaranteed that the memory at the designated header address represents such. This is, for instance, true when the data is corrupted. Since *chunk-wise* means reading as much as indicated by the psize, an arbitrary integer instead of the actual psize might be read, causing a read beyond buffer boundaries. Hence, in the implementation, a way for safely identifying a header will be introduced.

2. The array used in line 6 to copy the chunk descriptor into, must be pre-allocated with sufficient space. However, it is not known in advance how many chunks are expected. For this the best solution is to allocate the maximum, which results from the chunksize and the limit regarding size for an RPC.

4 | Related work

This chapter provides a brief overview over existing work. The focus is the (adaptive) compression of network traffic to reduce network throughput on the one hand and attempts for enhanced interaction between Lustre and ZFS on the other hand.

4.1 Compression of network traffic

When it comes to HPC, communication between multiple computers makes up a good deal of the overall system. However, as seen in Figure 1.1 in the introduction, internet data rates are not growing as fast as storage capacity or computational power. Hence, data reduction techniques are researched not only for Lustre.

This also includes approaches which analyze the on-going network traffic. An example is the US patent titled “Network characteristic-based compression of network traffic” [VGK11]. Compression as done, among others, by the Lempel-Ziv (LZ) family of algorithms is known as *dictionary-based*, which basically means to represent frequent occurences of a string with a shorter key for a dictionary entry containing that data. The basic idea of that patent is to introduce *network optimization devices* which take care of compression and decompression and own such dictionaries. For example, when a device wants to communicate, it may send its data to such a device (A) within the local network first. On the other end of the wire also exists such device (C). “If network optimization device A locates a previously stored string that is also stored at network optimization device C (as part of a compression dictionary shared or independently created [...]), [...] A may transmit the codeword [...] instead of the actual or literal string, to [...] C” [VGK11]. Consequently, when sending a key for a dictionary instead of the actual data, network traffic is ultimately reduced. However, to “efficiently implement a dictionary-based compression scheme, it is desirable to be able to efficiently identify data that was previously seen or sent” [VGK11].

Another approach harnesses the resources of an idling CPU to perform compression before transmitting the data [Wel+11]. The results showed that “compression is useful on slower networks, such as 1 Gbit Ethernet” and it is best to “avoid computationally expensive compression algorithms, such as bzip2 and [...] zlib” [Wel+11, p. 6]. Also, it was found that skipping decompression on the server-side benefits performance, as already discussed in this thesis.

As discussed before, the choice of a compression algorithm based on analysis of the data ultimately results in the best performance gain. Krintz and Sucu [KS06] presented a system called the *adaptive compression environment* that applies compression on-the-fly based on estimation

techniques "to make short-term forecasts of compressed and uncompressed transfer time at the 32 Kb block level". Eventually, transfer performance could be enhanced by 8 to 93 percent, depending on different factors.

4.2 Enhanced interaction between Lustre and ZFS

4.2.1 End-to-End data integrity for Lustre and ZFS

As mentioned before, one of ZFS' key features include end-to-end data integrity. However, this only relates to ZFS' interface layer on the one end and the underlying disk on the other. Lustre also includes data integrity checks using checksums to make sure the data was transferred correctly over the network.

As part of the High Productivity Computing Systems program in 2009, whose goal was "developing a new generation of economically viable high productivity computing systems for national security and industry" [Wik17] in order to meet the programs objective, "Lustre needs to have end-to-end resiliency" [DDD09, p. 3] with one end being the Lustre client and the other end the underlying disk. It was meant to be implemented as "efficient end-to-end data integrity verification by combining the Lustre data checksumming with the ZFS on-disk data checksumming" [DDD09, p. 3]. For example, the ZFS backend could skip verifying the data while reading and let the Lustre client deal with it.

4.2.2 Ongoing work

Regarding the interaction between Lustre and the underlying ZFS file system, there is also ongoing work which is worth to mention. This includes [Str17]:

- **80 times faster RAIDZ3 reconstruction**: The different levels of RAIDZ numbered one to three describe the number of hard drive failures within a ZFS pool for which it is still possible to recover the data using parity calculations. Gvozden Nešković [Nes16] proposed a patch for ZFS to fasten the reconstruction. "What this means [...] is that we can dedicate fewer cores to parity and reconstruction processing, and that saves costs building and running of Lustre clusters with a ZFS back end file system using RAIDZ for high availability and reliability" [Str17].
- **Lustre snapshots**: Fan Yong of Intel's High Performance Data Division (HPDD) [YJ17] is adding the capability to create mountable snapshots of Lustre setups with ZFS backend for any moment in time via a CLI.
- **Enhanced performance for metadata**: For metadata, Lustre's own ldiskfs has been a little faster than ZFS. Alexey Zhuravlev (also Intel HPDD) [Zhu16], contributed improvements for metadata performance on ZFS to get rid of known bottlenecks.

5 Implementation

In this chapter, it is described how the changes following the discussions in the Design were implemented into the existing Lustre sourcecode. It is divided into three sections. First, preparations for both, reading and writing, are described. Then, the modified write and read path are depicted each.

5.1 Preparations

5.1.1 Indicating compressed I/O

Future Lustre setups may involve communication between clients understanding compression and those who do not with the same OSS. Hence, in multiple situations, a way to determine in which group the client fits is neccessary. The most basic approach seems to check for the presence or absence of `struct chunk_desc *` on the RPC. However, in remembrance of Section 3.2.3, it is not possible to distinguish between a header missing by intent or by error.

Therefore, an additional flag was added to `enum obdo_flags`: `OBD_COMPRESSED_IO`, which is set on the body flags of the RPC within `compressed_osc_brw_prep_request` using `body->oa.o_flags |= OBD_COMPRESSED_IO`. Hence, within `tgt_io_data_unpack` the presence of the flag is checked and the absence of the chunk descriptors handled appropriately, as seen in Listing 5.1.

```
1  if (tsi->tsi_ost_body->oa.o_flags & OBD_COMPRESSED_IO) {
2    cdesc = req_capsule_client_get(tsi->tsi_pill, &RMF_CHUNK_DESC);
3
4    if (unlikely(cdesc == NULL)) {
5      /* (lines omitted): throw error and return protocol error */
6      RETURN(-EPROTO);
7    }
8
9    nchunks = req_capsule_get_size(tsi->tsi_pill, &RMF_CHUNK_DESC,
10     RCL_CLIENT) / sizeof(struct chunk_desc);
11
12   LASSERT(nchunks > 0);
13 }
```

Listing 5.1: Check for OBD_COMPRESSED_IO flag within tgt_io_data_unpack

A

dditionally, the software is enabled to handle the absence of the flag accordingly. For instance, assume the data is compressed and a client sends a read RPC for that data, without

OBD_COMPRESSED_IO being set. This is an indication for either compression turned off or a lack of the client-side compression feature. In both cases, the server can use the chunk descriptor extracted during read and decompress the data in the backend, eventually sending uncompressed data back to the client.

5.1.2 Introduce chunk sizes to PTLRPC

Following Sections 3.2.2 and 3.2.3, the chunksize is required to be set according to the record size of ZFS honoring the case of uncompressed data. For both, client and server to commit to the common chunksize, constants were added to PTLRPC[1] (see Figure 5.2). The constant is used for all further calculations done in both, server and client.

```
1 #define PTLRPC_MIN_CHUNK_SIZE (131072 - sizeof(struct chunk_desc))  /*
    ZFS record size for now */
2 #define PTLRPC_MAX_CHUNK_SIZE (PTLRPC_MAX_BRW_SIZE - sizeof(struct
    chunk_desc))
3 #define PTLRPC_MAX_BRW_CHUNKS (PTLRPC_MAX_BRW_SIZE /
    PTLRPC_MIN_CHUNK_SIZE)
```

Listing 5.2: Introduction of chunk size constants for PTLRPC

5.1.2.1 Adapting the client to use PTLRPC_MIN_CHUNK_SIZE

Shortly to be noted, the client had to be adapted to use PTLRPC_MIN_CHUNK_SIZE instead of a magic number. Moreover, in the original prototype, the chunksize was required to be a multiple of PAGE_SIZE. Instead of iterating over pages, the pages sizes are first added together and then the total sum is used accordingly. The basic changes are illustrated in Figure 5.1.

```
1 tocpy←sum all page sizes ;
2 while tocpy > 0 do
3 |   parse next page into chunk of size PTLRPC_MIN_CHUNK_SIZE ;
4 |   if page has remaining data then
5 |   |   carry for next chunk ;
6 |   tocpy←tocpy - PTLRPC_MIN_CHUNK_SIZE ;
7 foreach chunk do
8 |   dst←LZ4_compress_default(chunk) ;
9 |   split dst into pages ;
```

Figure 5.1: compress_chunks (client): New function to assemble chunks based on the chunksize not dividable by PAGE_SIZE

5.1.3 Passing chunk descriptors from the target handler to OSD-ZFS

As discussed in Section 3.3, the target handler handles the incoming RPC, unwraps the body and pre-processes the data. Then, in particular tgt_brw_write and tgt_brw_read, invoke functions within the underlying layers and pass the parameters they need. For obd_preprw, additionally

[1] lustre/include/lustre_net.h

the chunk descriptors as well as the number of chunks are required. They were introduced as `int *nr_chunks` (pointer to int) and `struct chunk_desc **cdesc` (pointer to pointer to `struct chunk_desc`) respectively. Later, `dt_bufs_get`, an operation on the specific `struct dt_object *d`, is called. It was changed to accept an additional `int compressed` to call either `d->do_body_ops->dbo_bufs_get` or `d->do_body_ops->dbo_bufs_get_compressed`. This decision was made due to the fact, that `OSD-ZFS` as well as `OSD-ldiskfs` both implement `struct dt_body_operations` pointed to by `do_body_ops`. Following of that, for OSD-ZFS the interface now provides `.dbo_bufs_get_compressed = osd_bufs_get_compressed` while OSD-ldiskfs passes `.dbo_bufs_get_compressed = NULL`. This way, an implementation of similar features for OSDs other than ZFS is possible with less effort.

