Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Masterarbeit

Support for external data transformation in ZFS

vorgelegt von

Niklas Behrmann

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	6324995
Erstgutachter:	Prof. Dr. Thomas Ludwig
Zweitgutachter:	Dr. Michael Kuhn
Betreuer:	Dr. Michael Kuhn, Anna Fuchs

Hamburg, 2017-04-06

Abstract

While computational power of high-performance computing systems doubled every two years over the last 50 years as predicted by Moore's law, the same was not true for storage speed and capacity. Compression has become a useful technique to bridge the increasing performance and scalability gap between computation and Input/Output (I/O). For that reason some local filesystems like ZFS support transparent compression of data. For parallel distributed filesystems like Lustre this approach does not exist. Lustre is frequently used in supercomputers. The Intel Parallel Computing Centers (IPCC) for Lustre filesystem project is aiming for compression support in Lustre at multiple levels. The IPCC are universities, institutions, and labs. Their primary focus is to modernize applications to increase parallelism and scalability. A prior thesis started the implementation of online compression with the compression algorithm LZ4 in Lustre. The focus of this implementation was to increase throughput performance. The data is compressed on clientside and send compressed to the server. However the compression leads potentially to a bad read performance.

This problem might be solved through modifying the ZFS filesystem which is utilized by Lustre servers as a backend filesystem. ZFS already has a compression functionality integrated which provides good read performance for compressed data. The idea is to make use of this and store the Lustre's data in ZFS as if it was compressed by ZFS. Therefore a new interface that takes the necessary information has to be created. Implementing this is the purpose of this thesis. The goal is to enable the Lustre compression to save space on disk and most importantly fix the bad read performance. Throughout this thesis the necessary modifications to ZFS are described. The main task is to provide information to ZFS about the compressed size and the uncompressed size of the data. Afterwards a possible implementation of the specified feature is presented.

First tests indicate that data which is compressed by Lustre can be read efficiently by ZFS if provided with the necessary metadata.

Contents

1 Introduction

This chapter first introduces high performance computing and describes the importance and role of compression. Following that an introduction to file systems is given. Then the goals and structure of this thesis are presented.

Motivation

Many computations in science, engineering or business cannot be done with a simple desktop computers. The field of high performance computing (HPC) is dedicated to provide the computational power and the algorithms to solve these problems in a short period of time. Computers used in HPC are called supercomputers. A modern supercomputer is composed of thousands individual computers called nodes. This results in supercomputers having millions of cores and costing several hundred million Euros. While also costing millions each year because of the immense electricity consumption as well as maintenance. The performance of these computers are measured in floating-point operations per second (FLOPS). The TOP 500 list keeps track of the 500 most powerful supercomputers. Computers on this list are usually petascale meaning they reach performance past one petaflops. For example Mistral at the Deutsches Klimarechenzentrum (DKRZ) reaches 3.0 PFLOPS [dkr]. Mistral provides 54 PB of usable disk space and has an aggregated main memory of about 266 TB. Supercomputers are commonly used for scientific computations that need high computing performance. These computations often need excessive input and output. So I/O as well as network throughput can be a bottleneck.

Especially given the fact that storage speed grew much slower than computation speed. While latter grew by a factor of 300 every 10 years and storage capacity by a factor of 100 in the same amount of time, storage speed only grew by a factor of 20 in 10 years [MK16]. This trend can be supported by comparing Mistral's performance with Blizzard's [bli10], the former supercomputer of the DKRZ from 2009 to 2015. The performance increased by a factor of 20, while the capacity is only 8 times higher in the new supercomputer. Even though climate research is especially data extensive, and thus the DKRZ is laying the focus on data capacity and throughput. This is shown in the storage throughput which could be increased by a greater factor than storage capacity.

This development leads to a need to perform I/O operations as efficiently as possible. Data reduction like compression is a rising approach to increase the I/O speed. For this reason some modern filesystems like btrfs [btr] and ZFS support compression of files. Data written to these filesystems will be transparently compressed with a choosable

	Blizzard (2009)	Mistral (2015)	Growth rate
Performance	150 TFLOPS	3 PFLOPS	2000%
Storage capacity	5.6 PB	45 PB	804%
Storage throughput	30 GB/s	400 GB/s	1333%

Table 1.1: Comparsion of Mistral and Blizzard [JMK14].

algorithm. Not only saves this disk storage it will also increase performance [Leo09]. This is the case because modern compression algorithms like LZ4 combined with the fast computation speed compress and decompress the data faster than it would take to read and write the uncompressed data.

In high performance computing having parallel distributed access on data is a necessity for efficent I/O. A commonly used filesystem for this purpose is Lustre. Over 70% of the supercomputers in the TOP 500 are using Lustre. Whereas 9 out of them are in the top 10. Several of these supercomputers have a Lustre filesystem with more than 50 PB of storage, providing more than 1TB/s of throughput to more than 20000 clients. Lustre was developed with focus on providing high performance on large distributed clusters. It uses a local filesystem as backend. For that either ldiskfs or ZFS are available.

The goal of the IPCC project for enhanced adaptive compression in Lustre is to implement compression support in Lustre. Compression on the clientside should allow to use the available network and storage capacity more efficiently. Additionally applications should provide hints to Lustre that are useful for compression. With adaptive compression it should be possible to choose appropriate settings depending on performance metrics and projected benefits. The project is carried out by the the Scientific Computing group located at the Universität Hamburg [ipc].

Lustre compression prototype

The thesis [Fuc16] proposed a basic compression functionality in Lustre. This basic functionality comprises asynchronous buffered write and read operations of user data. Thereby the most common use cases are covered. The focus of the design lays on increasing the network throughput. Therefore the data is compressed transparently on clientside and send compressed to the server.

The current state of the implementation only is able to cope with compressible data and includes no metadata send to the server. So while reading decompression is applied blindly. However, the goal is to send metadata containing the compression algorithm used and the compressed size of the data to the server. But even if the server has information about the compressed data, the underlying local filesystem has none. This ultimately will lead to bad read performance.

Thesis goals

To resolve this problem the idea is to make use of the compression functionality of ZFS. ZFS should be provided with enough information by Lustre to store the data like they were compressed by ZFS itself. This way the compression done by Lustre is able to save storage on disk while simultaneously increase the read performance since less data needs to be read in and effective readaheads can be made. The implementation should keep the wanted adaptive compression by Lustre in mind, and thus provide flexibility on the choice of the algorithm. Over the course of this thesis it is analyzed and later implemented how the data compressed by the Lustre client can be correctly integrated into ZFS.

Structure

First an overview of relevant topics for this thesis is given. It starts with a brief introduction on compression and its applications. Afterwards an introduction to the Lustre filesystem is presented. Then with ZFS the most important subject gets introduced. All significant features are described as well as the internal structure. The design how to integrate compressed data into ZFS is presented in Chapter 3. Chapter 4 displays related works done in transparent compression and the usage of ZFS as a backend of Lustre. Chapter 5 presents the implementation. For that detailed information about ZFS's I/O path are given. Followed by a description of the implementation for writing as well as reading Lustre's pre-compressed data. Chapter 6 evaluates the performance for reading the compressed data. At last Chapter 7 concludes the thesis and suggests future works.

2 Overview

This chapter provides an overview of all relevant topics for this thesis. First basic introductions to compression algorithms are conducted. The focus lies on algorithms that are available in ZFS. The following section presents essential information and features of Lustre. The next section deals with ZFS. Information about the features of ZFS and basic information is followed by a description of the internal functioning of ZFS.

2.1 Compression

The overall goal of compression is to encode information in a way that data can be represented with fewer bytes. Generally there are two types of compression lossless or lossy.

2.1.1 Lossy compression

Lossy compression reduces the data with discarding partial information and through approximations. This compression method is mostly used for multimedia data like video, images or audio files. Prominent examples are the mp3 format for audio files or JPEG for images. In the best the data reduction made by the lossy compression is not recognizable by the user. If speed and data reduction is the focus also a quality loss of the data might be acceptable for the user. Lossy compression is irreversible. The compressed data cannot be restored into its inital state. This makes lossy compression not applicable for filesystems. In filesystems the compression should be transparent to the user. This means that the data should be returned to the user in the same state as it was stored. However, this is not possible once the data is compressed using lossy algorithms.

2.1.2 Lossless compression

In difference to that lossless compression aims to compress data in a way that all information can still be retrieved through decompressing the data. This type of compression is used in a lot of areas. For text documents and executable programs as well as in some media formats like PNG for images. Lossless compression is applicable for filesystems as it can be used transparently. ZFS supports transparent lossless compression with various algorithms. In following the use and functioning of these compression algorithms are described.

Zero Length Encoding (ZLE)

Run-Length Encoding (RLE) is a very simple compression algorithm. Generally it will search through the data to find sequences of repeating data values. A detected sequence is replaced by a tuple of an identifier for the data value and the length of this sequence. So for example the sequence `aaaabbb` would be stored as `a4b3`. For bit patterns this algorithm simplifies because only two different values are possible. Hence, only the length of a sequence has to be stored. For example the sequence `1111011000` would be stored as `4123`. Additionally the first value of the first sequence has to be saved for decompression. Mostly this algorithm is done as preliminary step for other compression algorithms.

ZFS supports a variation of this algorithm which only compresses continuous runs of zeros called ZLE.

Lempel Ziv 4 (LZ4)

The compression algorithms LZ77 and LZ78 by by Abraham Lempel and Jacob Ziv are the bases of many widely used compression algorithms. The LZ77 algorithm builds up a dictionary of sequences.

It replaces repeated occurrences of sequences with a reference to the same sequence earlier in the uncompressed data stream. Instead of the reoccurred sequence a triple is stored with position of the sequence in the uncompressed data, the sequence length and the first differing symbol [wikc].

	(pos, len, char)
abrakadabra	(0,0,a)
a brakadabra	(0,0,b)
ab rakadabra	(0,0,r)
abr akadabra	(3,1,k)
abrak adabra	(2,1,d)
abrakad abra	(7,4,-)

Table 2.1: Example LZ77 compression of `abrakadabra` [lz7].

Table 2.1 shows an example of the LZ77 compression with the word `abrakadabra`. The first three chars are new, and thus have address and sequence length 0. The following `a` already occurred three chars to the left and the next char is a `k`. Hence the following triple is `(3,1,k)`. The final sequence `abra` begins 7 positions to the left and leads to a triple `(7,4,-)`.

LZ4 belongs to the family of Lempel-Ziv algorithms and was developed by Yann Collet [Col11]. It's high compression and especially decompression speeds made LZ4 a widely used algorithm. Since version 3.11 the Linux kernel supports LZ4 natively [Cor11]. In ZFS it replaced the former default compression algorithm LZJB. It outperforms LZJB especially on incompressible data and decompression [Kis13].

GZIP

The GZIP algorithm was created by Jean-Ioup Gailly and Mark Adler. The first release was on 31 October 1992. The newest stable release is version 1.8 which was released on 26 April 2016. GZIP is released under the GNU General Public License.

GZIP is based on the DEFLATE algorithm. GZIP first compresses the data with the LZ77 compression. The output is compressed again using Huffman coding.

Huffman coding reduces the data by representing frequently occurring symbols (or sequences) with a lower number of bits than less frequent symbols. Figure 2.1 shows an example of a Huffman code for the sequence `ABABBAACCAA`.

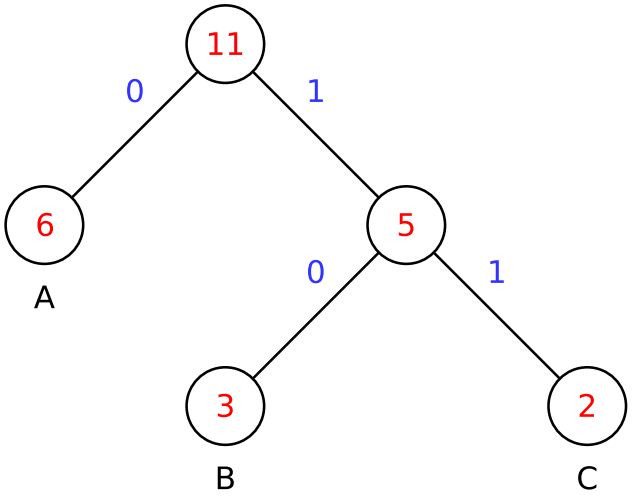

Figure 2.1: Huffman tree generated from the sequence `ABABBAACCAA`.

For computing the Huffman encoding the algorithm builds up a tree. First the number of occurrences for each symbol are counted. In this case A occured six, B three and C two times. Then a parent node is created for the two least frequent occurring symbols. The numbers of those symbols summed up. Now A and the parent of B and C are the least frequent. For those another parent node is created. Each branch to the left is assigned with a "0" and each to the right with a "1". This leads to the code "0" for A, "10" for B and "11" for C. This way instead of representing each letter with a 8-bit American Standard Code for Information Interchange (ASCII) encoding the symbols can be represented with one bit (A) or two bits (B and C) [huf].

Due to using two different algorithms sequentially on a file the GZIP algorithm has relatively slow compression and decompression speed but on the other hand provides better data reduction than algorithms only using compression based on LZ77 like LZ4. The GZIP algorithm offers different levels as options for compression ranging from 1 to 9, whereby higher levels provide better compression at the expense on runtime.

Comparison of compression algorithms

Table 2.2 shows a comparison of the compression algorithms supported by ZFS. The algorithms were applied to a large set of input data from the Max Planck Institute's Global Ocean/Sea-Ice Model (MPI-OM) [ML15]. The simple ZLE algorithm yields the shortest runtime but on the other hand is only able to save around 10% of the data. LZJB and LZ4 were developed to provide fast compression and decompression times, and thus have a significantly worse compression ratio than GZIP-1. GZIP-1 has the drawback of more than doubling the processor utilization. The highest compression level of GZIP provides only a slightly improved compression ratio at the expense of a high runtime.

Algorithm	Compression Ratio	Processor Utilization	Runtime Ratio
none	1.00	23.7%	1.00
ZLE	1.13	23.8%	1.04
LZJB	1.57	24.8%	1.09
LZ4	1.52	22.8%	1.09
GZIP-1	2.04	56.6%	1.06
GZIP-9	2.08	83.1%	13.66

Table 2.2: Comparison of compression algorithms supported by ZFS. Table taken from [ML15].

However the presented comparison only covers a single data set. For differing data the compression ratio might also differ significantly. For some data mainly containing zeros ZLE compression might be enough. Which algorithm is the best for a data set not only depends on the data but also on the goal. When performance is important a algorithm with a short runtime is preferred. On the other hand when high compression rates are the goal a compression algorithm providing that is the better choice.

Therein lies a wish to chose the algorithm adaptively based on the data or focus of the compression. In the thesis [Ehm15] a prototype of adaptive compression for ZFS was implemented. The compression algorithm is chosen based on the file type and three modes: *performance*, *energy* and *archive*.

2.2 Lustre

Lustre is a distributed file system licensed under GNU. It is widely used among the TOP500 supercomputers. In 1999 development of Lustre started as a research project by Peter Braam with the release of the first version being in 2003. In 2007 Sun Microsystems bought Peter Braam's Cluster File Systems corporation and continued the development of Lustre until Oracle ceased the development after they acquired Sun Microsystems. The work on Lustre was then continued by other companies like Xyratex and Whamcloud. The latter was afterwards acquired by Intel which continued working on Lustre. Xyratex

purchased the original Lustre trademark, logo, website and associated intellectual property by Oracle. Currently many organizations are working on Lustre to integrate new features and improvements of Lustre. In December 2016 the latest version of Lustre 2.9.0 was released. This release of Lustre for example introduced barge bulk I/O to get a more efficient network to disk I/O. Also additional serverside advise and hinting about the nature of the data access were implemented [lusb]. For the next release 2.10.0 snapshot support for Lustre using the ZFS backend is in development [lusa]

2.2.1 Basics

Lustre is a kernel file system developed to enable high performance on thousands of nodes and petabytes of storage. On serverside Lustre only supports enterprise kernels like Red Hat Enterprise Linux (RHEL) and Community Enterprise Operating System (CentOS). In contrast to the clientside which also supports newer kernels.

Traditional network file systems are only able to use one central server that is connected via one single network path to the clients. The obvious backdrops of this approach are that the server is a single point of failure and the bandwidth has a limited scalability. For this reason Lustre was designed to support distributed servers that can be accessed in parallel.

2.2.2 Architecture

Lustre consists of three main functional units which are Metadata Server (MDS), Object Storage Server (OSS) and clients. In addition to that a single Management Target (MGT) is connected to one Management Server (MGS). As Figure 2.2 shows Lustre allows to have different number of servers as well as volume sizes for OSTs and MDTs.

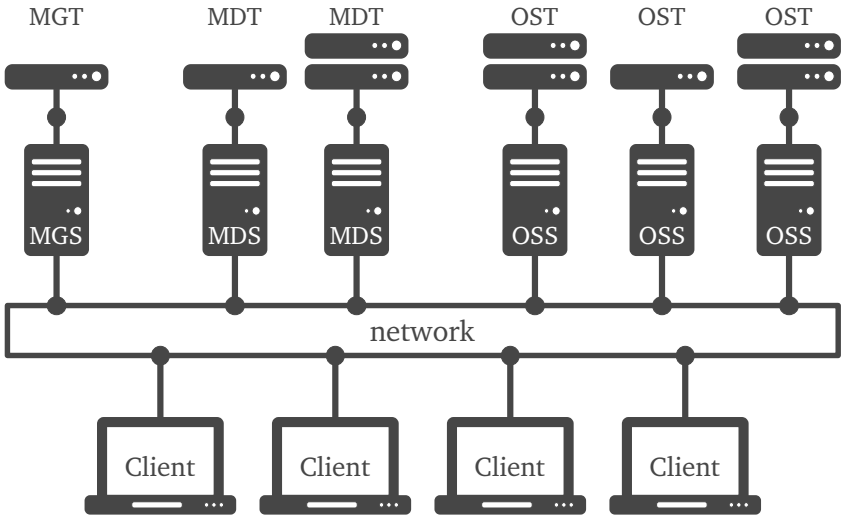

Figure 2.2: Lustre architecture. (Graphic taken from [Fuc16])

Management server

The MGS stores and retrieves the global configuration of Lustre cluster systems from the MGT. The configuration contains information like network identifiers and target lists. It also acts as a registration point for all components. Often the MGS is paired on one machine with an MDS and shares its storage. But due to special critical recovery issues the MGT should not be located together with metadata targets.

OSS

Lustre is an object-based file system. The object storage service consists of nodes called OSS and storage devices called Object Storage Target (OST). They are used for storing and retrieving data in Lustre. For a single file system Lustre allows up to 1000 OSS nodes with up to 32 OSTs per node. The total count is limited to 4000 OSTs [lusc].

Metadata server

Like the OSS the metadata services are made up of two components, nodes and storage devices. The MDS is a node that manages all metadata operations. It stores names and location of files and directories. Up to 256 MDSs and up to 256 MDTs per file system are supported.

Client

Up to 100000 clients can communicate with one Lustre system. Clients run computation, visualization or desktop nodes that mount Lustre and communicate with the filesystem's servers. The clients run a Portable Operating System Interface (POSIX) compliant file system that never has direct access to the underlying file storage. But still the files can be handled on mounted clients as if they were stored locally. Since Lustre provides storage a client does not need to have its own. Lustre supports transaction logic as well as logs for redoing and recovering operations.

2.2.3 Internal functioning

This section gives a basic overview of the functionality of Lustre. Clients have no access to the on-disk file structure of the servers. Clients first send a request like a read or write operation to the MDS. The MDS then sends it's respond back to the client. Now the MDS is able to grant permission for the requested operation. Subsequently the required locks, metadata attributes and the unique file layout are returned to the client.

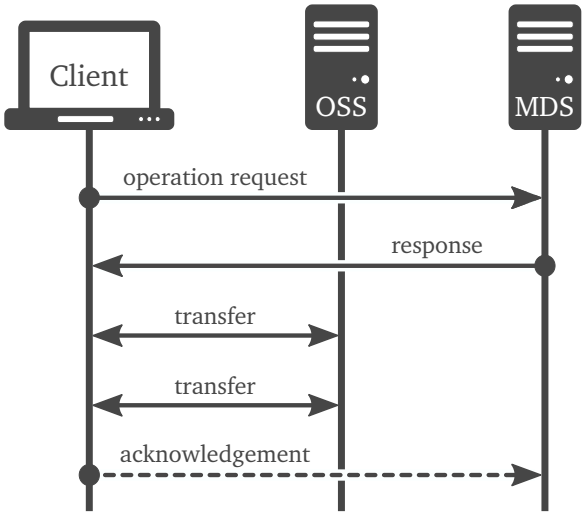

Figure 2.3: Request processing in Lustre. Picture taken from [Fuc16]

File striping

In Lustre file data is distributed in file stripes across multiple OSTs. This increases throughput because I/O operations are accessed in parallel. Lustre offers various options for striping the data over OSTs. The user is able to either specify the number of OSTs to stripe the data over or give a list with specific OSTs to use. It is also possible to state that every OST is used. As default files are not striped and stored on a single OST. The stripe offset determines which OST is used first during the striping process. Per default a random OST is selected. Additionally one is able to set the size of a stripe in multiple of 64KiB. Here the default is set to 1 MiB. These options are configurable per file, directory, tree or data pool. However they cannot be changed once the file is created.

Backend

For storing the data locally on metadata and object servers Lustre uses an underlying local file system. For that reason Lustre uses a layer called Object Storage Device (OSD) which is fully transparent to the client. The OSD provides object storage services for either ldiskfs or ZFS. ldiskfs is a Lustre-patched version of the Extended File System 4 (ext4) filesystem. All allocation and block metadata is handled by the OSD for the underlying filesystem [EB14]. Targets are formatted to either one of the local filesystems. It is also possible to use combinations of these filesystems for different servers. Lustre is able to use the features like compression, deduplication or snapshotting provided by ZFS.

2.3 ZFS

2.3.1 Basics

The development of ZFS started in 2001 by Matthew Ahrens, Bill Moore and Jeff Bonwick before the source code was released in 2005 [his]. Originally ZFS was developed by Sun Microsystems for the Solaris operating system. Three years later ZFS was issued in FreeBSD 7.0. After Oracle bought Sun Microsystems contribution to the source code for ZFS was stopped. In the same year Illumos was founded as the truly open successor to OpenSolaris and the development of ZFS continued as a open-source project. In 2013 the native Linux support of ZFS started. Also the different ZFS projects for all operating systems banded together to form OpenZFS. The goal of this is to reduce code differences and coordinate development between the different platforms.

ZFS is a modern 128-bit filesystem. It is a transactional copy-on-write filesystem designed for easy administration and large capacities. In difference to traditional filesystems ZFS offers many additional features and functions. In the following the most important are listed and described.

Copy on write

ZFS is realized as a hashtree of blockpointers with the actual data as its leaves. For every block a checksum is computed. At every read the integrity of the block is verified. To make metadata reconstructible copies of the data are held by ZFS.

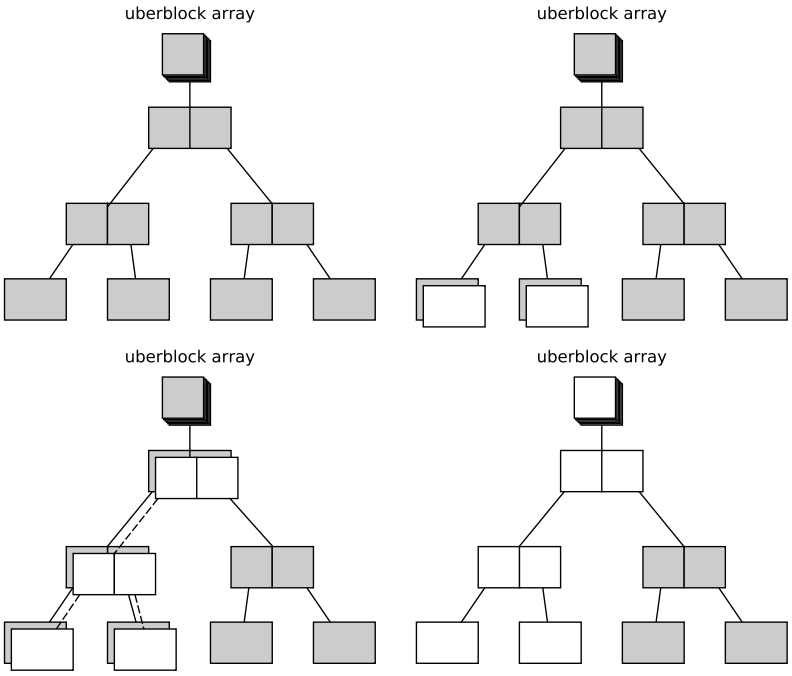

Figure 2.4: Copy-on-write process in ZFS.

ZFS holds an array of 128 uberblocks which is used in a round robin manner. If the filesystem is mounted the uberblock with the highest transaction number is used while the integrity of the uberblock is checked with the checksum. This array is replicated multiple times throughout the pool to ensure data integrity. Data in ZFS is structured as a Merkle tree. A Merkle tree labels every non-leaf node with the hash of the hashes of its child nodes. ZFS starts a copy-on-write with creating modified copies of the affected leaves which represent the data blocks. Then the hashes of the leaves and higher levels up the uberblock are computed. In the last step ZFS updates the uberblock in an atomic operation. This procedure guarantees that ZFS is never in an inconsistent state.

Pooled storage

Traditional file systems reside on a single physical device. These filesystems need an volume manager in order to span over multiple devices or to provide data redundancy. This additional layer between the filesystem and the physical storage devices not only increases complexity it also detains the filesystem from knowing the physical placement of the data on the virtualized volumes.

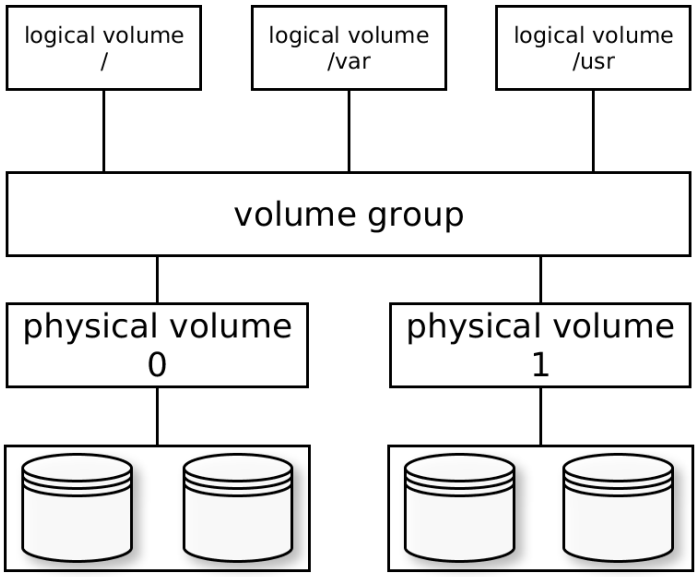

Figure 2.5: Architecture of a traditional filesystem.

Figure 2.5 shows an example of a Redundant Array of Independent Disks (RAID) architecture created with a traditional filesystem. Two RAIDs are created consisting of two disk each. With the volume manager these RAIDs are then merged into a physical volume. Which again are pooled into a single volume group (VG). This VG can be concatenated together into logical volumes (LV)s. LVs are virtual disk partitions that act just like raw block devices and can be used to create file systems on them [log].