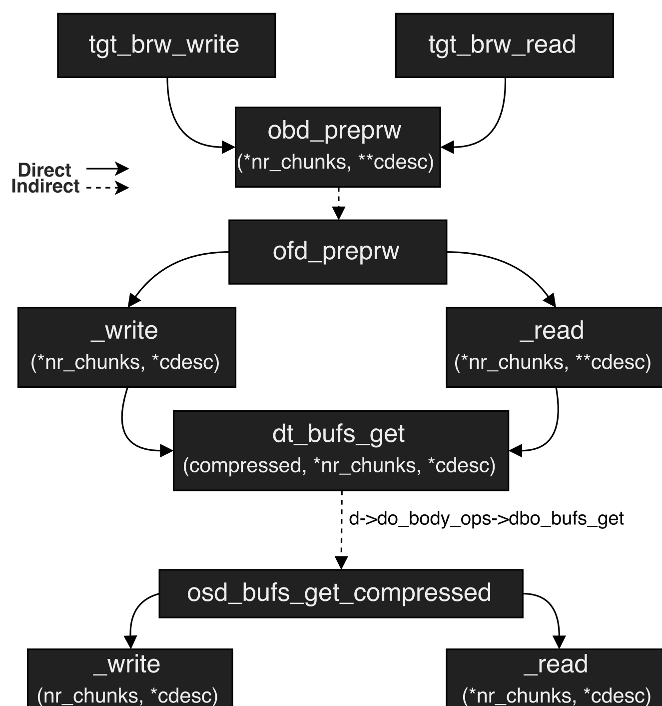

Figure 5.2: Invoked functions during read and write with new parameters

Figure 5.2 depicts the relationship of the invoked functions with their new parameters (for the meaning of the arrows, compare Figure 2.4). The need for passing pointers evolves from the different roles for reading and writing respectively:

- For writing, the data is passed from the client, so that `nr_chunks` and `cdesc` are already known. Hence, in `ofd_preprw_write` (`cdesc`) and `osd_bufs_get_compressed` (`nr_chunks`) the pointers are dereferenced.
- For reading, memory for `cdesc` must be allocated (which will be discussed in Subsection 5.3.2). Later, when extracting the chunk headers, `nr_chunks` is incremented according to the headers pushed into `cdesc` in `osd_bufs_get_compressed_read`.

5.1.4 Identifying a chunk header

A suitable method for determining if the `void *` pointer represents a chunk header is needed during read, especially when it comes to error handling and recovery (compare Subsection 3.4.2).

Hence, a new field was introduced for `struct chunk_desc`, `__u32 flags` (compare Appendix B.1). Two of them are used for checking if a pointer is a `struct chunk_desc *`: `CHUNK_HAS_HDR = 0x11110000` and `CHUNK_DESC_MASK = 0xffff0000`. The check is performed by checking if `((struct chunk_desc *)buffer)->flags & CHUNK_DESC_MASK == CHUNK_HAS_HDR`. The propability that the value leads to a false-positive is only $\frac{1}{16^4}$, which should be sufficient.
Additionally, as an example, `CHUNK_COMPRESSED_LZ4 = 0x00000010` was defined. It is suggested to replace `__u32 algo` with the usage of flags in the long-term.

5.2 The modified write path

5.2.1 Determine which chunk is written

When `osd_bufs_get_write` is invoked, it must be known which chunk actually is written to. Currently, the client sends each chunk in a disctinct rnb. However, no immanent association between a chunk descriptor and an rnb exists. For this task `poff` (*phhysical offset*) and `loff` (*logical offset*) were added to `struct chunk_desc` (see Appendix B.3). The function `cdesc_for_off`, whose implementation can be seen in Appendix B.1, was introduced to link chunk descriptor and rnb by using the offsets as unique identifiers. The one-to-one relationship between rnb and chunk, however, is not neccessary for the implementation. The only constraint is that a chunk does not span multiple rnbs.

5.2.2 Extracting the largest psize

While implementing the described approach, a kernel panic was triggered when the first chunk written was not the largest regarding `psize`. This happened because `dmu_request_compressed_arcbuf(obj, psize, lsize)` always returns the same size, that is, the size of the first use, ultimately writing beyond buffer boundaries when a subsequent chunk is larger. In the course of this thesis, it was not identified whether this is a bug within the work of [Beh17] or within ZFS.
As a temporary workaround, a function `cdesc_max_psize` was introduced, returning the maximum psize among all chunks (Appendix B.2). Hence: `dmu_request_compressed_arcbuf(obj, max_psize, lsize)`.

5.2.3 osd_bufs_get_compressed_write

To apply the header to each chunk during the allocation of buffers, a new function, `osd_bufs_get_compressed_write`, was introduced which is called when `OBD_COMPRESSED_IO` is set. Its prototype is shown in Listing 5.3. The full sourcecode is found in Appendix B.4.

```
1 static int osd_bufs_get_compressed_write(const struct lu_env *env,
    struct osd_object *obj, loff_t off, ssize_t len, struct niobuf_local
     *lnb, int nr_chunks, struct chunk_desc *cdesc)
```

Listing 5.3: Prototype of `osd_bufs_get_compressed_write`

In the following, the functioning of `osd_bufs_get_compressed_write` is illustrated by means of the sourcecode.

```c
while (len > 0) {
  curr_cdesc = cdesc_for_off(off, cdesc, nr_chunks); /* get current
    chunk descriptor */
  ...
  abuf = dmu_request_compressed_arcbuf(obj->oo_db, max_psize + hdr_sz,
   max_t(int, PTLRPC_MIN_CHUNK_SIZE + hdr_sz, curr_cdesc->lsize +
   hdr_sz)); /* requst arcbuf of the appropriate size */
  ...
  chunk_remaining = curr_cdesc->psize + hdr_sz;
```

Listing 5.4: `osd_bufs_get_compressed_write`: Request arcbufs and prepare parsing buffers

The outer-most loop as seen in Listing 5.4 is designed under the assumption that an rnb only consists of full chunks. Hence, it would run with $len \in \{n \cdot chunksize, (n-1) \cdot chunksize, ..., 1 \cdot chunksize\}$ and $off \in \{0, chunksize, 2 \cdot chunksize, ..., n \cdot chunksize\}$ respectively. In line 2, the current chunk descriptor is retrieved using the offset as described above, that is, for $off = 0$ the chunk descriptor with $loff = 0$ is returned. Following that, an arcbuf is requested by `dmu_request_compressed_arcbuf(obj, psize, lsize)`. Two things must be noted: First, `max_t` is only relevant when the chunk is smaller than the chunksize to ensure equal-sized blocks. Second, the additional space seen in Figure 5.3 is added to the maximum `psize`.

```c
while (chunk_remaining > 0) {
  plen = min_t(int, chunk_remaining, PAGE_SIZE);
  if (off_in_buf == 0) {
    lnb[i].lnb_data = abuf; /* remember arcbuf for commit */

    plen -= hdr_sz; /* page is smaller due to header */
    lnb[i].lnb_page_offset = hdr_sz; /* adapt offset for header */

    memcpy(abuf->b_data, curr_cdesc, hdr_sz); /* copy header into buffer
     */
  } else
    ...
  lnb[i].lnb_file_offset = off;
  lnb[i].lnb_len = plen;
  len -= min_t(int, len, plen);
  off += min_t(int, chunk_remaining, PAGE_SIZE);
  chunk_remaining -= min_t(int, chunk_remaining, PAGE_SIZE);
  off_in_buf += PAGE_SIZE;
  i++;
  npages++;
}
```

Listing 5.5: `osd_bufs_get_compressed_write`: Process received arcbuf into lnbs

In contrast to the original implementation, the inner loop as seen in Listing 5.5 iterates as long as $chunk_remaining > 0$, which turned out to be more reliable than using the arcbufs size. Most lines (2, 14 - 17)) are important to ensure the correct sizes and offsets are used for each local niobuf. The most notable task is performed in the true branch (lines 4 to 9). For the first page of each chunk,

1. the arcbuf is remembered for commitment, as done within the original function (compare Figure 3.2),
2. the offset is set to the header size,
3. the header size is subtracted from the length and
4. the header is `memcpy`ied to offset 0 of the arcbuf

Alltogether, when PTLRPC eventually writes data into these niobufs, it honors the presence of the header. An illustration for writing data for a single chunk, is given in Figure 5.3.

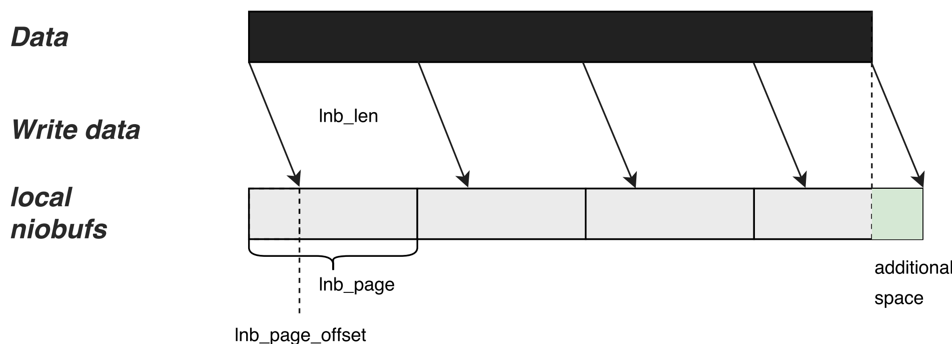

Figure 5.3: Force PTLRPC to write the data beyond the header using `lnb_page_offset` and `lnb_len`

5.2.4 Committing data into ZFS

Commiting the arcbuf into ZFS within the modified path comes down to two tasks:

- Set the blocksize to the chunksize
- Use `dmu_assign_compressed_arcbuf`

Both can be performed in `osd_write_commit` with minor changes, therefore no additional function was introduced.