In difference to this approach the volume manager is integrated in ZFS. ZFS filesystems use virtual storage pools called zpools. A zpool consists of vdevs, which are virtual

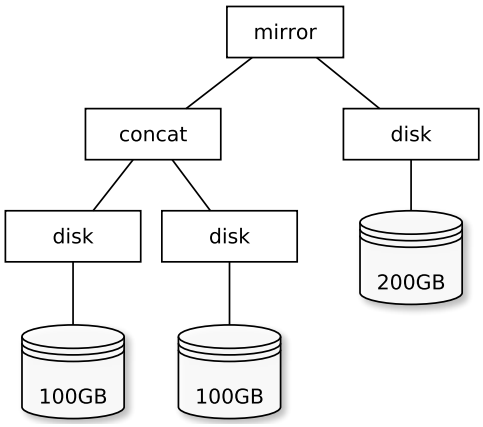

Figure 2.6: Example of a virtual device.

devices that are either a physical device or a composition of them. These compositions can be either a mirror or a RAID system. Vdevs are structured in a tree with physical disks as leafs. Figure 2.6 shows an example of such a tree. Here two 100GB disks are concatenated. The concatenated disks and another 200GB disk are making up a mirrored storage.

This simplifies the creation of a filesystem architecture. The architecture shown in figure 2.7 is able to describe the same architecture as figure 2.5. All four disks are added to the zpool. At the same time each two disks are configured to two RAIDs. One is able to create new sub-filesystems on top of the zpool.

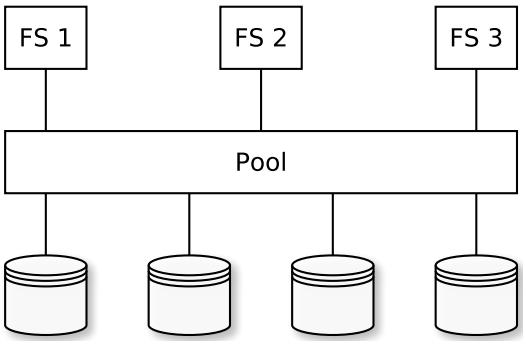

Figure 2.7: ZFS pool architecture

Capacity

Originally ZFS stood for "Zettabyte File System" referring to the potential capacities on zettabyte level possible because of the 128 bit architecture. In the following theoretical limits in ZFS are listed [ZFSa].

Entries per directory:	2^{48}
Maximum size of a single file:	2^{64} bytes (16 EiB)[1]

Maximum size of a pool:	2^{78} bytes (256 ZiB)[2]
Maximum number of devices:	2^{64} bytes
Maximum number of pools:	2^{64}
Maximum filesystems per pool:	2^{64}

RAID-Z

ZFS maintains its own software implementation of RAID systems for compositions of virtual devices. Additionally to mirroring the data ZFS offers RAID-Z, RAID-Z2 and RAID-Z3 as options. For all RAID-Z systems uses striping over all devices to increase throughput similar to RAID 0. RAID-Z works similar to RAID 5 using distributed parity bits while RAID-Z2 is similar to RAID 6 using double parity. Since 2009 RAID-Z3 using triple-parity was implemented in ZFS [Lev09]. Due to the transactional copy-on-write semantics and the usage of dynamic RAID stripe width ZFS overcomes the problem of write-holes [Bon05a].

Snapshots and clones

Due to the copy-on-write property creating snapshots and clones becomes easy and cheap. The read-only snapshot can be created by simply not dropping the reference to the blocks and pointers. These snapshots can be rolled back and used as backups. Clones can be created from snapshots. In contrast to snapshots clones are writable and can be promoted to a filesystem which will turn the current filesystem to a clone.

Data integrity

Data integrity plays a major role in the design of ZFS. Compared to filesystems like ext4 which checksums only for metadata ZFS uses checksums for metadata as well as data blocks. Either Fletcher-based checksums or Secure Hash Algorithm (SHA)-256 hashes are available. As explained in 2.4 the checksums are creating a Merkle-tree. Also as shown in 2.8 checksums are stored in the block pointer rather than at the actual block. Thus, every block in the tree contains the checksums for all its children. All blocks can be checked against the checksum of their parent. The parents checksum is validated by its parent. This ensures end-to-end data integrity. When there is a mismatch of the data and the checksum in the pointer, it is known that the checksum is correct and the data corrupted. The checksum was already validated by its parent's checksum. This enables ZFS not only to detect all kinds of data corruption but also to repair it. If a redundant copy of that block exists in a mirror or RAID-Z system ZFS tries to access the copy of that block and heals the corrupted block by overwriting it with the accessed copy [Bon05b].

This is an important advantage to RAID systems alone. Some errors for example through HDD malfunctions are unnoticed and unrecoverable without end-to-end data integrity. Such errors are called silent data corruption. A study by European Organization

for Nuclear Research (CERN) showed that over six months and involving about 97 petabytes of data 128 megabytes became permanently corrupted [LNB07].

Data reduction

ZFS supports transparent compression of data. At the moment ZLE, GZIP, LZJB or LZ4 can be enabled for all data at filesystem level. Furthermore deduplication is supported. Reoccurring records are stored once and referenced. In case deduplication is activated ZFS changes the checksum algorithm to SHA-256 to correctly detect the duplicate blocks. The downside of deduplication is that a table containing the blocks and checksums has to be held in main memory. This table has to be accessed for every write operation.

ZFS design trade-offs

To close the listing features the trade-offs made in the design of ZFS are shortly discussed. The focus of ZFS was to provide data integrity, recoverability and easy administration. Because of that ZFS checksums everything to ensure better integrity. However this extensive checksumming costs performance. The pooled storage provides easy administration. Because of the copy-on-write property ZFS accumulates data in memory and thus is able to combine many small random write operations into one large sequential operation. On the other hand copy-on-write requires smarter block allocation and leads to bad throughput performance for nearly full disks. For the best performance at least a quarter of a pool should be free. In contrast to overwriting filesystem which still work well when 95% of a disk is full. That all additional features like the RAID-Z system and the volume manager are integrated into ZFS the implementation still is relatively simple. In the CPU and memory extensive design of ZFS it was assumed that many fast 64-bit CPUs combined with large amount of memory are available. If this prerequisites are fulfilled ZFS is able to manage even enormous filesystems performantly. On the downside it is problematic to use ZFS on smaller systems using 32-bit CPUs with less than 8 Gbyte memory and one small nearly-full disk.

2.3.2 Internals

This section introduces the internal structure and architecture of ZFS. First a detailed description of the ZFS block pointer is given. Afterwards the modules of ZFS and their purpose are described. Followed by a presentation of the organization of ZFS.

Block pointer

The ZFS block-pointer is a 128 byte structure. Its contents are shown and in figure 2.8.

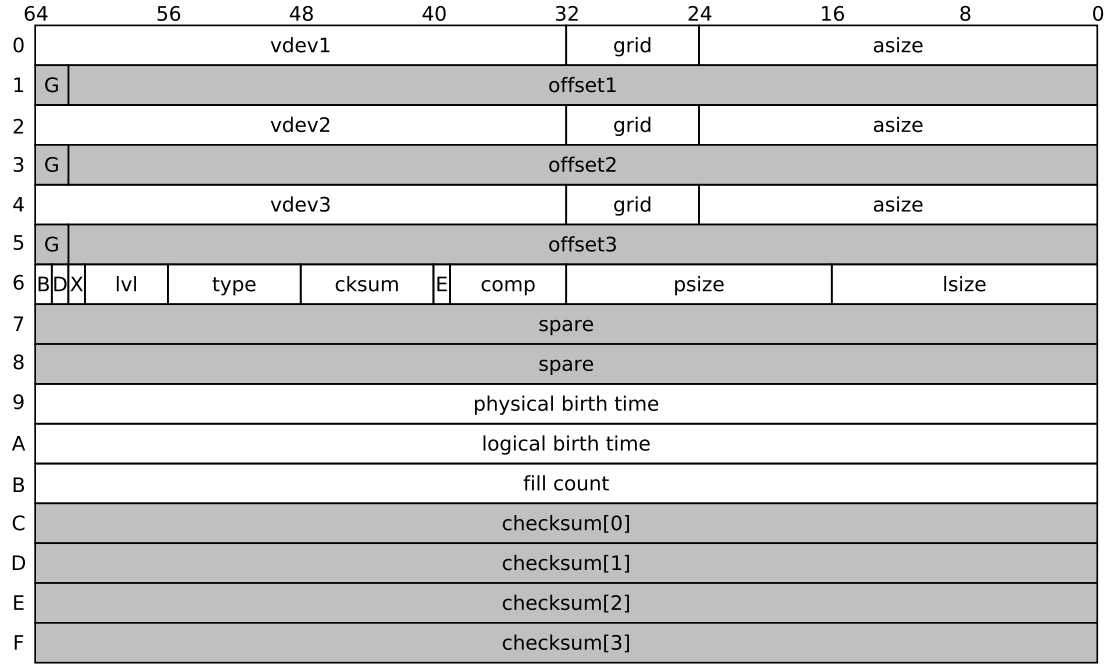

Figure 2.8: Block pointer

vdev	virtual device ID
offset	offset into virtual device
LSIZE	logical size
PSIZE	physical size
ASIZE	allocated size
GRID	RAID-Z layout information (reserved for future use)
cksum	checksum function
comp	compression function
G	gang block indicator
B	byteorder (endianness)
D	dedup
X	encryption
E	blkptr_t contains embedded data
lvl	level of indirection
type	Data Management Unit (DMU) object type
phys birth	txg of block allocation; zero if same as logical birth txg
log. birth	transaction group in which the block was logically born
fill count	number of non-zero blocks under this bp
checksum[4]	256-bit checksum of the data this bp describes

Block sizes

The block pointer stores three different sizes. The logical size (lsize), the physical size (psize) and the allocated size (asize). All are stored in terms of 512 byte sectors. Whereby

a size equal 0 represents 512 bytes, size 1 represents 1024 bytes and so on. The exact size in bytes is rounded up to the next sector. If compression is turned off the lsize always equals the psize.

lsize and *psize* are 16 bit number and therefore theoretically able to store up to 32 MiB. The *asize* is a 24 bit number and consequently capable of storing larger values. This is necessary because of RAID-Z parity and gang block headers that possibly end up increasing the block size that needs to be allocated. Also the *asize* may vary for each data copy and hence is stored for every copy that is allocated.

Data virtual address

The combination of the vdev and the offset make up a Data Virtual Address (DVA). A DVA uniquely identifies the block adress of the data it points to. The physical block address is specified as:

$$physical\ block\ address = (offset << 9) + 0x400000 \tag{2.1}$$

The offset is multiplied by $2^9 = 512$ because it is stored as the number 512 byte blocks. Additionally $0x400000$ which represents 4 Mbyte is added to take two vdev labels and the boot block at the beginning of the vdev into account. The block pointer is able to store up to three unique DVAs and thus three copies of the data pointed to by the block pointer. Per default ZFS will replicate metadata and thus two of the three DVAs set. Furthermore filesystem may be configured to replicate file data. In this case metadata will be triplicated. The number of of DVAs used per block pointer is called "wideness".

Gang blocks

The *G* stands for the gang block bit. This bit indicates that the block pointed to is a *gang block*. Gang blocks are used when there is not enough space available to store a contiguous block. In this case the block will be portioned into several smaller blocks. Gang blocks are handled transparently and will be depicted as a single block to requesters.

Modules

Figure 2.9 shows all modules and their connection to one another. The modules of ZFS can be assigned to three layers, the Interface Layer, the Transactional Object Layer and the Pooled Storage Layer. In the following ZFS' individual modules are described.

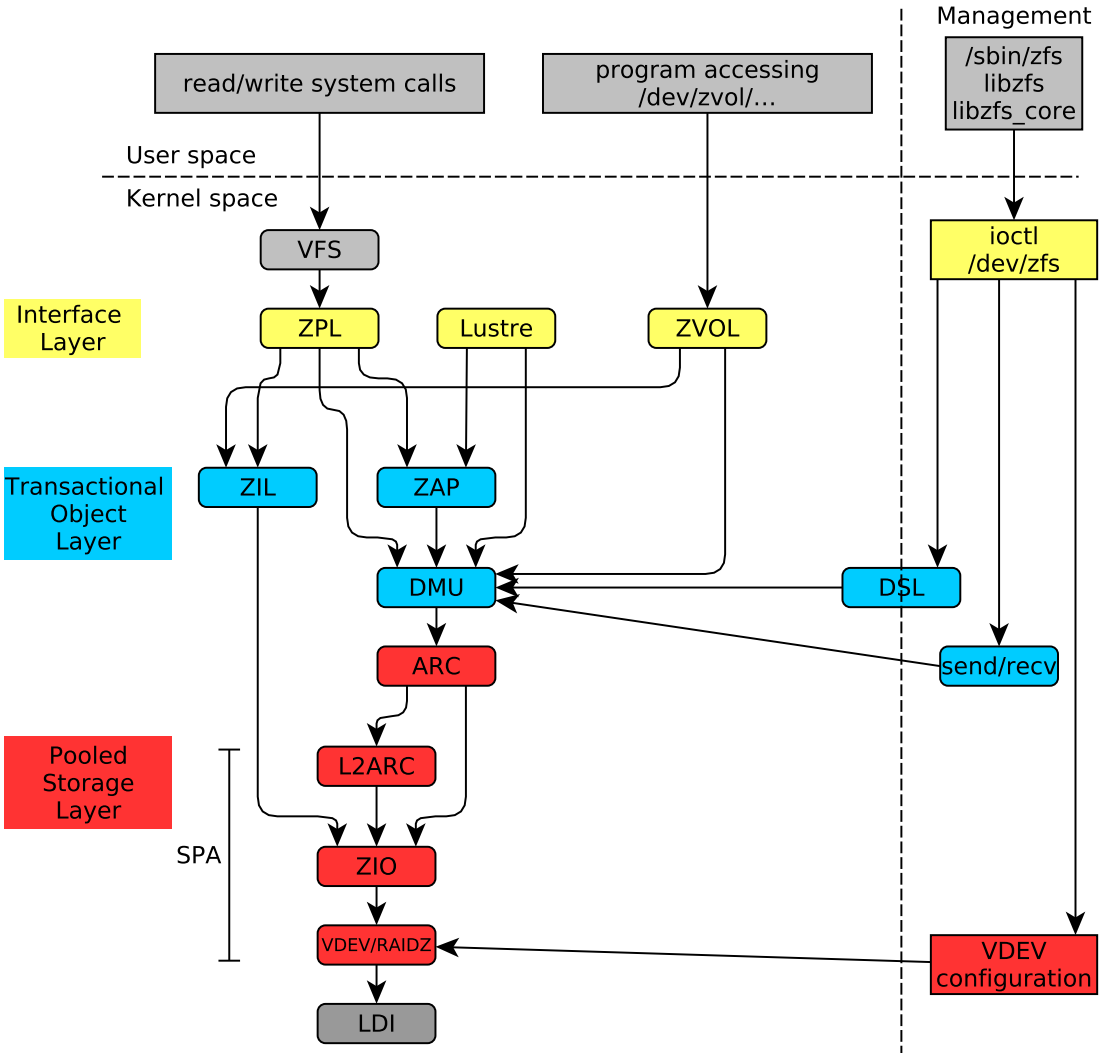

Figure 2.9: Modules of ZFS and their association to each other. (Based on graphic from [MKM14].)

Virtual File System (VFS)

The Virtual Filesystem creates an abstraction layer for filesystem operations. User programs specify their operations according to this generic interface. All file systems provide an implementation for the VFS interface. Thus operating systems like Linux are able to support multiple filesystems [Ker10].

ZFS Posix Layer (ZPL)

The ZPL implements the POSIX compliant VFS Interface. It provides a filesystem abstraction of files and directories. It manages the tree-structured directory structure. Also it manages the znode which keeps track of the metadata required to support POSIX semantics.

ZFS Intent Log (ZIL)

ZFS always remains in a consistent state. Hence ZFS does not need to log information that is needed to bring a filesystem back into a consistent state. But still for changes that need to be persisted, for example due to a fsync call, need to be logged. In this cases changes can be lost due to a panic or power failure. Hence the ZFS intent log was introduced. It saves all system calls on a ZPL-level that change will change files in memory. Enough information is saved to replay these changes to disk. Logs can be removed as soon as a checkpoint was committed to disk. They only need to be flushed to the stable log (SLOG) when required due to a fsync or other synchronous operations.

ZFS Volume (ZVOL)

ZVOLs are logical volumes that appear as traditional fixed-size disk partitions. They can be used like a normal block device.

ZFS Attribute Processor (ZAP)

The ZFS Attribute Processor is a module on top of the DMU. A ZAP object is a key-to-value hashtable foremost used to map directories to their object number and thus to the location on disk. Since it provides fast entry lookup, new entry insertion, old entry deletion and full-directory scanning ZAP objects are also used for other metadata like dataset and storage pool properties.

ARC and Level 2 Adaptive Replacement Cache (L2ARC)

Instead of a Least Recently Used (LRU) ZFS uses an extended ARC that was proposed in [NM03] by Megiddo and Modha. A LRU uses a single list of elements that inserts new elements at the beginning and entries at the end of the list are evicted. In difference to that ARC uses four lists. The Most Recently Used (MRU), Most Frequently Used (MFU) and a so-called ghost list for both MRU and MFU. The ghost lists are used to track recently evicted objects. They are not holding the data itself but provide metadata about the block [Wika].

If a block is accessed it is added to the top of the MRU list. New entries in the MRU will push other caches to the bottom until they are evicted and added to the MRU ghost list. If an entry of the MRU cache is accessed again it will be moved to the top of the MFU cache. New entries will push other entries to the bottom just like in the MRU list. Subsequent hits will push an entry in the MFU list back to the top of the list. The

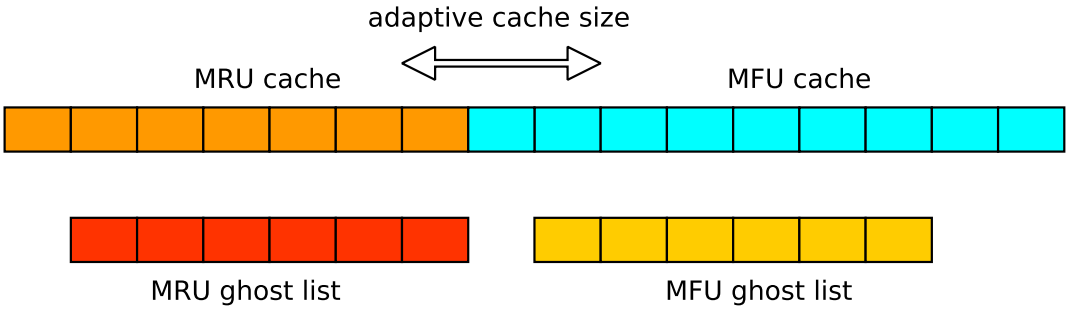

Figure 2.10: ARC

ARC cache as a whole has a fixed size but the cache sizes of the MRU and MFU lists are adaptively adjusted. Hits in the ghost list of the MRU will increase the size of the MRU cache because it shows that least recently used elements are evicted too early. The same goes for the MFU ghost list [Eva14].

Modifications of the ARC for ZFS

ZFS uses a modified variant of the ARC. In difference to the model by Megiddo and Modha in ZFS elements can be made unevictable. This is the case if there is a active reference to the object. So a cache might be unevictable but lowest in the list. In this case the entry lowest in the list that is evictable is chosen. This can lead to the situation that not enough space for new caches can be freed. Thus, the flow of new data is slowed down.

In addition to that the ARC in ZFS has a variable cache size as well as variable block sizes. The cache size increases with high use. Cache size will decrease if system memory is tight. Because of the variable block sizes caches might not be replaced one to one. Instead as many blocks are evicted to approximate the needed space as closely as possible.

Additionally to the Level 1 Adaptive Replacement Cache (L1ARC) in Random-access Memory (RAM) ZFS is able to use a L2ARC. This cache is stored on Solid-state Drive (SSD)s or other devices with substantially faster read latency than disks. Mainly the L2ARC is intended to increase performance of random read workloads. The L2ARC is populated by a thread that scans the L1ARC and searches for caches that are soon to be evicted from the MRU or MFU list. Data that is read from the L2ARC it is integrated into the L1ARC [arc]

DMU

The Data Management Unit implements the transactional object model of ZFS. It is a general-purpose transactional object store. The copy-on-write property is realized by the DMU. Blocks are consumed from the Storage Pool Allocator (SPA) and exported to objects. These objects are part of an object set. In ZFS object sets are referred

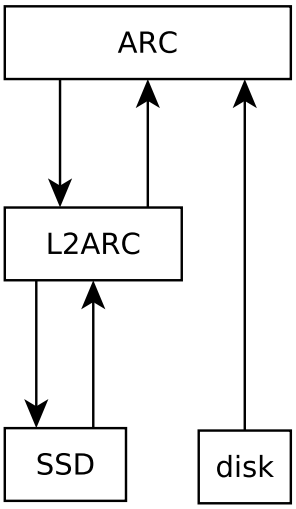

Figure 2.11: L2ARC

to as *objset*. An objset is the set of objects in a filesystem, a snapshot, a clone or a ZVOL. Operations on objects like creation or a writing are always attached to a transaction [Dil10].

Dnode

Dnods are an important data structure in ZFS managed by the DMU. In ZFS the dnode has many use cases. It describes various objects in the Meta Object Set Layer like filesystems, snapshots, clones, ZVOLs, space maps, property lists and dead-block lists. Similar to an inode in traditional filesystems like Unix File System (UFS) or ext4 a dnode also is used to hold metadata for files and directories. In this cases the dnode embeds a znode. The znode is managed by the ZPL and contains the required metadata to support POSIX semantics.

Block size

The block size of datablocks is in ZFS referred to as *recordsize*. It can be set via a filesystem property. ZFS supports recordsizes from 512 bytes to 16 Mbytes. However, the size is limited per default the maximal recordsize to 1 Mbyte. Larger recordsizes would have a negative impact on I/O latency and memory allocation. So one is able to change the recordsizes of one block by a filesystem property from 512 bytes to 1 Mbyte. The default recordsize is 128 Kbyte. For smaller files ZFS is able to dynamically change the record size. Highly compressible files compressed to less than 112 bytes are embedded directly into the block pointer except deduplication is active. For files smaller than the recordsize the dnode will point to one direct block pointer. This block pointer will reference a record with a size that equals the file size. So for example a 42 Kbyte file would be stored in a 42 Kbyte record.

Indirection levels

Bigger file sizes are handled by dnodes analogous to inodes using indirect blocks. If the file outgrows one record a 16 Kbytes indirect block is created that the dnode points to. A block pointer is 128 bytes in size. Thus the indirect is able to reference 128 pointers. If these 128 records are not enough a second level of indirection is introduced. The dnode will reference an indirect block which again will reference up to 128 indirect block. Hence providing place for 128^2 data blocks. For higher file sizes the indirection is increased similarly. For files with more than one block the record sizes are set to the maximum record size at creation time. Therefore if the default record size of 128 Kbyte is used for a 129 Kbyte file two blocks with 128 Kbyte are allocated.

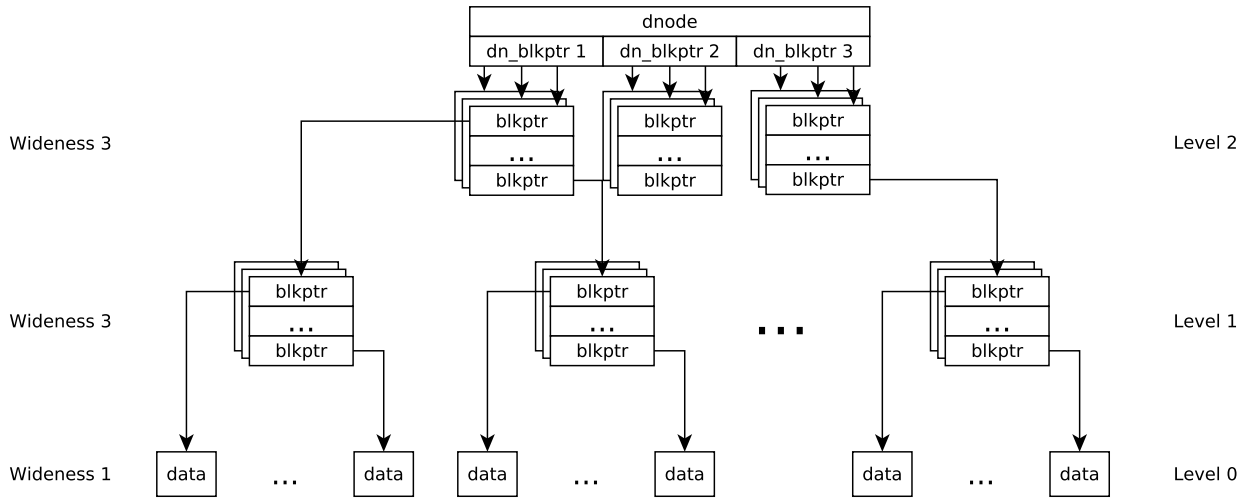

Figure 2.12: Object with 3 levels. (Based on graphic from [SM06].)

Figure 2.12 shows an object with 3 levels. The dnode is able to store at most three block pointers. In this example for metadata the wideness 3 is used. Hence all three DVAs of the dnode and block pointer in level 2 and level 1 are used. They all point to a copy of the same metadata block. Level 2 and 1 are indirect blocks containing the block pointer. For the user data (level 0) a wideness of 1 is used.

Dbuf Cache

Additionally to the ARC cache an LRU cache exists in the DMU layer. It holds the buffers used in the DMU. These buffers are referred to as *dbufs* and are representing one block. It contains dbufs that are not currently held by any caller for reading or writing. If the last hold of a dbuf gets released it is added to the head of the list. Dbufs accessed from the cache get removed from the list and added to the top of the list as soon as they are released. Cached dbufs cannot be evicted form the ARC. Per default the size of the dbuf cache is limited to $\frac{1}{32}$nd of the ARC size. Increasing the cache size has to be done via changing the tunable `dbuf_cache_max_shift` in the source code.

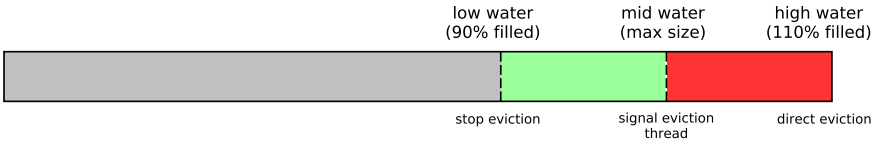

Figure 2.13: The dbuf cache.

Figure 2.13 symbolizes the eviction strategy used by the dbuf eviction thread. The *low water mark* signals that the thread should stop evicting dbufs from cache. When the *mid water mark* is reached the thread will start evicting. Reaching the *high water mark* indicates that callers adding elements to the cache will evict dbufs directly from cache. Per default the *low water mark* is set to 90% of the dbuf cache and the *high water mark* to 110%.

ZFS I/O (ZIO)

The ZFS I/O pipeline is the centralized I/O framework. I/Os are executed in pipelines. This allows to reorder and combine operations to increase performance. Compression, checksums and data redundancy are managed by ZIO. Also the ZIO is responsible for creating gang blocks and filling in the block pointer.

Block allocation

Block allocation is a central part for a filesystem. In ZFS the SPA manages allocation of blocks. Space maps are used to identify free space. A pool is divided up into regions of a fixed size called metaslabs. Usually around 200 metaslabs are used for a disk. When allocating new blocks the SPA first will try to use the disk with the most free space of a pool. On the chosen disk the metaslab providing the highest available bandwidth will be used to allocate the new blocks. Decisive for that is how fragmented the data on the metaslab is. Also the outer regions on a HDD are faster and therefore have a higher bandwidth [Bon06].