5.2.4.1 Setting the blocksize

For the latter, the original `osd_grow_blocksize` was split into `osd_grow_blocksize` and `osd_set_blocksize` (see Appendix B.5). When `OBD_COMPRESSED_IO` is not set, the former simply calls the latter and, therefore, performs the original logic. Otherwise, it is invoked as `(void)osd_set_blocksize(obj, oh, PTLRPC_MIN_CHUNK_SIZE + sizeof(struct chunk_desc))`, respectively.

5.2.4.2 `dmu_assign_compressed_arcbuf`

Actually committing the arcbuf into ZFS comes down to adding a third case (compressed arcbuf) to the existing ones (arcbuf, anonymous page; see Figure 3.3 in Section 3.3.2). The branch can be seen in Listing 5.6.

Since the chunksize does not equal the blocksize yet because it was decremented by `sizeof(struct chunk_desc)` to ensure enough space for the header on the same block as the data (compare Subsection 3.2.3), it is now adjusted accordingly. However, adding the header size to each chunk

```
1   if (compressed) {
2     lnb_cdesc = cdesc_for_off(lnb[i].lnb_file_offset,
3       cdesc, nchunks);
4     LASSERT(lnb_cdesc != NULL);
5     ...
6     dmu_assign_compressed_arcbuf(obj->oo_db,
7       lnb_cdesc->loff + ((lnb_cdesc->loff / cdesc->lsize) * sizeof(
    struct chunk_desc)),
8       lnb_cdesc->psize + sizeof(struct chunk_desc),
9       lnb[i].lnb_data, oh->ot_tx);
```

Listing 5.6: `osd_write_commit`: A third branch for committing compressed arcbufs was added

is not sufficient. Whilst the offset of the first chunk aligns with the block boundaries within ZFS, the second block is shift by the header size, the third block is shift two times the header size (with one shift resulting from the previous chunk) etc. Hence, `((lnb_cdesc->loff / cdesc->lsize) * sizeof(struct chunk_desc))` is added to the respective logical offset, which is

$$\{0, sizeof(struct\ chunk_desc), ..., nchunks - 1 \cdot sizeof(struct\ chunk_desc)\}$$

(sum of space taken by all preceding headers) respectively.

5.3 The modified Read path

5.3.1 Preparing the reply

Other as for the write path, reading data demands a reply from the server to the client. The reply is prepared in `tgt_handle_request0`. However, read-RPCs have a fixed format, which means the sizes of the fields must already be set. For the chunk descriptors, however, it is not known yet. Hence, the maximum is assumed and set (see Listing 5.7). This way, it is ensured enough space is reserved, no matter how many chunk descriptors follow.

```
1 if (req_capsule_has_field(tsi->tsi_pill, &RMF_CHUNK_DESC, RCL_SERVER))
2   req_capsule_set_size(tsi->tsi_pill, &RMF_CHUNK_DESC, RCL_SERVER,
3   PTLRPC_MAX_BRW_CHUNKS * sizeof(struct chunk_desc));
4
5   rc = req_capsule_server_pack(tsi->tsi_pill);
```

Listing 5.7: `tgt_handle_request0`: The field `&RMF_CHUNK_DESC` is set to the maximum size for the reply

Later, when the exact size is known (that is, after `obd_preprw` returns), the field is shrunk to the exact size (see Listing 5.8).

5.3.2 Allocating memory for the headers

In the write path, the headers are provided by the client via an RPC. In the opposite direction, the client awaits the descriptors wrapped in the reply. For the sake of extracting the headers, sufficient memory must be allocated. Hence, the `tgt_handler` declares `struct chunk_desc`

```
rc = obd_preprw(tsi->tsi_env, OBD_BRW_READ, exp, &repbody->oa, 1, ioo,
    remote_nb, &npages, local_nb, &nchunks, &cdesc);
  ...
  LASSERT(ergo(nchunks > 0, cdesc != NULL));
  ...
  req_capsule_shrink(tsi->tsi_pill, &RMF_CHUNK_DESC, nchunks * sizeof(
    struct chunk_desc), RCL_SERVER);
```

Listing 5.8: `tgt_brw_write`: The field `&RMF_CHUNK_DESC` is set to the exact size

`*cdesc = NULL` first and passes `&cdesc` to `obd_preprw` (compare Figure 5.2). Eventually, in `ofd_preprw_read`, the `NULL` pointer is replaced via a memory allocation (see Listing 5.9). Similar to the preparation of the reply, it is not yet known how many chunk descriptors are expected, therefore, again, the maximum is allocated.

```
if (compressed) {
  OBD_ALLOC_LARGE(*cdesc, PTLRPC_MAX_BRW_CHUNKS * sizeof(struct
    chunk_desc));

  LASSERT(*cdesc != NULL);
}
```

Listing 5.9: `ofd_preprw_read`: Allocating memory for extracting the chunk headers

Since the memory is needed until the read pipeline finished, the target handler is responsible of freeing the memory after it is done (`OBD_FREE_LARGE(cdesc, PTLRPC_MAX_BRW_CHUNKS * sizeof(struct chunk_desc))`).

5.3.3 osd_bufs_get_compressed_read

The main reading logic is performed in a new function, `osd_bufs_get_compressed_read`. Its prototype is depicted in Listing 5.10. In the following, the most important parts of the sourcecode are presented. The complete code is shown in Appendix B.6.

```
static int osd_bufs_get_compressed_read(const struct lu_env *env, struct
     osd_object *obj, loff_t off, ssize_t len, struct niobuf_local *lnb,
     int *nr_chunks, struct chunk_desc **cdesc)
```

Listing 5.10: Prototype of `osd_bufs_get_compressed_read`

The function is invoked in place of `osd_bufs_get_read` when `OBD_COMPRESSED_IO` is set. However, other as for the write path, determining whether the data is compressed or not is subject to the backend. Therefore, the flag here just indicates that the client understands compressed data and, hence, is able to decompress it. As suggested before, it is possible to decompress the data on the server when the data is compressed but `OBD_COMPRESSED_IO = 0`. The actual compression of the data is indicated by the field `db_compressed` of the dbuf [Beh17, p. 64] and checked as seen in Listing 5.11. However, it is currently always false and, following of that, the fallback is disabled until the cause is known.

```
compressed = obj->oo_db->db_compressed;

if (0 /* !compressed TODO */)
  RETURN(osd_bufs_get_read(env, obj, off, len, lnb));
```

Listing 5.11: `osd_bufs_get_compressed_read`: Fallback to `osd_bufs_get_read` when the data is not actually compressed

The basic idea of the reading of pre-compressed data is the same as for the unmodified code shown in Figure 3.6. The main difference is that the data is processed chunk-wise.
The outer-most loop `while` (*len* > 0) remains unmodified. However, it calls the equivalent function for retrieving dbufs in their raw state, `dmu_buf_hold_array_by_bonus_compressed`. The inner-most loop, that is the third level of nesting, holds the main logic.

```
while (tocpy > 0) {
  if (curr_chunk == NULL) {
    curr_chunk = (struct chunk_desc *)(dbp[i]->db_data); /* try to read
    header */
    if (((curr_chunk->flags & CHUNK_DESC_MASK) == CHUNK_HAS_HDR)) {
      chunk_bn = 1; /* chunk begin */
      chunk_rmng = curr_chunk->psize;
      chunk_rmng_l = curr_chunk->lsize;
      memcpy(&((*cdesc)[nchunks++]), curr_chunk, hdr_sz);
    } else
      ...
  }
```

Listing 5.12: `osd_bufs_get_compressed_read`: Reading data chunk-wise and extracting the header when a chunk begins

During each iteration, the true branch of the `if` statement (Listing 5.12, line 2) refers to the fact that currently no chunk is associated with the data. Hence, it is tried to interpret the current position within the buffer as a pointer to `struct chunk_desc` (compare Figure 3.7). It is then validated using the flag introduced in Subsection 5.1.4. In line 8, when the read data actually represents a header, it is pushed into the pre-allocated array.

```
if (chunk_rmng > 0) {
  if (chunk_bn) { /* begin of chunk */
    lnb->lnb_page_offset = hdr_sz;
    thispage -= hdr_sz;
  }
  /* fill lnb */
else
  /* fill empty lnb */
```

Listing 5.13: `osd_bufs_get_compressed_read`: Adjusting the fields of the local niobuf after reading the header

As done for the write path, the local niobufs page offset as well as the current pages size are adapted accordingly for the first page within the chunk (see Listing 5.13). One might argue that, instead of adjusting the offset within the page, the header size can be added to the offset

to skip the more complex logic. However, the Kernel `AND`s the offset with `PAGE_MASK`, leaving only the higher bits representing the pages address unchanged while zeroing the offset.
Lines 5 and 7, respectively, are omitted. Nevertheless, they represent an important requirement. Since the dbuf is of size `chunksize + header size`, which is the `lsize` of the chunk plus the header, it is only desired to parse memory into local buffers when `offset in buf` < `psize`. It seems likely that the remaining iterations can be skipped when all data was read. However, it is required to return exactly as much niobufs as indicated by `len` for subsequent read path. Therefore, empty lnbs are assembled.