ZFS organization

Figure 2.14 gives an overview of how ZFS is organized. The organization of ZFS is based on object sets. Commonly referred to as *objset*. An *objset* describes an array of dnodes. The uberblock points to an object set in the Meta-Object Set (MOS) layer. The MOS is managed by the Dataset and Snapshot Layer (DSL) and the SPA. This object set contains an array of meta-objects like filesystems, ZVOLs, snapshots, clones. It also contains one master object at the beginning of the array and a single space map describing allocated and free blocks of the pool. For creating a new meta-objects like filesystems they just have to be added to the object set.

All blocks of a pool are shared among all meta-objects. Therefore it is possible to move space between filesystems. When a filesystem needs more space the space map

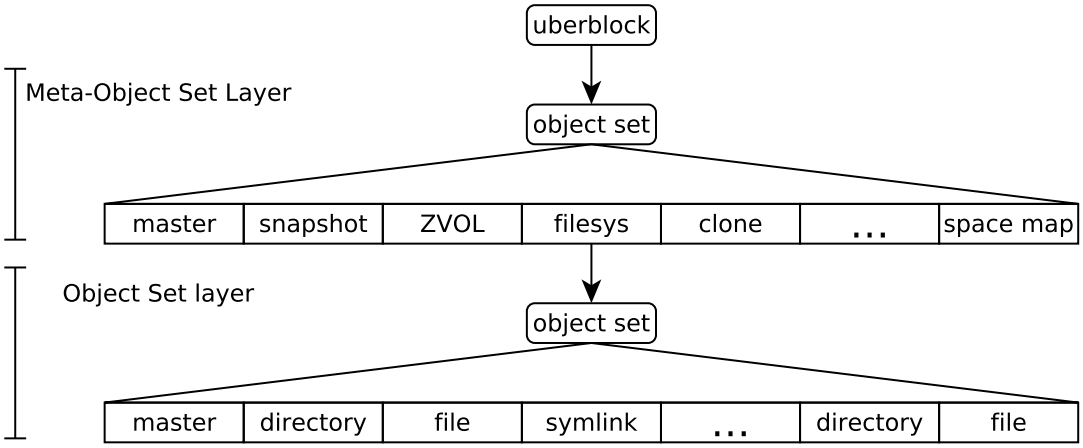

Figure 2.14: Organization of ZFS. (Based on graphic from [MKM14].)

is able to find available free blocks and grant them to the filesystem which requested it. The same way these blocks can be returned to the pool when a filesystem does not need the blocks anymore. This way a filesystem potentially may use all available space of a pool. For this reason it is possible to limit the maximum amount of space usable for a filesystem, or on the other hand set a minimum amount of space reserved for a filesystem.

Meta-objects are themselves object sets which make up the Object Set layer. For example in a filesystem this object set describes an array of files, directories and other filesystem items. The ZPL is responsible for creating the POSIX compliant tree structure from this object set array.

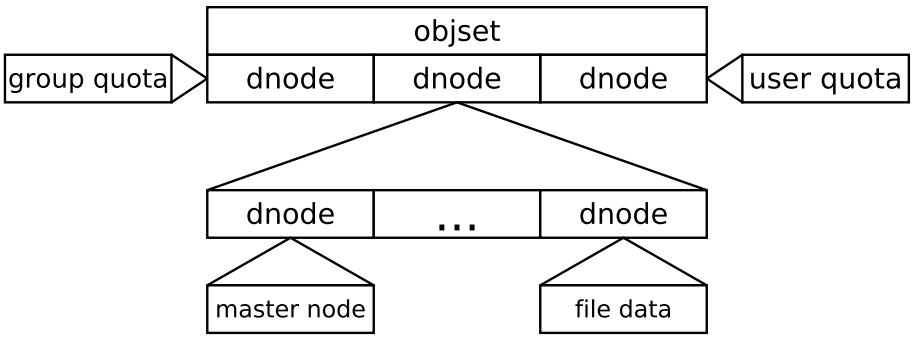

Figure 2.15: Structure of a filesystem. (Based on graphic from [MKM14].)

Figure 2.15 shows the structure of a filesystem in more detail. The object set contains three dnodes. Two of them are ZAP objects that record the user and group space usage for a filesystem. The filesystem's array of files and directories is referenced by the remaining dnode. This array starts with the master node. It is a ZAP object containing the object number of the root inode, the object numbers for the user and group quota

files as well as the set of files hat have been unlinked but still referenced by an open file descriptor. Hence are first removed when they become unreferenced.

2.3.3 Tools

ZFS offers some tools for monitoring and debugging. This section gives a short overview of these tools.

`zpool iostat`

The zpool command for configuring ZFS pools provides through `zpool iostat` a tool to view statistics on the performed I/O operations. Listing 2.1 shows the output of `zpool iostat 1` for a few seconds.

```
1 capacity     operations     bandwidth
2 loc   free   read  write   read  write
3 ---  -----  -----  -----  -----  -----
4 37G  6,57G     17      6  2,18M   605K
5 37G  6,57G      0      0      0      0
6 37G  6,57G    556      0  69,5M      0
7 37G  6,57G    730      0  91,3M      0
8 37G  6,56G    585    120  73,0M  5,35M
9 37G  6,56G    574      0  71,8M      0
```

Listing 2.1: Output of `zpool iostat 1` while reading a file

`arcstat.py`

The arcstat tool prints out ARC statistics [Gre12]. Among many others information about ARC hits and misses or the ARC size can be printed. All fields are shown in listing 1. Listing 2.2 shows an output for `arcstat.py 1` while reading a file.

```
1 time  read  miss  miss%  dmis  dm%  pmis  pm%  mmis  mm%  arcsz     c
2 3:14     3     3    100     3  100     0    0     0    0   115M  845M
3 3:15   680   667     98   667   98     0    0     6   31   200M  845M
4 3:16   624   624    100   624  100     0    0     0    0   278M  845M
5 3:17   844   802     95   388   90   414  100     2   66   377M  845M
```

Listing 2.2: Output of `arcstat.py 1` while reading a file

`arc_summary.py`

The tool `arc_summary.py` shows general and overall statistics for the ARC. Six different pages of information are available. Listing 2.3 shows the output of the first page.

```
1 ------------------------------------------------------------------------
2 ZFS Subsystem Report                Tue Apr 04 16:47:21 2017
3 ARC Summary: (HEALTHY)
4     Memory Throttle Count:          0
```

```
5
6  ARC Misc:
7      Deleted:                  4.24m
8      Mutex Misses:                 1.21k
9      Evict Skips:                  1.21k
10
11 ARC Size:                99.88%  5.85    GiB
12     Target Size: (Adaptive)      100.00% 5.86    GiB
13     Min Size (Hard Limit):       0.53%   32.00   MiB
14     Max Size (High Water):       187:1   5.86    GiB
15
16 ARC Size Breakdown:
17     Recently Used Cache Size:    99.73%  5.85    GiB
18     Frequently Used Cache Size: 0.27%   16.00   MiB
19
20 ARC Hash Breakdown:
21     Elements Max:                 139.28k
22     Elements Current:        34.81%  48.49k
23     Collisions:              180.80k
24     Chain Max:               3
25     Chains:                  595
```

Listing 2.3: Output of `arc_summary.py -p 1`

`dbufstat.py`

Information about the dbuf cache can be obtained with `dbufstat.py`. The dbufs currently in cache can be either printed for each dnode `-d` or for each dbuf `-b`. Additional extended statistics are shown with `-x`. Listing 2.4 shows an output for `dbufstat.py -d`.

```
1  pool  objset      object                          dtype  cached
2  test       0          10              DMU_OT_SPACE_MAP    4.0K
3  test       0          61                  DMU_OT_BPOBJ     16K
4  test       0           1      DMU_OT_OBJECT_DIRECTORY     32K
5  test       0           0                  DMU_OT_DNODE     48K
6  test       0           9              DMU_OT_SPACE_MAP    4.0K
7  test       0           8              DMU_OT_SPACE_MAP    4.0K
8  test      51           0                  DMU_OT_DNODE    656K
9  test      51           7   DMU_OT_PLAIN_FILE_CONTENTS     24M
10 test      51           8   DMU_OT_PLAIN_FILE_CONTENTS    128K
```

Listing 2.4: Output of `dbufstat.py -d`

ZFS Debugger (ZDB)

Zdb is a powerful utility for displaying the on-disk structure of a pool. Additionally some consistency checking can be performed. The option `-C` displays the configuration of all pools, `-u` displays the uberblock of a specified pool [zdb15]. `zdb -b poolname` first verifies all checksums of blocks and prints information about the number and size of blocks. While `zdb -c poolname` does the same for metadata blocks. Information about

datasets is shown with `zdb -d dataset`. For each repetition of the options `-b` or `-d` the verbosity of the output increases [Roc08].

- `zdb -d dataset`: Output list of datasets with some metadata like size and number of objects in dataset.
- `zdb -dd dataset`: Output list of objects within dataset together with for example object ID, logical size or the type.
- `zdb -ddd dataset`: Same output as for `-dd`.
- `zdb -dddd dataset`: This puts out detailed information of every object. For files all POSIX metadata like size, atime and filepath.
- `zdb -ddddd dataset`: Additionally prints the DVAs for every object.

Listing 2.5 shows an excerpt of the output of `zdb -ddddd test`. Here metadata for `/file` is shown. At the top it begins with information about the object number, the indirection level of the file and the data sizes. In this case the logical size of the file is 640 Kbyte and the physical size called "dsize" is 120 Kbyte. In the middle metadata for POSIX compliance are shown. Amongst others the size of the file in bytes (size = 581842 bytes).

```
1  Object  lvl   iblk   dblk  dsize  dnsize  lsize   %full  type
2      12    2   128K   128K   120K     512   640K  100.00  ZFS plain
       ↪ file
3                                               168   bonus  System
                                                   ↪ attributes
4  dnode flags: USED_BYTES USERUSED_ACCOUNTED USEROBJUSED_ACCOUNTED
5  dnode maxblkid: 4
6  path    /file
7  uid     0
8  gid     0
9  atime   Sat Mar 25 16:08:58 2017
10 mtime   Sat Mar 25 16:08:58 2017
11 ctime   Sat Mar 25 16:08:58 2017
12 crtime  Sat Mar 25 16:08:58 2017
13 gen 649
14 mode    100644
15 size    581842
16 parent  4
17 links   1
18 pflags  40800000004
19 rect blocks:
20            0 L1  0:12805ce00:400 20000L/400P F=5 B=649/649
21            0  L0 0:12803f600:1e00 20000L/1e00P F=1 B=649/649
22        20000  L0 0:128041400:1a00 20000L/1a00P F=1 B=649/649
23        40000  L0 0:128042e00:7800 20000L/7800P F=1 B=649/649
24        60000  L0 0:12804a600:4600 20000L/4600P F=1 B=649/649
25        80000  L0 0:12804ec00:e200 20000L/e200P F=1 B=649/649
26
```

```
27      segment [0000000000000000, 00000000000a0000) size  640K
```

Listing 2.5: zdb -ddddd test

Taking the output of the DVAs for a file from `zdb -ddddd pool` the file contents can be displayed. Listing 2.6 shows the contents of a file. `zdb -R pool vdev:offset:size`.

```
1  $ zdb -R test 0:6c380600:200
2  Found vdev: /dev/sdb1
3
4  0:6c380600:200
5            0 1 2 3 4 5 6 7   8 9 a b c d e f  0123456789abcdef
6  000000:  48656c6c6f20576f  726c64210a000000  Hello World!....
7  000010:  0000000000000000  0000000000000000  ................
8  000020:  0000000000000000  0000000000000000  ................
9  000030:  0000000000000000  0000000000000000  ................
10 000040:  0000000000000000  0000000000000000  ................
11 ...
```

Listing 2.6: zdb -R pool 0:6c380600:200

3 Design

This chapter proposes a way how the compressed data stripes can be properly integrated into ZFS. First the implemented prototype for compression in Lustre and the compression functionality of ZFS is described. This motivates and specifies the design and behaviour of the wanted feature to storing pre-compressed data efficiently in ZFS. For that first the desired behaviour is described followed by a characterization of the necessary metadata and the interface for the pre-compressed data.

3.1 Lustre clientside compression

The compression design implemented in the prototype is based on the stripes. A stripe is per default 1 MiB. The stripes are chunked and then each of the chunks is compressed independently. This is done because compressed data is logically coherent and indivisible. Modifications cannot just be written into the compressed data. The stripe whole stripe first has to be read, decompressed, than modified and compressed again before finally writing it back to disk. This problem is shown in figure 3.2.

So by dividing the data into smaller chunks when modifying the data only the affected chunk has to be decompressed. Figure 3.1 shows how compression is conducted on a single stripe. After the stripes are compressed there are two ways to write the chunks. Either approach B to move all chunks together or approach A to leave the chunks at their original place in the stripe. Approach B has the problem that a mapping from

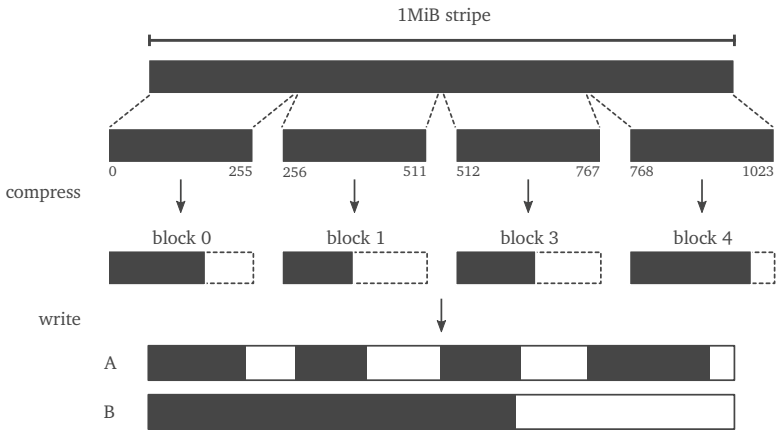

Figure 3.1: Chunking a 1 MiB stripe into 4 sub-stripes and approaches to store them on disk. (Graphic taken from [Fuc16].)

original offsets and the offsets of the compressed data indexes is needed. Also when the size of the compressed chunk changes through modification all following chunks have to be shifted which creates high overhead.