The last two aspects are adjusting the working variables for the next iteration to start with the correct values on the one hand and the adjustment of the offsets on the other hand. The latter is seen in Listing 5.14. It turned out that the usage of `poff` and `loff` associated with a chunk header is more reliable than handling different edge cases for incrementing the offset. Hence, after the end of the chunk was read, the offset is set to the next address where a chunk is expected.

```
1 if (curr_chunk != NULL && chunk_rmng_l == 0) {
2   off = max_t(int, curr_chunk->loff
3     + curr_chunk->lsize
4     + ((curr_chunk->loff / curr_chunk->lsize) * hdr_sz),
5     PTLRPC_MIN_CHUNK_SIZE + hdr_sz);
6   curr_chunk = NULL;
7   len -= tocpy;
8   tocpy = 0;
9 }
```

Listing 5.14: `osd_bufs_get_compressed_read`: Adjusting offsets after the chunks end

6 Evaluation

In this chapter, it is presented how the implementation was tested. In the first section, the correct transmission of the data is proved and it is shown, that the chunk descriptors are available after reading. In the second section, benchmarks regarding network transfer and read performance are presented.

6.1 Correctness

For proving the correctness of the data transfer, the following points must be considered:

1. The data received from the server on the read path must be equal to the data originally written.
2. The reply of the server must contain the `struct chunk_desc *` and each descriptor must equal the counterpart transferred during write.

In general, the Lustre-internal macro `CERROR` was added to relevant lines of code for tracing the function calls and the values of variables at given moments. `CERROR` is a wrapper for `printk` and logs messages to the standard error stream `stderr` of the kernel.

6.1.1 Transmission of data

When data is written to the mountpoint and the client performs compression, the resulting data fits into one of three categories: Uncompressed or insufficiently compressed as well as compressed data (compare Subsection 3.2.3). For the former two, a modification to the client code was made as shown in Listing 6.1. The `if` statement acts as a switch to replace the copression with `memcpy`ing the data to the destination buffer. The tests in this subsection were performed using two clients, using the true and false branche respectively.

```
1 comprsd = LZ4_compress_default(src, dst, tmp_cdesc[c].lsize, tmp_cdesc[c
      ].lsize, wrkmem);
2 if (1) {
3   memcpy(dst, src, tmp_cdesc[c].lsize);
4     comprsd = tmp_cdesc[c].lsize;
5 } else
6     comprsd = LZ4_compress_default(src, dst, tmp_cdesc[c].lsize,
     tmp_cdesc[c].lsize, wrkmem);
```

Listing 6.1: `compress_chunks`: Modified to return uncompressed data

6.1.1.1 Uncompressed data

For this test case, it turned out human-readable data is most convenient. Thus, in this example, parts of *Faust* by Johann Wolfgang von Goethe were used for debugging purposes. A test for a 16 KiB excerpt is shown in Listing 6.2.

```
# diff -s /mnt/lustre/client2/faust files/faust
Files /mnt/lustre/client2/faust and files/faust are identical
```

Listing 6.2: Test of the correctness for uncompressed data (first chunk)

Overall, the writing of uncompressed data works as intended. However, due to the chunksize not being dividable by `PAGE_SIZE`, additional pages must sometimes be added because of the fact that the space reserved for the header is *carried* to the next page, which might require an additional page at the end of the chunk. Due to the limit of 256 pages per RPC, further work within the client is required to properly deal with uncompressed data, since those additional pages make the RPC exceed this limit.

6.1.1.2 Compressed data

In the current implementation, the client returns the data in its compressed form. Eventually, an implementation detail was exploited for testing the data anyway: The client overwrites the original pages. As an example for the idea of the test, assume data of 12 KiB (lsize) and 8 KiB (psize). Then, for instance, `cat /mnt/lustre/client/file` returns a file consisting of:

- Compressed data at offset $[0, 8192]$
- Uncompressed data at offset $(8192, 12288]$

That means, since in that example the file is smaller than the chunk size, the bytes within $[0, 8192]$ must equal the read data, that is, the first chunk.

```
# cd /root
# hexdump /mnt/lustre/client/faust > orig
# hexdump /mnt/lustre/client2/faust > now
# diff -ay --left-column orig now | grep -v "("
```

Listing 6.3: Test of the correctness for the first chunk

Listing 6.3 demonstrates the commands performed after writing the data. For the first `hexdump`, the modified original pages are read while the second usage with a distinct mountpoint forces Lustre to request the file from the server. The usage of `diff` piped through `grep` skips equal lines. In summary, addresses 0_{16} up to $2CCD_{16} = 11469_{10}$ are equal. The correctness follows from the output of the compressed size via `CERROR`, which is also 11469.
For subsequent chunks, the same approach can be adapted but requires manually parsing the output since each chunk is followed by uncompressed data up to its `lsize`, respectively.

6.1.2 Transmission of the chunk descriptors

For testing if the chunk descriptors were correctly transferred within the reply on the read path, the function osc_brw_fini_request was modified as depicted in Listing 6.4.

```
1  struct chunk_desc *cdesc;
2  int nchunks = 0, c = 0;
3
4  if (req_capsule_has_field(&req->rq_pill, &RMF_CHUNK_DESC, RCL_SERVER))
     {
5    cdesc = req_capsule_server_get(&req->rq_pill, &RMF_CHUNK_DESC);
6    nchunks = req_capsule_get_size(&req->rq_pill, &RMF_CHUNK_DESC,
    RCL_SERVER) / sizeof(struct chunk_desc);
7
8    for (; c < nchunks; cdesc++, c++)
9      CERROR("chunk[%d]: lsize=%d, psize=%d, flags=%d, cksum=%x\n", c,
    cdesc->lsize, cdesc->psize,  cdesc->flags, cdesc->cksum);
10 }
```

Listing 6.4: osc_brw_fini_request: Reading the chunk descriptors during read

The function is invoked at some point after the reply reaches the client. It is checked if the reply contains the field for the chunk descriptors (line 4), then it is unwrapped and the number of chunks is calculated from the size of that field (lines 5 and 6). Eventually, for each chunk, the members of the struct are written to stderr. Similar code blocks were inserted within the target handler and in osc_brw_prep_request.
In conclusion, at each given moment, the output is equal.

6.2 Benchmarks

Since this thesis aims to interconnect the previous work in [Fuc16] and [Beh17], in the following a first clue for the overall performance for client-side compression is given. However, since the implementation is still a prototype in an early phase, the informative value with respect to the final implementation is limited.

Each benchmark was performed for 1, 7 and 10 MiB of data respectively. These sizes result from the fact, that for some data sizes the implementation still causes the kernel to panic, which is for instance true for 5 MiB but not for 7 MiB. However, when no errors occur, the filesystem works as intended. It is to be investigated what causes the errors for certain sizes.

Each measurement was repeated 5 times and averaged for eliminating spikes.

6.2.1 Network transfer size

The following test measures the network transfer size from the client to any other device after writing the data. Since no outgoing internet connection was established, only communication from and to the MGS/MDT and OST was taken into concern. However, this includes actual

bulk data transfer as well as any other communication. For gathering the data, the bash script shown in Listing 6.5 was used. The result is shown in Figure 6.1.

```
# stat="cat /proc/net/dev | grep enp0s3"; before=$(eval $stat | awk '{
    print $10}'); cp test/10M /mnt/lustre/client/t10; sleep 60; after=$(
    eval $stat | awk '{print $10}'); echo "$after-$before" | bc
```

Listing 6.5: Script used to extract network statistics from `/proc/net/dev`

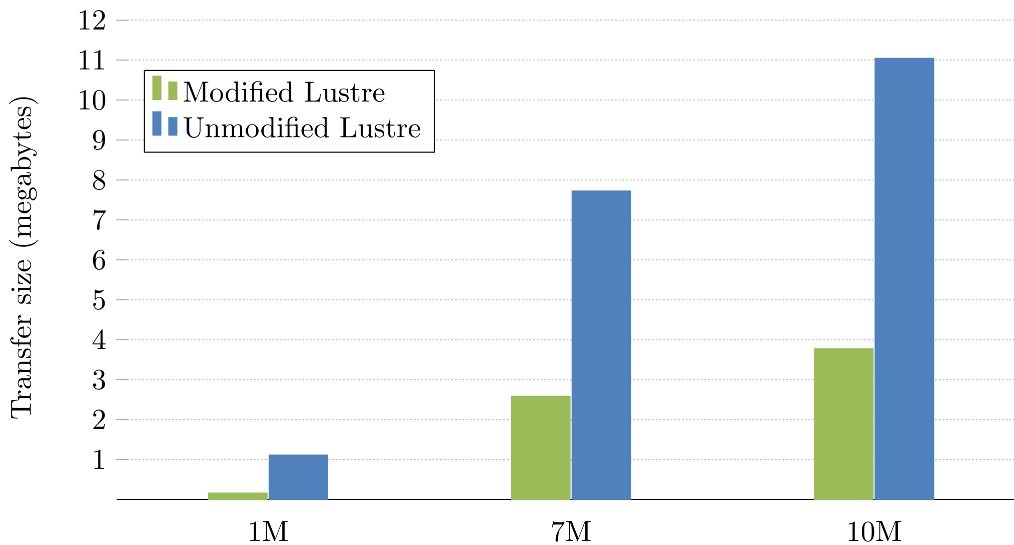

Figure 6.1: Network throughput for 1 MiB, 7 MiB and 10 MiB of climate data for the unmodified Lustre and after performing compression in the modified Lustre

For 10 MiB of data, the original Lustre client transferred 11,042,937 bytes, which includes a huge amount of communication overhead. For the same data, the modified client was able to reduce the transferred size by almost 66 % using compression:

$$1 - \frac{37744306}{110429372} \approx 0.6582$$

Overall, the results show that the client-side compression saves a fair amount of data for the transfer over the network. This is promising with respect to the future work introducing the actual *adaptiveness* of the compression.