In difference to that the sparse stripe of approach A mitigates this problem. Here only one chunk has to be read, decompressed, modified, compressed and written back to disk. However, when storing the stripe with gaps in between the chunks has to be explicitly handled, otherwise the read performance reduces significantly and no storage can be saved on disk. Unhandled, without any metadata, the underlying filesystem has no knowledge that the data is stored compressed and where a compressed chunk starts and where it ends. To the filesystem the stripe would look like a normal 1MiB data sequence. So either the Lustre server reads in the whole 1 MiB or makes several smaller read operations. But since the underlying filesystem has no information about where the next chunk starts, no readahead can be done. For both the read performance would be worse than storing the data without gaps. But, if the underlying filesystem could handle the compressed sparse stripes the approach is superior.

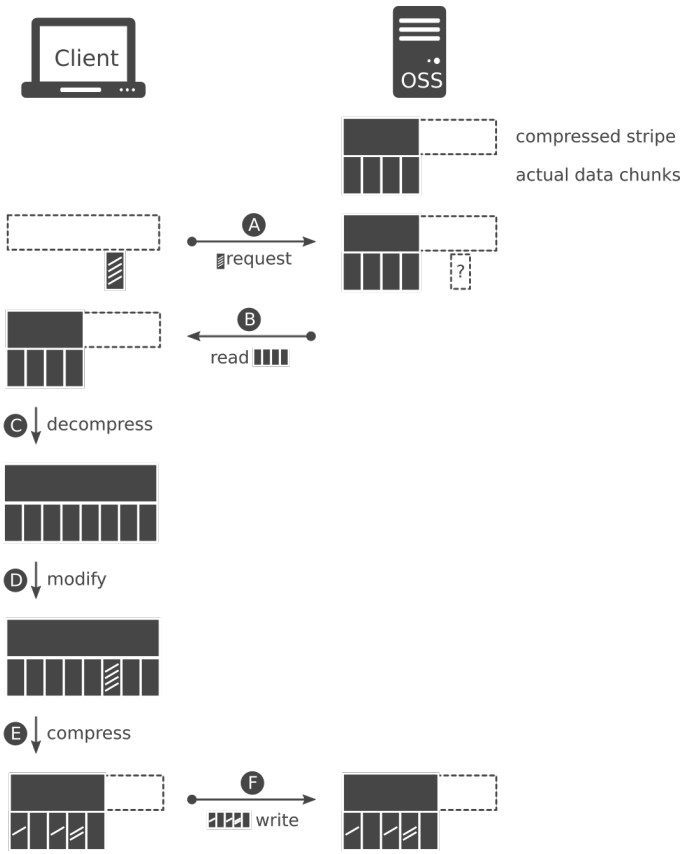

Figure 3.2: Read-modify-write problem with clientside compression support. (Graphic taken from [Fuc16].)

Here the idea is to use the infrastructure of ZFS which is able to handle compressed data. All necessary metadata should be passed to ZFS to make smart readaheads possible.

To figure out how the the compressed chunks are best stored and which metadata is necessary in the following ZFS's compression functionality is explored.

3.2 Compression in ZFS

Compression in ZFS is set with a filesystem property. This way compression is set on a per-filesystem basis.

```
1 $ zfs set compression=on pool/filesystem
```

The option `on` will set the compression algorithm to LZ4. To use other compression algorithms the corresponding value must be specified. Valid are `lzjb`, `lz4`, `gzip`, `gzip[1-9]` and `zle`. With `off` the compression is deactivated. Compression can be either set for an entire pool or for a sub-filesystem by specifing it. This leads to an static approach where every file is compressed with the same algorithm that is currently set. Also changing the compression applies only to newly written data, it does not work retroactive on old data.

3.2.1 Metadata compression

ZFS metadata is compressed with LZ4 per default but can be disabled through the tunable `zfs_mdcomp_disable` in the source code. This might be beneficial if SSDs are used for primary storage which perform better with aligned blocks. Compressed blocks could be unaligned [zfsb].

3.2.2 User data compression

Though compression of user data is set on filesystem-level the actual compression is conducted on data blocks. This means every block is compressed independently. When less than 12.5% of the data or no 512 byte sector is saved due to compression of a block the block is stored uncompressed. These conditions are fixed in the source code.

Storage of compressed blocks

Figure 3.3 shows how the data is stored when ZFS compresses the data. In this example the recordsize is set to 128 Kbyte and 600 Kbyte of data should be either read or written. The data is passed to ZFS in one portion. In buffer the data is divided into fixed size blocks. Before these blocks are written to disk they are compressed separately. On disk ZFS will, if suitable and possible, write sequentially without gaps. When the data is read from disk the blocks are decompressed into buffer and then returned to the user as one portion. Here the size stored on disk is given as the physical size but RAID-Z parity information or gang-block headers might increase the actual data written to disk. Therefore the block pointer additionally stores the allocated size.

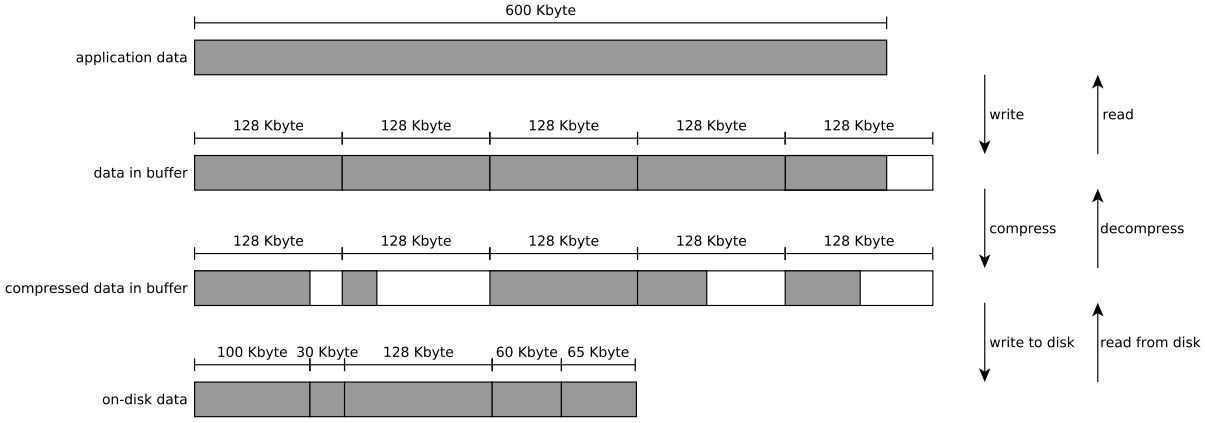

Figure 3.3: Compression behaviour in ZFS.

Metadata of a compressed block

The on-disk information stored to correctly handle compression is stored in the block pointer. There both the used compression algorithm and the physical size are stored.

```
1 0       L1  0:12805ce00:400  20000L/400P  F=5  B=649/649
2 0       L0  0:12803f600:1e00  20000L/1e00P  F=1  B=649/649
3 20000   L0  0:128041400:1a00  20000L/1a00P  F=1  B=649/649
4 40000   L0  0:128042e00:7800  20000L/7800P  F=1  B=649/649
5 60000   L0  0:12804a600:4600  20000L/4600P  F=1  B=649/649
6 80000   L0  0:12804ec00:e200  20000L/e200P  F=1  B=649/649
```

Listing 3.1: Output of metadata for blocks of an example file.

Listing 3.1 shows an excerpt of an output by the ZDB. With this tool one is able to print out information of the data stored in a ZFS pool. This example shows metadata of all blocks of a compressed file with five data blocks and one indirect block. For every block the logical offset in the file, the indirection level, the DMU (`vdev:offset:asize`), the logical size and the physical size, the fill count and the birth time of the block are shown. Table 3.1 lists the variables against the first data block.

It shows that the user data with level 0 is written on disk sequentially without gaps. The indirect block is written afterwards on disk.

ZFS compressed send/receive

The commands `zfs send` and `zfs receive` allow to copy a file system from one system to another system. `zfs send` creates a stream representation of a snapshot that is written to standard output while `zfs receive` creates a snapshot whose contents are specified in the stream that is provided on standard input [Ora]. One advantage of that method is to change properties like the RAID-Z replication of the newly created filesystem. To increase network throughput of the send operation the option `-compressed` was introduced [Kim]. This opens the possibility to send the file data of a filesystem in a compressed state

Variable	Value
Offset in file	0
Level of indirection	L0
vdev ID	0
Offset on vdev	12803f600
Allocated size	1e00
Logical size/physical size	20000L/1e00P
Fill count	F=1
Logical birth time/physical birth time	B=649/649

Table 3.1: Variable listed against the value for the first block in listing 3.1.

without decompressing it. For this purpose allocating and writing a compressed ARC buffer was added to ZFS.

3.3 Functionality for pre-compressed data in ZFS

This section lays out how pre-compressed data can be handled by ZFS. The requirements on the data that is passed to ZFS in order to correctly integrate it into a filesystem. Present new interface that is needed to write and read the data.

ZFS should be able to handle externally compressed data is if it was compressed by ZFS itself. The compression should be flexible and applied on a block level. So a file can consist of uncompressed and compressed blocks. In addition to that it should be possible that the compressed blocks are compressed with different algorithms. ZFS itself should not compress or decompress the blocks. When a block is written to disk it ZFS should not try to compress the block. When a block is read the data should be returned in the same state as it is written to disk. As described in the compression functionality of ZFS the compression will be disabled if not enough space is saved. But since the data is already compressed by Lustre ZFS should be forced to write the data compressed even if only a few bytes are saved.

3.3.1 Behaviour

The flowchart in Figure 3.4 gives an overview of the desired behaviour. The values used are exemplary and could be changed but they reflect the default cases for Lustre and ZFS.

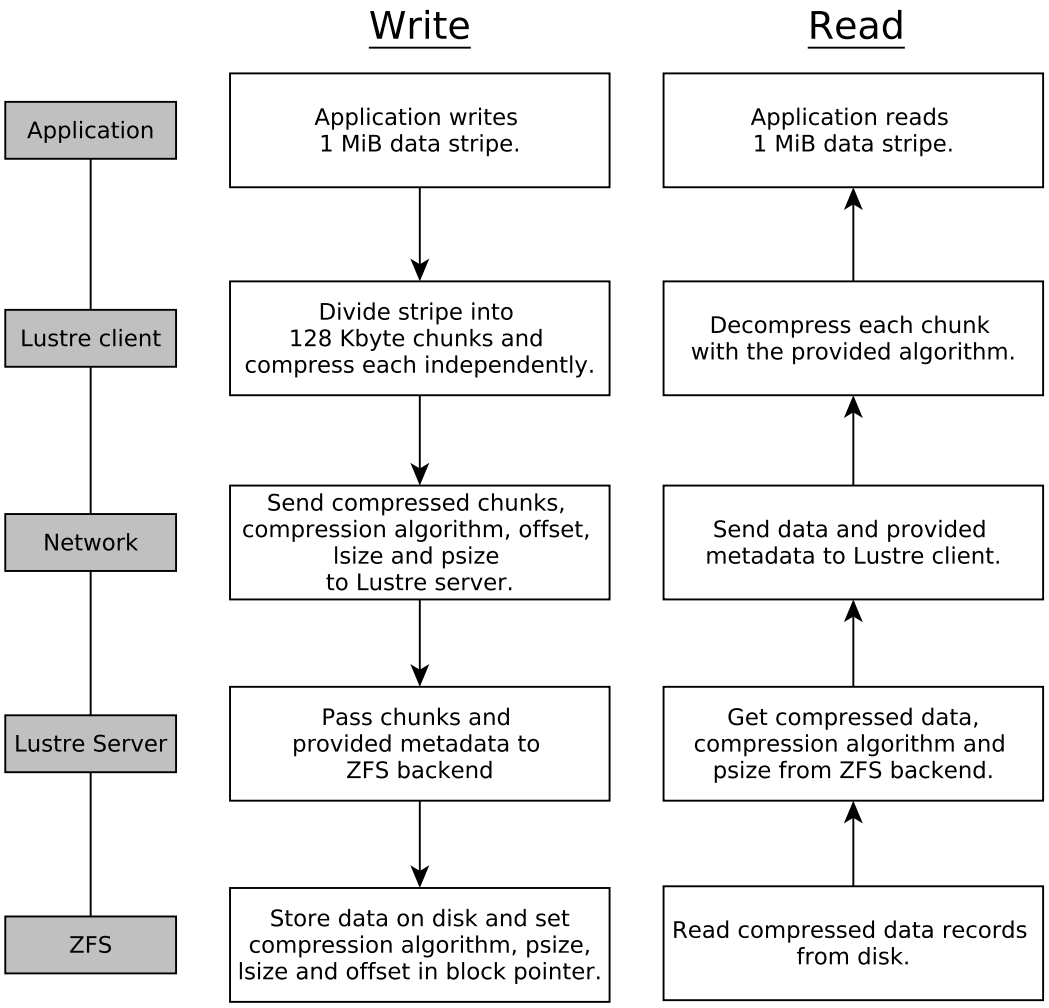

Figure 3.4: Flowchart of the desired behaviour.

A write operation starts with the Lustre client striping the data into 1 MiB parts. These stripes are divided into chunks and each block is compressed independently. The compressed chunks are send over the network to a Lustre server. For each chunk metadata about the compression algorithm used, the offset, the logical size and the physical size are provided. The Lustre server uses the ZFS backend and passes the data chunk together with the metadata to ZFS. There ZFS sets the provided metadata for each record in the block pointer.

For read operations ZFS is able to make use of the physical size. As ZFS knows how many bytes have to be read. With the additional information about the physical size of the block ZFS should be able do readaheads and reach a good read performance. After reading the compressed blocks into the buffer they are passed to the Lustre server. Additionally the physical size and the compression algorithm used for each block are sent to the Lustre server. Afterwards the compressed chunks together with their psize and compression algorithm are send over the network to the client. Here the client makes use of the physical size and the algorithm to decompress each block. Hence the stripe is

uncompressed and can be returned to the application.

Modifying pre-compressed blocks

As shown in Figure 3.2 for updating a compressed chunk the whole chunk is passed to the Lustre client. Where it is decompressed, modified, compressed and then written back to ZFS. So every write operation made to ZFS by Lustre is a whole compressed block. This compressed block always has the same logical size as ZFS's recordsize. Therefore logically overwriting the whole block. Which means that there are no read-modify-write operations within ZFS. For the pre-compressed data an update of a block will replace the old block.

Behaviour of pre-compressed blocks in ZFS

Figure 3.5 shows the behaviour for pre-compressed data records in difference to figure 3.3. The data records are already passed compressed to ZFS and stay compressed in buffer as well as on disk.

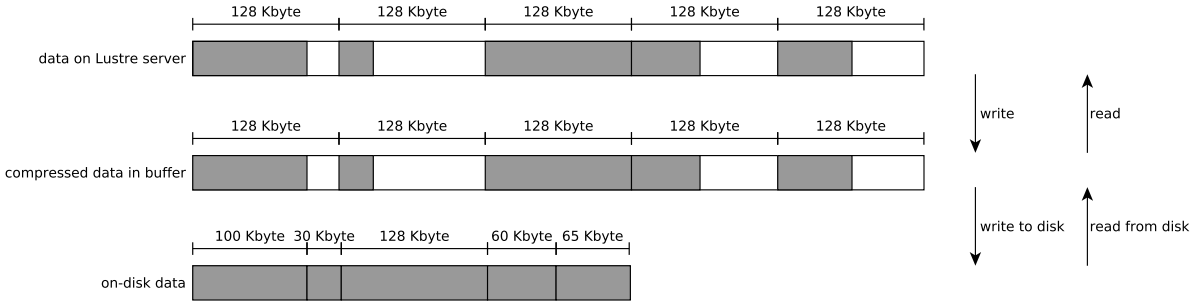

Figure 3.5: Compression behaviour for pre-compressed data in ZFS.

Every compressed chunk should be mapped to a data record. For that there are two main requirements that would need to be fulfilled for every data record.

1. The original uncompressed chunk size of the data for a file has to equal the recordsize in ZFS.

2. The offset for these chunks has to be aligned. So every offset passed to ZFS is a multiple of the recordsize.

Compression specific metadata

For compressed data ZFS internally needs to know the physical compressed size and the compression algorithm used to compress the data. In ZFS all sizes (logical size, physical size, allocated size) in the block pointer are stored in 512 Byte steps. Meaning for example a psize of 2 stored in the block pointer equals 1024 Bytes. When ZFS compresses a block and the algorithm returns the physical in bytes which is immediately

rounded up and stored into the block pointer. But for decompression LZ4 needs the uncompressed size in bytes. Therefore in ZFS the compressed size is encoded at the at the beginning of the block in front of the data. In difference to that practice the other possible algorithm in ZFS GZIP, LZJB and ZLE either store the compressed size transparently or do not need the compressed size for decompressing.

On disk the only size ZFS stores in bytes is the uncompressed size of the whole file. This size is stored in a znode managed by the ZPL. So when reading the file uncompressed it is obvious that all blocks have the same size except the last block which can easily be computed from the file size. But when reading the file with compressed blocks the precise size in bytes of every block needs to be known. As explained this poses a problem because this is only available for blocks compressed with LZ4.

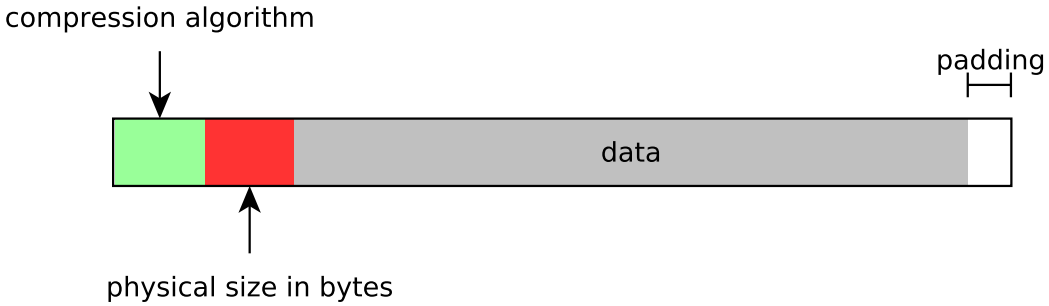

Figure 3.6: Pre-compressed data record with header.

One approach would be to change the block pointer to store the physical size in bytes but changing the on-disk format is a expensive task. The layout of the block pointer would have to be changed.

The better approach is to encode the compressed size in bytes at the beginning of the data block like ZFS is already doing it for LZ4 compression. The other compression algorithms (GZIP, LZE, LZJB) would become incompatible for decompression. Hence they have to be marked as not decompressible. For this issue instead of storing the specific algorithm in the block pointer the block is marked as externally compressed. The downside is that for a possible reader of the compressed block the used algorithm becomes unknown. Therefore the algorithm is also stored in front of the data block. Thus, creating a header consisting of a tag for the used algorithm and the compressed size in bytes. A data record with a header is shown in Figure 3.6.

Recordsize

The recordsize is set in ZFS with a system property on filesystem level. Recordsizes from 512 bytes to 1 Mbyte are possible. Changing the recordsize will only apply to new files.

For files smaller than one record the uncompressed size or logical size is set dynamically. Meaning the file size is rounded up to the next 512 Byte step and set as logical size for the block. If the file is bigger than one record block sizes are immediately set the to the recordsize. So for example a 10 bytes file the lsize is set to 512 bytes. A 130 KByte file with a recordsize of 128KB the logical size for both blocks is set to 128 KB. Thus, ZFS allocates 256 KB for this file.

The variable block size should be turned off for writing the pre-compressed data. So it is expected that the logical size for all blocks equals the recordsize.

3.3.2 Interface

A new interface is necessary to distinguish operations for pre-compressed data with usual operations.

Write interface

For every block in ZFS a logical uncompressed size and the physical compressed size has to be assigned. Hence for an operation writing a pre-compressed block the new interface would have to take an parameter containing the physical size of the record. For the logical size of a block the maximum block size set for the file is expected. But it might be useful to also pass the logical size for checking purposes. So for every logical and physical size at most one record can be written. Obviously it would also be possible to provide an array of physical sizes to write multiple records.

Because the compression algorithm will be saved in the header the used compression algorithm does not need to be passed to ZFS. It should only be ensured that data written to this interface is marked as pre-compressed in the block pointer. Though if the pre-compressed data could be made compatible to ZFS the compression algorithm has to be passed to ZFS explicitly.

Read interface

For read operations it cannot be known up front if the data is compressed or uncompressed. Although the information if the block is compressed is stored in the block pointer, and thus it can be determined if a block is compressed without actually reading the block. Also the physical size of data blocks is unknown. Hence both must be provided for every block to the Lustre frontend. Therefore it should be possible that the interface for reading files is able to read compressed and uncompressed blocks and report which block holds compressed data. The actual compression algorithm as well as the physical size in bytes is stored in the header. For read operations it would not make much sense to read a block partially because it could not be decompressed and therefore not read. So a read will cover at least one block.

ZPL Integration

In this thesis the functionality is only described and implemented in regard to Lustre using ZFS as backend. However writing pre-compressed blocks and reading the blocks compressed could also be used for other consumers of ZFS. A natural choice would be an integration into ZPL. However this is problematic because the ZPL implements the VFS layer. For the needed change of ZPL to pass the physical size or to tag data as compressed also changes to the VFS layer would have to be made.

4 Related work

This chapter explores existing transparent compression functionality. Mainly other filesystems are examined but also an approach for SSD-based compression and in HDF5 file format is introduced. Furthermore an existing design document making use of ZFS's end-to-end data integrity by Lustre is presented.

4.1 Transparent compression

The increasing performance gap of I/O and computation continuously decreases the overhead for transparent compression. Hence making it more and more promising for the use in file systems. While already some local filesystems besides ZFS are offering some implementation of transparent compression, the same is not true for parallel distributed filesystems. However, for example OrangeFS and BeeGFS [Hei14] are able to support ZFS as a backend, and thus can use its compression functionality on the server like Lustre currently can. Although in opposite to Lustre they use ZFS through the POSIX compliant interface. This approach gains flexibility because it can use any POSIX compliant filesystem as backend. On the other hand Lustre's implementation of the DMU interface makes better use of ZFS's features possible. Hence an feature like implemented in this thesis seems only feasible for Lustre without conducting changes to the VFS layer.

4.1.1 btrfs

Alongside ZFS the btrfs may be the most popular file system providing transparent compression. Btrfs currently supports ZLIB and Lempel-Ziv-Oberhumer (LZO) which are similar to gzip and LZ4, respectively. btrfs is an extent-based copy-on-write filesystem. Extent-based filesystems reserve an contiguous area of storage for a file. They do not need a pointer to every block but rather one pointer to the beginning of the extent and it's length. This way for large files metadata overhead for the block-allocation tree is saved also sequential I/O speed increases due to the data being in one consecutive area on disk [wikb]. Compression in btrfs is set on a per-extent level, and thus on a file level. This makes having uncompressed files, files compressed with ZLIB and files compressed with LZO on a single mountpoint possible. Compression in btrfs is only conducted on each page individually which is a drawback because it leads to inferior compression ratios in comparison to ZFS.

At the Lustre User Group conference 2015 Ihara and Xi presented a prototype for btrfs-OSD [SI15]. The goal is to make btrfs usable as a backend for Lustre. The initial benchmark results showed good performance. With a btrfs backend it might be possible to use it for clientside compression in Lustre like shown in this thesis for the ZFS backend.

4.1.2 NTFS

NTFS is proprietary file system only available for Windows operating systems. It compresses files using a variant of LZ77 called LZNT1. Files are compressed in chunks up to 64 Kbytes. A chunk consists of 16 (4Kbyte) clusters. Space can be saved if a compressed chunk needs less clusters to store a data. For example a chunk compressed to 58 Kbyte will save one cluster, and thus 4 Kbyte on disk. Compression can be activated von drive, directory and directory tree level.

4.1.3 APCFS

In [KKK10] Kella and Khanum present the APCFS. It is a virtual layer inserted over existing file system, compressing and decompressing data by intercepting kernel calls. This layer is implemented as a FUSE file system and compresses the files uses LZ77 and Hufmann coding in parallel. They aim for compression support for multimedia files. Each media type (e.g. images, videos, audio) is classified and a different lossless algorithm is applied. For example images should be compressed with a wavelet transform followed by quantization and different linearization schemes.

4.1.4 HDF5

The HDF5 file format used for storing large scientific data supports transparent compression with GZIP and SZIP. But also provides the option to use thrid-party compression methods. For compression the data is chunked into equally sized chunks each of which is stored separately in the file. Each chunk is compressed individually. Thus, for I/O operations only chunks containing the accessed data have to be decompressed or compressed [hdf15].

FlaZ

Many storage systems make use of SSDs as a cache due to it's superior performance over hard disk drives but have a limited capacity. To increase the latter for SSD-based caches Makatos et al. present *FlaZ* in [TM10]. *FlaZ* is an I/O system that operates at block-level and is transparent to filesystems. It provides support variable-size blocks, mapping of logical to physical blocks, block allocation, and cleanup. This way both transparent online compression and high performance can be achieved. *FlaZ* uses the Lempel-Ziff-Welch (LZW) algorithm.

4.2 End-to-End Data Integrity for Lustre

As part of the DARPA High Productivity Computing Systems program in 2009 several design documents for Lustre were prepared. Deskmukh, Dilger and Dawson proposed a way to combine Lustre's data checksumming with the ZFS on-disk data checksumming [RD09]. By doing this Lustre would profit from the end-to-end data integrity of ZFS. The Lustre client computes the data checksum on write and sends it to the server with the RPC. The server verifies the data checksum once and re-use the same checksum for the disk blocks in ZFS. For reads the ZFS backend is able to skip the checksum verification completely and let the Lustre client verify the checksum. In case of a bad checksum a re-read could be issued which verifies the checksum with the end-to-end data integrity of ZFS.

5 Implementation

In this chapter the implementation of the described functionality from chapter 3 is discussed. First the ARC and dbuf datastructures are described. Followed by a detailed presentation of the I/O paths of a write and a read operation in ZFS. On this basis the necessary changes that have to be made are shown. Beginning with the metadata for the pre-compressed data. Afterwards the implementation for writing and reading pre-compressed blocks is presented.

5.1 Datastructures

5.1.1 ARC buffer

When a block is cached in the ARC it is tracked by an ARC header `arc_buf_hdr_t`. The ARC header consists of fields common to L1- and L2ARC, the header for a buffer in the L2ARC and the header for a buffer cached in L1ARC. Common fields are for example the DMU that the header is tracking and the logical size as well as physical size. For recently evicted blocks these common fields can provide information about the block. The sub-structure `l2arc_buf_hdr_t` holds information to retrieve the block from the L2ARC device.

For blocks in the L1ARC the header contains a structure called `l1arc_buf_hdr_t` which points to the data block in memory. When an ARC is cached in L1ARC the fields for the L2ARC are undefined. The L1ARC has a private data block stored in a `b_pabd`. This data block is a copy of the on-disk physical block. When a data block is compressed on-disk the `b_pabd` is also compressed. In addition to that the data is also cached in a list of ARC buffers `arc_buf_t`. So a header is able to hold multiple buffers. The `arc_buf_t` is the structure that is actually accessed by consumers of the ARC. Every consumer references its own buffer in the list. The consumer either requests the buffer compressed or uncompressed. Currently compressed buffers are only requested by `zfs send`. This allows the buffer for sharing the data with the `b_pabd`. Otherwise the buffer will hold an uncompressed copy of the data [arc].