6.2.2 Read performance

In this test, the read performance was measured using `dd` (see Listing 6.6). After each read, the cache was cleared before performing the next read. Also, for the modified Lustre, `CERROR` was re-defined to expand to `(void)(0)` since writing to the kernel logs comes with significant expenses which would falsify the outcome. The results are shown in Figure 6.2.

```
# rm /tmp/test; dd if=/mnt/lustre/client/test of=/tmp/test oflag=direct
```

Listing 6.6: Command used for measuring the read performance

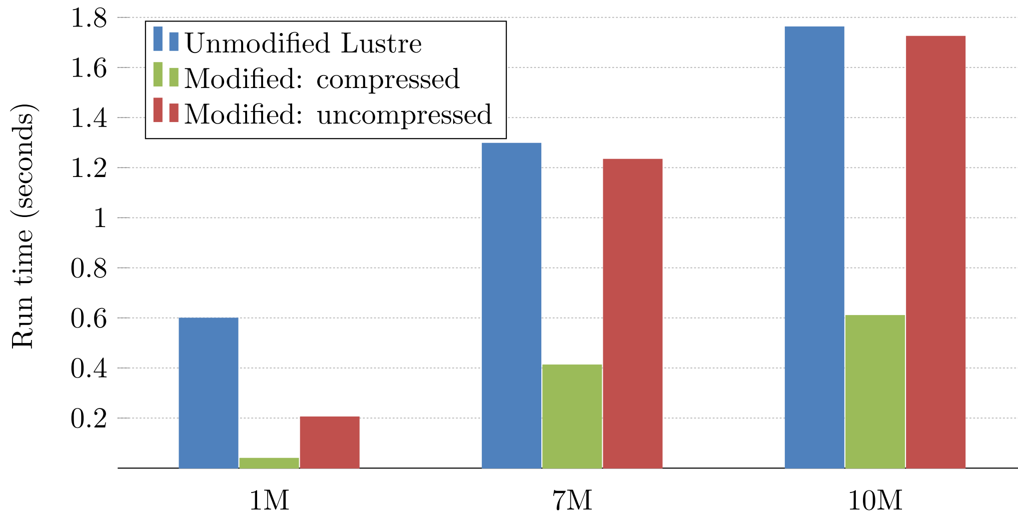

Figure 6.2: Read time for 1 MiB, 7 MiB and 10 MiB of climate data for the unmodified and modified Lustre respectively

Two aspects should be noted:

1. First, when comparing the unmodified Lustre to the modified read path for uncompressed data, no performance regressions occured. Rather, the modified version is a little faster in all test cases. While for 7 and 10 MiB the difference is minor, for 1 MiB it is unexpecpectedly significant. However, it is not clear whether this is due to the different reading approach or some unknown factor.

2. The read path with compressed data is significantly faster than the unmodified one. However, it must be noted that in the current implementation, no decompression is done. Since decompression also comes at a cost, it must be considered that the read path involving decompression will take longer. Nevertheless, decompression will not nullify the gap. It is likely, that the major factor for the difference is the transfer of less data over the network (compare Figure 6.1).

7 | Conclusion and future work

This chapter concludes the thesis. In the first section, a summary of the implemented new features with respect to the overall goal is given. Furthermore, ideas for future work which were out of the scope of this thesis are depicted.

Conclusion

This thesis ultimately aimed to interconnect the previos work regarding the introduction of client-side compression within Lustre:

- The fundamentals were presented by Anna Fuchs in her thesis "Client-side Data Transformation in Lustre" ([Fuc16]). The idea is to split data stripes into smaller chunks, which are compressed independently and, hence, allow to exploit the parallelism of Lustre as best as possible. Foremost she described the requirements and evaluated the space of feasible solutions. Also, a prototype for compression using LZ4 was presented.

- For storing pre-compressed data, it was concluded that the existing structures of the underlying file systems (ZFS, ldiskfs) must be connected as tightly as possible with the new features. An approach for ZFS was introduced by Niklas Behrmann in his thesis "Support for external data transformation in ZFS" ([Beh17]), ultimately aiming to allow Lustre to store compressed chunks as if they were compressed by ZFS.

The need for compression evolves mainly from I/O as performance bottleneck, the high cost of (fast) storage systems and – for distributed file systems such as Lustre – from the need to transfer data over the network. Following of that, the communication between Lustre and ZFS regarding pre-compressed data must be as efficient as possible.

In this thesis, the requirements evolving from the previous work were presented. This includes the storage of metadata (describing the compression) for each chunk as well as adapting the read and write paths, respectively, to handle pre-compressed data, that is receiving it from the client on the one side, adding the metadata to the data and handing it to ZFS on the other hand and vice versa. In the main part, the best-suited layer for introducing the new functionality was evaluated. Eventually, it showed that writing the metadata as a header while allocating buffers on the target, before receiving the data, was the best approach for the write path. While reading, the headers are extracted on-the-fly. The new functionality was tailored to integrate as tightly as possible with the existing structures, allowing to implement the storage of pre-compressed data for other local file systems as well.

In the end it was shown, that the size of data transferred over the network could be reduced by a fair amount in this early prototype. What is more, no performance regressions were measured compared to the original implementation. Rather, reading compressed data is faster, even when taking the lack of decompression into account.

Future work

As Anna Fuchs conclued in her thesis, "Introducing such extensive mechanisms like compression into a stable complex distributed system is an ambitious project" [Fuc16, p. 76]. In the course of this thesis it turned out, that the introduction of handling pre-compressed data includes lots of edge cases which must be taken into account. As of this thesis, subsequent tasks include:

- It must be evaluated why for pre-compressed data blocks the appropriate flag `db_compressed` within ZFS is always false and the temporary work-around (always assuming compressed data) has to be removed.
- For some data sizes such as 5 MiB, the file system still causes the kernel to panic. However, this is not true, for instance, for 1, 7 or 10 MiB. It must be tracked down, which part of Lustre triggers the panic.
- It should be evaluated which fields of the header are really needed, since it is added to the existing data and can sum up to a fair amount of space for large files.
- Tests for writing multiple stripes must be done, which was not possible in the course of this thesis due to the fact that the client is not yet able to properly compress multiple stripes.
- The current handling of uncompressed data, that is, adding a header nonetheless, can require additional pages, which exceeds the limit of 256 pages per RPC. It might be evaluated, if rather changes to the client or to the backend are more reliable. The former means allowing more pages when compression is enabled, the latter comes down to not adding the header after all and assuming that a missing header always refers to this case (and not, for instance, to data corruption)

Besides these tasks, the work done in this thesis results in the client to be able to write as well as read compressed data chunks with an ZFS backend. This ultimately is an important step on the way to fully integrate client-side data compression in Lustre. As of now, the full I/O stack for pre-compressed data is basically functioning and can be considered as a a whole when implementing further functionality. For instance, when reading data, decompressing is still missing, which can now be implemented. For this, the chunk descriptors are received and can be used to chose the appropriate decompressor. Also, the proper storage of the metadata allows to do first tests regarding the actual adaptiveness of the compression.

List of Acronyms

ARC Adaptive Replacement Cache. 7, 8, 14–16, 26, 28

CLI command-line interface. 19, 34

CPU Central Processing Unit. 19, 33

DMU Data Management Unit. 7, 11, 14–16, 20

HDD Hard Drive Disk. 1, 9

HPC High Performance Computing. 1, 2, 33

I/O Input and Output. 1, 5, 8, 10–12, 14–16, 25, 51, 52, 55

LDI Linux Device Interface. 7

lnb local niobuf. 25–28, 31, 39, 44, 56

LOV Lustre Object Volume. 13, 15

LZ Lempel-Ziv. 33

LZO Lempel-Ziv-Oberhumer. 19

MDC Metadata Client. 12

MDS Meta Data Server. 9, 10, 13, 26

MDT Meta Data Target. 12, 14, 21, 47, 62

MGC Management Client. 12

MGS Management Server. 9, 10, 47, 62

MGT Management Target. 12, 14

MMU Memory Management Unit. 26

niobuf Network I/O Buffer. 13, 15, 16, 25, 30, 31, 44

OFD Object Filter Device. 14, 15, 24

OS Operating System. 7, 62

OSC Object Storage Client. 12, 13, 15, 16, 24, 55

OSD Object Storage Device. 9, 37

OSS Object Storage Server. 9, 10, 15, 35

OST Object Storage Target. 9, 10, 12–14, 26, 47, 62

POSIX Portable Operating System Interface. 6, 10, 12, 14

rnb remote niobuf. 25–27, 38, 39

RPC Remote Procedural Call. 13–16, 20, 21, 24, 25, 32, 35, 36, 41, 46, 52, 56

SPA Storage Pool Allocator. 7, 8

SSD Solid State Drive. 1, 8

TCP Transmission Control Protocol. 13

VDEV Virtual Device. 8

VFS Virtual File System. 6, 11, 12, 14

ZAP ZFS Attribute Processor. 7, 11, 21

ZIL ZFS Intent Log. 7

ZIO ZFS I/O. 8, 17, 20, 27, 28, 31

ZPL ZFS POSIX Layer. 6, 11

ZVOL ZFS Volume. 6, 11

List of Figures

1.1 Development of computational speed, internet bandwidth, storage capacity and storage speed (based on [KKL16] and [Nie98]) . . . 2

2.1 Relationship between ZFS' modules (based on graphics from [McK+14] (original) and [Beh17]) . . . 6
2.2 Lustre architecture [Fuc16] . . . 9
2.3 Lustre entering the ZFS I/O stack (based on graphics from [McK+14] (original) and [Beh17]) . . . 11
2.4 I/O stack including Lustre client, Lustre server and ZFS backend . . . 12
2.5 Basic idea of the currently implemented compression feature within Lustre's OSC layer . . . 16

3.1 Basic idea for making the metadata a part of the data . . . 22
3.2 Pseudocode for the main loop in `osd_bufs_get_write` . . . 27
3.3 Pseudocode for the main loop in `osd_bufs_get_write` . . . 28
3.4 Writing the `struct chunk_desc` before committing using `dmu_write` . . . 29
3.5 Assigning the header following the approach to modify arcbufs before receiving data . . . 29
3.6 Pseudocode for the main loop in `osd_bufs_get_read` . . . 30
3.7 Layout of the compressed data with headers in ZFS . . . 31
3.8 Pseudocode for the modified main loop for reading compressed data . . . 32

5.1 `compress_chunks` (client): New function to assemble chunks based on the chunk-size not dividable by `PAGE_SIZE` . . . 36
5.2 Invoked functions during read and write with new parameters . . . 37
5.3 Force PTLRPC to write the data beyond the header using `lnb_page_offset` and `lnb_len` . . . 40

6.1 Network throughput for 1 MiB, 7 MiB and 10 MiB of climate data for the unmodified Lustre and after performing compression in the modified Lustre . . . 48
6.2 Read time for 1 MiB, 7 MiB and 10 MiB of climate data for the unmodified and modified Lustre respectively . . . 49

List of Listings

3.1 The unmodified, but commented struct chunk_desc as introduced in [Fuc16] . . 20
3.2 Layout of an RPC for bulk read or writes (brw) 21
3.3 tgt_brw_write (commented): Most important steps 25
3.4 struct niobuf_local and struct niobuf_remote 26
3.5 Prototype of osd_bufs_get_read 30
5.1 Check for OBD_COMPRESSED_IO flag within tgt_io_data_unpack 35
5.2 Introduction of chunk size constants for PTLRPC 36
5.3 Prototype of osd_bufs_get_compressed_write 38
5.4 osd_bufs_get_compressed_write: Request arcbufs and prepare parsing buffers . 39
5.5 osd_bufs_get_compressed_write: Process received arcbuf into lnbs 39
5.6 osd_write_commit: A third branch for committing compressed arcbufs was added 41
5.7 tgt_handle_request0: The field &RMF_CHUNK_DESC is set to the maximum size for the reply 41
5.8 tgt_brw_write: The field &RMF_CHUNK_DESC is set to the exact size 42
5.9 ofd_preprw_read: Allocating memory for extracting the chunk headers 42
5.10 Prototype of osd_bufs_get_compressed_read 42
5.11 osd_bufs_get_compressed_read: Fallback to osd_bufs_get_read when the data is not actually compressed 43
5.12 osd_bufs_get_compressed_read: Reading data chunk-wise and extracting the header when a chunk begins 43
5.13 osd_bufs_get_compressed_read: Adjusting the fields of the local niobuf after reading the header 43
5.14 osd_bufs_get_compressed_read: Adjusting offsets after the chunks end 44
6.1 compress_chunks: Modified to return uncompressed data 45
6.2 Test of the correctness for uncompressed data (first chunk) 46
6.3 Test of the correctness for the first chunk 46
6.4 osc_brw_fini_request: Reading the chunk descriptors during read 47
6.5 Script used to extract network statistics from /proc/net/dev 48
6.6 Command used for measuring the read performance 48

Bibliography

[Ahr16] Matthew Ahrens. *Read Write Lecture.* The lecture was presented as Lecture 6 of Marshall Kirk McKusick's class, FreeBSD Kernel Internals: An Intensive Code Walkthrough, taught in the spring of 2016 at the historic Hillside Club in Berkeley, California. 2016. URL: `http://open-zfs.org/wiki/Documentation/Read_Write_Lecture` (visited on 06/30/2017).

[Beh17] Niklas Behrmann. "Support for external data transformation in ZFS". Master's Thesis. Universität Hamburg, Apr. 2017.

[BM01] David A. Bader and Lisa Moret Bernard M. and Vawter. *Industrial applications of high performance computing for phylogeny reconstruction.* 2001. DOI: `10.1117/12.434868`. URL: `http://dx.doi.org/10.1117/12.434868`.

[Bon06] Jeff Bonwick. *You say zeta, I say zetta.* May 2006. URL: `https://blogs.oracle.com/bonwick/you-say-zeta,-i-say-zetta` (visited on 06/30/2017).

[CRK05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition.* 3rd ed. O'Reilly Media, Inc., 2005. ISBN: 978-0-596-00590-0.

[CT05] Benny Chor and Tamir Tuller. "Maximum likelihood of evolutionary trees: hardness and approximation". In: *Bioinformatics* 21.suppl_1 (2005), p. i97. DOI: `10.1093/bioinformatics/bti1027`. eprint: `/oup/backfile/content_public/journal/bioinformatics/21/suppl_1/10.1093/bioinformatics/bti1027/2/bti1027.pdf`. URL: `http://dx.doi.org/10.1093/bioinformatics/bti1027` (visited on 07/04/2017).

[DDD09] Andreas Dilger, Rahul Deshmukh, and John Dawson. *End to End Data Integrity Design.* Tech. rep. Sun microsystems, June 2009.

[Dil10] Andreas Dilger. *ZFS Features & Concepts TOI.* 2010. URL: `http://wiki.lustre.org/images/4/49/Beijing-2010.2-ZFS_overview_3.1_Dilger.pdf` (visited on 06/30/2017).

[Edi] The TOP500 Editors. *PERFORMANCE DEVELOPMENT.* URL: `https://www.top500.org/statistics/perfdevel/` (visited on 07/03/2017).

[Fuc+16] Anna Fuchs et al. "Enhanced Adaptive Compression in Lustre". ISC High Performance. 2016. URL: `http://isc-hpc.com/isc17_ap/presentationdetails.htm?t=presentation&o=1144&a=select&ra=personendetails`.

[Fuc16] Anna Fuchs. "Client-Side Data Transformation in Lustre". Master's Thesis. Universität Hamburg, May 2016.

[Gor17] Brent Gorda. *Podcast: Brent Gorda on the State of Lustre at ISC 2017*. June 2017. URL: https://insidehpc.com/2017/06/podcast-brent-gorda-state-lustre-isc-2017/ (visited on 06/30/2017).

[Gre08] Brendan Gregg. *ZFS L2ARC*. July 2008. URL: https://bigip-blogs-cms-adc.oracle.com/brendan/entry/test (visited on 07/04/2017).

[KKL16] Michael Kuhn, Julian Kunkel, and Thomas Ludwig. "Data Compression for Climate Data". In: *Supercomputing Frontiers and Innovations*. Volume 3, Number 1 (June 2016). Ed. by Jack Dongarra and Vladimir Voevodin, pp. 75–94. DOI: http://dx.doi.org/10.14529/jsfi1601. URL: http://superfri.org/superfri/article/view/101.

[KS06] C. Krintz and S. Sucu. "Adaptive on-the-fly compression". In: *IEEE Transactions on Parallel and Distributed Systems* 17.1 (Jan. 2006), pp. 15–24. ISSN: 1045-9219. DOI: 10.1109/TPDS.2006.3.

[Lar17] Michael Larabel. *Facebook Looking To Add Zstd Support To The Linux Kernel, Btrfs*. Aug. 2017. URL: https://www.phoronix.com/scan.php?page=news_item%5C&px=Facebook-Linux-Zstd.

[LI06] Peter Lindstrom and Martin Isenburg. "Fast and Efficient Compression of Floating-Point Data". In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (Sept. 2006), pp. 1245–1250. ISSN: 1077-2626. DOI: 10.1109/TVCG.2006.143. URL: http://dx.doi.org/10.1109/TVCG.2006.143.

[Lin] Pao-Yen Lin. "Basic Image Compression Algorithm and Introduction to JPEG Standard". In: (). URL: http://disp.ee.ntu.edu.tw/meeting/%E4%BF%9D%E8%A8%80/Basic%20Image%20Compression%20Algorithm%20and%20Introduction%20of%20JPEG%20Standard/Basic%20Image%20Compression%20Algorithm%20and%20Introduction%20of%20JPEG%20Standard.pdf (visited on 07/07/2017).

[Lus] Lustre. *Lustre Software Release 2.x Operations Manual*. URL: http://doc.lustre.org/lustre_manual.xhtml (visited on 06/30/2017).

[McK+14] Marshall Kirk McKusick et al. *The design and implementation of the FreeBSD operating system*. 2nd ed. Addison Wesley, Sept. 2014.

[Nes16] Gvozden Neskovic. "Lustre ZFS Snapshot Overview". 2016. URL: http://cdn.opensfs.org/wp-content/uploads/2016/04/LUG2016D1_Vectorized-ZFS-RAIDZ_Gvozden-Neskovic.pdf.

[Nie98] Jakob Nielsen. *Nielsen's Law of Internet Bandwidth*. Apr. 1998. URL: https://www.nngroup.com/articles/law-of-bandwidth/ (visited on 07/02/2017).

[Ope11] OpenSolaris. *ZFS FAQ*. Archived version of the original not being online anymore. Mar. 2011. URL: https://web.archive.org/web/20110515061128/http://hub.opensolaris.org/bin/view/Community+Group+zfs/faq/ (visited on 06/30/2017).

[Ope14] OpenZFS. *History*. May 2014. URL: http://open-zfs.org/wiki/History (visited on 06/30/2017).

[Sch17] Sven Schmidt. *Implementierung von LZ4Fast im Linux-Kernel*. In German. Mar. 2017. URL: https://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2016_2017/pre1617-schmidt-lz4-im-kernel-ausarbeitung.pdf.

[SM14] Vahan Simonyan and Rajan Mazumbder. "High-Performance Integrated Virtual Environment (HIVE) Tools and Applications for Big Data Analysis." In: *Genes* (2014), pp. 957–981. DOI: 10.3390/genes5040957. URL: +%20http://dx.doi.org/10.1093/bioinformatics/bti1027 (visited on 07/04/2017).

[Str17] Ken Strandberg. *Bolstering Lustre on ZFS: Highlights of continuing work*. Jan. 2017. URL: https://www.nextplatform.com/2017/01/11/bolstering-lustre-zfs-highlights-continuing-work/ (visited on 07/13/2017).

[VGK11] S. Vadlakonda, N. Gugle, and R. Kasturi. *Network characteristic-based compression of network traffic*. US Patent 7,953,881. May 2011. URL: https://www.google.com/patents/US7953881.

[Wan+09] Feiyi Wang et al. *Understanding Lustre filesystem internals*. Apr. 2009. URL: http://wiki.old.lustre.org/images/d/da/Understanding_Lustre_Filesystem_Internals.pdf (visited on 06/30/2017).

[Wel+11] Benjamin Welton et al. "Improving I/O Forwarding Throughput with Data Compression". In: *Proceedings of the 2011 IEEE International Conference on Cluster Computing*. CLUSTER '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 438–445. ISBN: 978-0-7695-4516-5. DOI: 10.1109/CLUSTER.2011.80. URL: http://dx.doi.org/10.1109/CLUSTER.2011.80.

[Wik17] Wikipedia. *High Productivity Computing Systems — Wikipedia, The Free Encyclopedia*. 2017. URL: https://en.wikipedia.org/w/index.php?title=High_Productivity_Computing_Systems&oldid=778259637 (visited on 07/13/2017).

[YJ17] Fan Yong and Zhang Jinghai. "Lustre ZFS Snapshot Overview". 2017. URL: http://cdn.opensfs.org/wp-content/uploads/2016/04/LUG2016D1_Lustre-ZFS-Snapshots_Fan-Yong_Revised_v2.pdf.

[zfs15] zfsbuild. *Explanation of ARC and L2ARC*. Apr. 2015. URL: https://www.zfsbuild.com/2010/04/15/explanation-of-arc-and-l2arc/ (visited on 07/02/2017).

[Zhu16] Alexey Zhuravlev. "ZFS: Metadata performance". 2016. URL: https://www.eofs.eu/_media/events/lad16/02_zfs_md_performance_improvements_zhuravlev.pdf.

Appendices

A | Infrastructure

As a basis for implementing the new functionality described in this thesis as well as the tests and benchmarks, several virtual machines managed by VirtualBox were used. For both, the modified and unmodified Lustre, a setup with at least three machines, respectively, was created: An OST, a shared MDT/MGS and at least one client. After setting up the first machine with either of the Lustre versions, the others were created as clones. The basic attributes are outlined in the following:

- OS: CentOS 7 (64 Bit, Linux Kernel 3.10)
- 4 GiB main memory
- Dynamically allocated secondary memory
- Connection via host-only adapter with static IP addresses
- Lustre: Version 2.9.0 with the modifications done in [Fuc16]
- ZFS: Version 0.7.0 RC3 with the modifications described in [Beh17]
- SPL (on which ZFS depends): Version 0.7.0 RC3 (upstream version)

The setup of Lustre mostly followed the description in [Fuc16, p. 91 - 96]. However, the described version did not yet include the build parameter for compression, hence `--enable-compression` was additionally passed to `./configure`.
For ZFS, the version of [Beh17] was used instead of the version available upstream. For the installation of ZFS from scratch, the steps depicted in Listing A.1 must be executed for both, SPL and ZFS respectively, starting with SPL.

```
# cd <spl|zfs>
# sh autogen.sh
# ./configure --with-spec=redhat
# make pkg-utils pkg-kmod # generate rpms
# yum localinstall *.$(arch).rpm
```

Listing A.1: Installation of ZFS from scratch

B Sourcecodes

B.1 cdesc_for_off

```
static inline struct chunk_desc *cdesc_for_off(loff_t const off,
  struct chunk_desc *cdesc, size_t nchunks)
{
  for (nchunks--; nchunks >= 0; cdesc++, nchunks--)
    if (cdesc->poff == off || cdesc->loff == off || cdesc->poff == (off
   - sizeof(struct chunk_desc)))
      return cdesc;

  return NULL;
}
```

Listing B.1: Prototype of cdesc_for_off

B.2 chunks_max_psize

```
static size_t chunks_max_psize(struct chunk_desc *cdesc, int nchunks)
{
  size_t max = 0;

  for (nchunks--; nchunks >= 0; nchunks--)
    if (cdesc[nchunks].psize > max)
      max = cdesc[nchunks].psize;

  return max;
}
```

Listing B.2: Prototype of chunks_max_psize

B.3 Modified struct chunk_desc

```
struct chunk_desc {
```

```
2   __u32 ppages;
3   __u32 psize;
4   __u32 lpages;
5   __u32 lsize;
6   __u32 cksum;
7   __u32 algo;
8   __u32 flags;
9   __u32 pad_1; /* We won't need padding as soon as we remove algo in
      favor of flags */
10  __u64 poff;
11  __u64 loff;
12 };
```

Listing B.3: Modified `struct chunk_desc`

B.4 osd_bufs_get_compressed_write

```
1  static int osd_bufs_get_compressed_write(const struct lu_env *env,
       struct osd_object *obj,
2          loff_t off, ssize_t len, struct niobuf_local *lnb,
3          int nr_chunks, struct chunk_desc *cdesc)
4  {
5    struct osd_device *osd = osd_obj2dev(obj);
6    struct chunk_desc *curr_cdesc = NULL;
7    int                 plen, off_in_buf;
8    int                 rc, i = 0, npages = 0;
9    arc_buf_t          *abuf;
10   uint64_t       chunk_remaining;
11   size_t const       hdr_sz = sizeof(struct chunk_desc),
12          max_psize = chunks_max_psize(cdesc, nr_chunks);
13
14   ENTRY;
15
16   /* this is _COMPRESSED_write */
17   LASSERT(nr_chunks > 0);
18   LASSERT(cdesc != NULL);
19   LASSERT(max_psize > 0);
20
21   /* It is thinkable to have multiple chunks per rnb,
22    * therefore the double loop.
23    * Ideally, if rnb = chunk, the outer loop iterates exactly once */
24   while (len > 0) {
25     /* get current chunk descriptor
26      * if we're at position 0, there must exist one anyway
27      * (since this is bufs_get_COMPRESSED_write),
28      * else, when one chunk ends but len is still > 0,
29      * another chunk is supposed to start or something went wrong */
30     curr_cdesc = cdesc_for_off(off, cdesc, nr_chunks);
31
32     LASSERT(curr_cdesc != NULL);
33
34     /* requst arcbuf of the appropriate size */
```

```
35      abuf = dmu_request_compressed_arcbuf(obj->oo_db,
36        max_psize + hdr_sz,
37        max_t(int, PTLRPC_MIN_CHUNK_SIZE + hdr_sz,
38          curr_cdesc->lsize + hdr_sz));
39
40      if (unlikely(abuf == NULL))
41        GOTO(out_err, rc = -ENOMEM);
42
43      atomic_inc(&osd->od_zerocopy_loan);
44
45      off_in_buf = 0;
46      chunk_remaining = curr_cdesc->psize + hdr_sz;
47
48      /* go over pages arcbuf contains, put them as
49       * local niobufs for ptlrpc's bulks */
50      while (chunk_remaining > 0) {
51        plen = min_t(int, chunk_remaining, PAGE_SIZE);
52
53        if (off_in_buf == 0) {
54          /* beginning of chunk, reserve space for the chunk header
55           * and assign it */
56
57          /* remember arcbuf for commit */
58          lnb[i].lnb_data = abuf;
59
60          /* adapt offset for header */
61          plen -= hdr_sz;
62          lnb[i].lnb_page_offset = hdr_sz;
63
64          memcpy(abuf->b_data, curr_cdesc, hdr_sz);
65        } else {
66          lnb[i].lnb_data = NULL;
67          lnb[i].lnb_page_offset = 0;
68        }
69
70        /* this one is not supposed to fail */
71        lnb[i].lnb_page = kmem_to_page(abuf->b_data + off_in_buf);
72        LASSERT(lnb[i].lnb_page);
73
74        lprocfs_counter_add(osd->od_stats,
75            LPROC_OSD_ZEROCOPY_IO, 1);
76
77        lnb[i].lnb_file_offset = off + hdr_sz;
78        lnb[i].lnb_len = plen;
79        lnb[i].lnb_rc = 0;
80
81        len -= min_t(int, len, plen);
82        off += min_t(int, chunk_remaining, PAGE_SIZE);
83        chunk_remaining -= min_t(int, chunk_remaining, PAGE_SIZE);
84        off_in_buf += PAGE_SIZE;
85        i++;
86        npages++;
87      }
88    }
89
90    LASSERT(npages < PTLRPC_MAX_BRW_PAGES);
```

```
91
92	RETURN(npages);
93
94 out_err:
95	osd_bufs_put(env, &obj->oo_dt, lnb, npages);
96	RETURN(rc);
97 }
```

Listing B.4: The newly introduced function `osd_bufs_get_compressed_write`

B.5 Changing the blocksize

```
1 static int osd_set_blocksize(struct osd_object *obj, struct osd_thandle
     *oh,
2        uint32_t blksz)
3 {
4   struct osd_device *osd = osd_obj2dev(obj);
5   dmu_buf_impl_t     *db = (dmu_buf_impl_t *)obj->oo_db;
6   dnode_t      *dn;
7   int       rc = 0;
8
9   ENTRY;
10
11   DB_DNODE_ENTER(db);
12   dn = DB_DNODE(db);
13
14   if (dn->dn_maxblkid > 0) /* can't change block size */
15     GOTO(out, rc);
16
17   if (dn->dn_datablksz >= osd->od_max_blksz || dn->dn_datablksz == blksz
     )
18     GOTO(out, rc);
19
20   down_write(&obj->oo_guard);
21
22   if (dn->dn_datablksz >= osd->od_max_blksz  || dn->dn_datablksz ==
     blksz)
23     /* check again after grabbing lock */
24     GOTO(out_unlock, rc);
25
26   if (!is_power_of_2(blksz))
27     blksz = size_roundup_power2(blksz);
28
29   if (blksz > dn->dn_datablksz) {
30     rc = -dmu_object_set_blocksize(osd->od_os, dn->dn_object,
31                 blksz, 0, oh->ot_tx);
32     LASSERT(ergo(rc == 0, dn->dn_datablksz >= blksz));
33     if (rc < 0)
34       CDEBUG(D_INODE, "object "DFID": change block size"
35              "%u -> %u error rc = %d\n",
36              PFID(lu_object_fid(&obj->oo_dt.do_lu)),
37              dn->dn_datablksz, blksz, rc);
```

```
38    }
39    EXIT;
40  out_unlock:
41    up_write(&obj->oo_guard);
42  out:
43    DB_DNODE_EXIT(db);
44    return rc;
45  }
46
47  static int osd_grow_blocksize(struct osd_object *obj, struct osd_thandle
        *oh,
48              uint64_t start, uint64_t end)
49  {
50    struct osd_device *osd = osd_obj2dev(obj);
51    dmu_buf_impl_t    *db = (dmu_buf_impl_t *)obj->oo_db;
52    dnode_t      *dn;
53    uint32_t      blksz = 0;
54
55    ENTRY;
56
57    DB_DNODE_ENTER(db);
58    dn = DB_DNODE(db);
59    blksz = dn->dn_datablksz;
60    DB_DNODE_EXIT(db);
61
62    /* now ZFS can support up to 16MB block size, and if the write
63     * is sequential, it just increases the block size gradually */
64    if (start <= blksz) { /* sequential */
65      blksz = (uint32_t)min_t(uint64_t, osd->od_max_blksz, end);
66    } else { /* sparse, pick a block size by write region */
67      blksz = (uint32_t)min_t(uint64_t, osd->od_max_blksz,
68            end - start);
69    }
70
71    return osd_set_blocksize(obj, oh, blksz);
72  }
```

Listing B.5: osd_grow_blocksize was split into two functions, osd_grow_blocksize and osd_set_blocksize for the compressed case and the unmodified case

B.6 osd_bufs_get_compressed_read

```
1  static int osd_bufs_get_compressed_read(const struct lu_env *env, struct
       osd_object *obj,
2          loff_t off, ssize_t len, struct niobuf_local *lnb,
3          int *nr_chunks, struct chunk_desc **cdesc)
4  {
5    struct chunk_desc *curr_chunk = NULL;
6    struct osd_device *osd = osd_obj2dev(obj);
7    dmu_buf_t   **dbp;
8    unsigned long     start = cfs_time_current();
9    int                 rc, i, numbufs, npages = 0, nchunks = 0, compressed
      ;
```

```
10  size_t const        hdr_sz = sizeof(struct chunk_desc);
11  ENTRY;
12
13  compressed = obj->oo_db->db_compressed;
14
15  /* fallback to uncompressed read if dbuf was not stored pre-compressed
      */
16  if (0 /* !compressed TODO */)
17  /* TODO Sv: HOW do we know if it is compressed? db_compressed is
     always 0! */
18    RETURN(osd_bufs_get_read(env, obj, off, len, lnb));
19
20  *nr_chunks = 0;
21
22  record_start_io(osd, READ, 0);
23
24  while (len > 0) {
25    rc = -dmu_buf_hold_array_by_bonus_compressed(
26      obj->oo_db, off, len, TRUE,
27      osd_zerocopy_tag, &numbufs,
28      &dbp);
29
30    if (unlikely(rc))
31      GOTO(err, rc);
32
33    for (i = 0; i < numbufs; i++) {
34      void *dbf = dbp[i];
35      int bufoff, tocpy, thispage, chunk_rmng = 0, chunk_rmng_l = 0,
    chunk_bn = 0;
36
37      if (len == 0)
38        break;
39
40      atomic_inc(&osd->od_zerocopy_pin);
41
42      bufoff = 0;
43
44      tocpy = min_t(int, dbp[i]->db_size, len);
45
46      /* kind of trick to differentiate dbuf vs. arcbuf */
47      LASSERT(((unsigned long)dbp[i] & 1) == 0);
48      dbf = (void *) ((unsigned long)dbp[i] | 1);
49
50      while (tocpy > 0) {
51        /* when we're reading compressed (chunked)
52         * data, each chunk is preceeded by a header
53         * describing the chunk. Since we can only make
54         * guesses based on the assumptions we made
55         * during write, we try to read a header at each
56         * iteration to be sure and be independent of
57         * that assumptions.
58         */
59        if (curr_chunk == NULL) {
60          /* try to read header */
61          curr_chunk = (struct chunk_desc *)(dbp[i]->db_data);
62
```

```
63            /* as stated above:
64             * we can't be sure this is an actual
65             * chunk header -> do some checks */
66            if (((curr_chunk->flags & CHUNK_DESC_MASK) == CHUNK_HAS_HDR)
67              && (int)curr_chunk->psize > 0
68              && (int)curr_chunk->lsize > 0) {
69              chunk_bn = 1;
70
71              /* valid header! From this point,
72               * we know how large the chunk
73               * is  and will read until its end */
74              chunk_rmng = curr_chunk->psize;
75              chunk_rmng_l = curr_chunk->lsize;
76
77              /* headers are sent back to client
78               * for decompression; extract it */
79              memcpy(&((*cdesc)[nchunks++]), curr_chunk, hdr_sz);
80            } else {
81              /* not a valid chunk header,
82               * try again later */
83              curr_chunk = NULL;
84              chunk_rmng = chunk_rmng_l = 0;
85            }
86          }
87
88          thispage = min_t(int, tocpy, PAGE_SIZE);
89
90          if (chunk_rmng > 0) {
91            /* we're within the actual data of the
92             * chunk and want to take each page */
93
94            thispage = min(chunk_rmng, thispage);
95
96            if (chunk_bn) {
97              /* this is the "did not read any
98               * chunk data" case
99               * at this point, we're removing
100              * the header since it is no
101              * longer needed */
102             lnb->lnb_page_offset = hdr_sz;
103             thispage -= hdr_sz;
104           } else
105             lnb->lnb_page_offset = 0;
106
107           chunk_bn = 0;
108
109           lnb->lnb_len = thispage;
110           lnb->lnb_rc = 0;
111           lnb->lnb_page = kmem_to_page(dbp[i]->db_data +
112             bufoff);
113           lnb->lnb_file_offset = max_t(int, curr_chunk->loff + bufoff -
    hdr_sz, 0);
114
115           /* mark just a single slot: we need this
116            * reference to dbuf to be released once */
117           lnb->lnb_data = dbf;
```

```c
          dbf = NULL;
        } else {
          /* this is data not related to a chunk,
           * hence just continue */

          lnb->lnb_rc = 0;
          lnb->lnb_file_offset = 0;
          lnb->lnb_page_offset = 0;
          lnb->lnb_len = 0;
          lnb->lnb_page = NULL;
        }

        npages++;
        lnb++;

        chunk_rmng = max_t(int, chunk_rmng - thispage, 0);
        chunk_rmng_l = max_t(int, chunk_rmng_l - thispage, 0);
        tocpy -= PAGE_SIZE;
        len -= PAGE_SIZE;
        bufoff += PAGE_SIZE;

        if (curr_chunk != NULL && chunk_rmng_l == 0) {
          /* we reached the physical and logical
           * end of the chunk, hence we're done */
          off = max_t(int, curr_chunk->loff
            + curr_chunk->lsize
            + ((curr_chunk->loff / curr_chunk->lsize) * hdr_sz),
            PTLRPC_MIN_CHUNK_SIZE + hdr_sz);
          curr_chunk = NULL;
          len -= tocpy;
          tocpy = 0;
        }
      }

      /* steal dbuf so dmu_buf_rele_array() can't release
       * it */
      dbp[i] = NULL;
    }

    dmu_buf_rele_array(dbp, numbufs, osd_zerocopy_tag);
  }

  record_end_io(osd, READ, cfs_time_current() - start,
          npages * PAGE_SIZE, npages);

  *nr_chunks = nchunks;

  RETURN(npages);

err:
  LASSERT(rc < 0);
  osd_bufs_put(env, &obj->oo_dt, lnb - npages, npages);
  RETURN(rc);
}
```

Listing B.6: The newly introduced function `osd_bufs_get_compressed_read`

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum   Unterschrift

Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Ort, Datum   Unterschrift