Figure 5.1 shows an example of a block that is cached in the ARC and the relation between the header and buffers. In this case the ARC is cached in L1ARC. Hence the L2ARC header is undefined. The header `b_pabd` is referencing a copy of the compressed data which is currently stored on disk. The ARC buffer referenced by the header is requested with compressed data, and thus able to share the data with the header. The

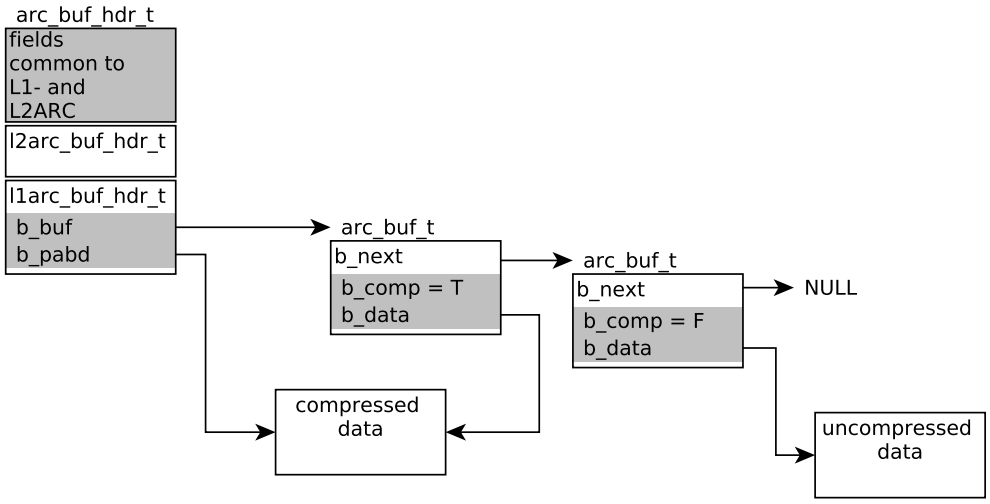

Figure 5.1: Example of a ARC header referencing two ARC buffers

second buffer in the list was requested with uncompressed data and holds its own uncompressed copy.

Writing to the ARC

First before writing to disk the ARC header will drop the data stored in `b_pabd` due to the fact that the physical data is about to be rewritten. Either loaned ARC buffer or a released ARC buffer are filled with the updated data that should be written to disk. Loaned ARC buffers are newly created, and thus cannot have any other readers. For a released buffer a new header is created if there were other readers of the buffer. This ensures that only one buffer and one header have to updated conducting a write operation. After the data is written to disk the `b_pabd` is filled with the data block that was just written.

Reading from the ARC

Consumers can only read blocks cached in the `l1arc_buf_hdr_t`. So first it has to be checked if the ARC header tracking the block exists. If the header either does not exist or the L1ARC was evicted, the data has to read from disk and a `l1arc_buf_hdr_t` has to be created. For the case of data cached in L2ARC the data cannot be accessed directly. When retrieved from the L2ARC the data will be integrated into the L1ARC and read from there. If it exists a new ARC buffer will be created. Three options for the data the new buffer will reference are possible.

1. If a uncompressed buffer already exists the data of the existing are copied and referenced by the new buffer.

2. The data of `b_pabd` is decompressed and copied into a new buffer.

3. The data of `b_pabd` can be shared and is referenced by the new buffer. Sharing uncompressed data is only possible if the buffer is the last buffer in the list. However compressed buffers can be anywhere in the header's list.

5.1.2 Dbuf

The buffer in the DMU layer is a structure called `dmu_buf_impl_t` which is referred to as dbuf. Every dbuf represents one block either direct or indirect blocks on disk. It inherits a publicly visible structure called `dmu_buf_t` shown in listing 5.1. `db_data` points to the data buffer of an ARC buffer.

The `dmu_buf_impl_t` holds, among others, additional information about the indirection level and the block id. So for example a dbuf with level 0 and block id 0 describes the first block in the file, level 0 and block id 1 describes the second block in the file.

```
1 typedef struct dmu_buf {
2     uint64_t db_object;      /* object that this buffer is
        ↪ part of */
3     uint64_t db_offset;      /* byte offset in this object */
4     uint64_t db_size;        /* size of buffer in bytes */
5     void *db_data;           /* data in buffer */
6 } dmu_buf_t;
```

Listing 5.1: dmu_buf struct

States

The state of a dbuf is also stored in `dmu_buf_impl_t`. Figure 5.2 shows a simplified transition diagram for dbufs. First when a dbuf is allocated it is uncached. Then a dbuf can either be read, filled or not filled. `NOFILL` is used for blocks embedded in a dnode or preallocated blocks. `FILL` is set when a full block is written. In this case the dbuf is not first read in but just filled with the data. When reading a block the state changes to `READ`.

After the dbuf is filled with either through a `FILL` or a `READ` the dbuf is added to the dbuf cache. When the dbuf is evicted from the cache it first enters the `EVICTING` state until it is freed.

Retrieving and creating a dbuf

For retrieving a dbuf from the cache or for creating a dbuf the function `dbuf_hold` is used. On the basis of a provided dnode, block id and level a block on disk can be uniquely identified. Either the dbuf for that block already exists in the cache and is retrieved or the dbuf is created. Also the hold count for that dbuf is incremented. A hold count indicates that the dbuf is currently accessed by some caller.

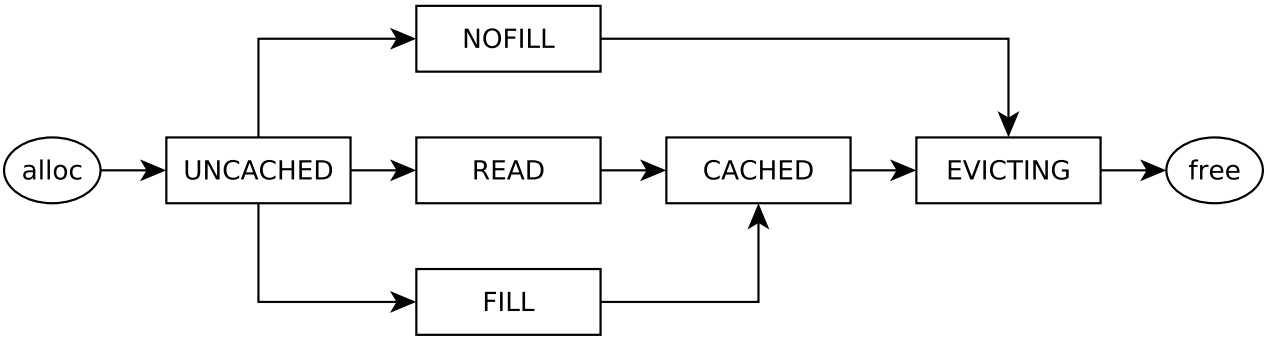

Figure 5.2: Transition diagram for the states of a dbuf [dbu].

5.2 Write I/O path

Lustre's OSD layer implements the DMU interface of ZFS. In the following it is described how Lustre makes use of the DMU interface for reading and writing. The `osd_object` structure has a reference to a dbuf from the DMU. The structure `niobuf_local` shown in Listing 5.2 is used to get or write the data to the DMU layer. It holds an array of (4 Kbyte) pages which hold the data.

```
1 struct niobuf_local {
2     __u64       lnb_file_offset;
3     __u32       lnb_page_offset;
4     __u32       lnb_len;
5     __u32       lnb_flags;
6     int     lnb_rc;
7     struct page *lnb_page;
8     void        *lnb_data;
9 };
```

Listing 5.2: `niobuf_local` structure

5.2.1 OSD layer

The function `osd_bufs_get_write` fills in the pages in `niobuf_local`. Here generally two cases have to be considered, a partial block write and a full block write.

Full block write

If a write covers a full block Lustre is able to make use of the zero-copy approach. Zero-copy has the advantage that the buffer allocated in Lustre does not have to be copied into a buffer in ZFS. The function `dmu_request_arcbuf` "borrows" an ARC buffer. This is done before a transaction is entered. This avoids the possibility of holding up the transaction if the data copy hangs up on a pagefault (e.g., from an Network File System (NFS) server mapping). The `lnb_data` pointer is set to reference this ARC buffer. The

data is now copied into the ARC buffer and a transaction is entered and the data can be written to the DMU.

In `osd_write_commit` first calls `osd_grow_blocksize`. As explained in Chapter 2 for small files with only one block ZFS is able to adjust the blocksize dynamically. If the write operation will increase the block size `osd_grow_blocksize` increases the block accordingly. For files with more than one block all blocks already have the recordsize, and thus `osd_grow_blocksize` will not do anything. Afterwards a for loop iterates over the data and calls `dmu_assign_arcbuf` for every full block that should be written.

Partial block write

For partial block writes the path is mostly the same. However `osd_bufs_get_write` does not request an ARC buffer. Instead it fills in the data for as much pages as needed in `niobuf_local`. The zero-copy approach only makes one call for the whole data block. Partial block writes result in calls of `dmu_write` for every page individually.

5.2.2 ZFS write I/O path

Updates within a ZFS pool are first accumulated in the ARC cache and later written to disk via a sync operation. Usually this is triggered until a specified time has passed or 64 Mbyte of dirty data have been accumulated. The sync operation can also be started through system calls like a fsync or administrative action like an export of a pool. But before the data in memory is flushed to disk ZFS has to create a checkpoint to ensure the filesystem always transitions from a consistent state to another consistent state. Therefore all writes and updates are assigned to a specific Transaction Group (TXG). Synchronizing all data in a TXG may take several seconds. To be able to modify data while synchronizing new write operations are tagged with a new TXG. When a block is modified in this new TXG that was also modified in the currently syncing TXG the block has to be copied in memory and the modification have to be conducted on this copy. As a result of this proceeding writing to disk in ZFS is done in two phases referred to as *open context* and *syncing context*.

Open context

In the open context the data from the user's buffer is copied into ZFS. Figure shows 5.3 a simplified program flow for the open context [Mac14].

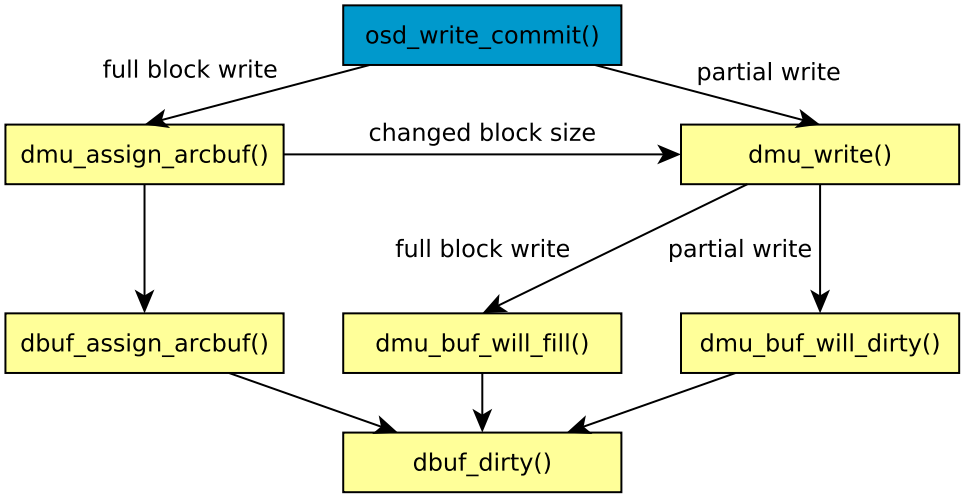

Figure 5.3: Write operation in open context.

Full block write

As explained for the OSD layer for full block writes the zero-copy approach is possible. For this case `dmu_assign_arcbuf` is called. The ARC buffer created by `dmu_request_arcbuf` and then filled with the data is provided to `dmu_assign_arcbuf`.

In `dmu_assign_arcbuf` first the dbuf associated with the requested ARC buffer is retrieved. For example if the third block of a file should be written, the dbuf with level 0 and block id 2 is retrieved.

If the offset is not aligned and the ARC buffer is not the same size as the dbuf the function falls back to `dmu_write`. Otherwise `dbuf_assign_arcbuf` is called. This function assigns the ARC buffer to the dbuf so that the dbuf points to the data of the ARC buffer. If the dbuf already pointed to an ARC buffer this buffer is destroyed. At the end the data is marked dirty for this transaction.

Full block writes are also possible with `dmu_write`. With the Lustre interface this only happens when the function `osd_grow_blocksize` grows the block size and `dmu_assign_arcbuf` falls back to `dmu_write`. Also this path is mostly analogous for a partial block write. The only difference is that dbuf first does not have to be read in.

Partial block write

For partial block writes `dmu_write` is called. `dmu_write` first calls `dmu_buf_hold_array`. This function basically fetches all dbufs that will be affected by the write operation and returns buffers for them in an array. So `dmu_write` is able to write to more than one block.

Next iterating over all dbufs for full block writes `dmu_buf_will_fill` is called or for

partial writes `dmu_buf_will_dirty` is called. A partial write requires a read-modify-write operation so the dbuf first is needed to be read in. `dmu_buf_will_fill` allocates a new ARC buffer associated with the dbuf. Now both functions call `dbuf_dirty` to mark the dbuf dirty in this transaction. `dbuf_dirty` adds the dbuf and the dnode associated with it to a list of dirty records. These dirty records then later are written to disk in syncing context. After marking the dbuf as dirty it is save to actually modify them with the new data. So after that the data passed to `dmu_write` is memcopied into `db_data` which points to the ARC buffer allocated in `dmu_buf_will_fill` or `dmu_buf_will_dirty`.

Dirtying dbufs

The main use of `dbuf_dirty` is to keep track of multiple versions of this dbuf's dirty data. Figure 5.4 shows how the dbufs are dirtied. When a dbuf is first dirtied in a transaction a `dbuf_dirty_record_t` is created which points to the ARC buffer holding the data. If the ARC buffer's data is modified in the same transaction it will just be overwritten. If the dbuf is dirty for an old transaction the ARC buffer is copied, a new `dbuf_dirty_record_t` created and the dbuf assigned to the copied ARC buffer. This ARC buffer then safely can be modified.

At most three transaction can be active at the same time, one in open context, one in syncing context and one quiescing. A quiescing transaction is used as an additional buffer for heavy workloads.

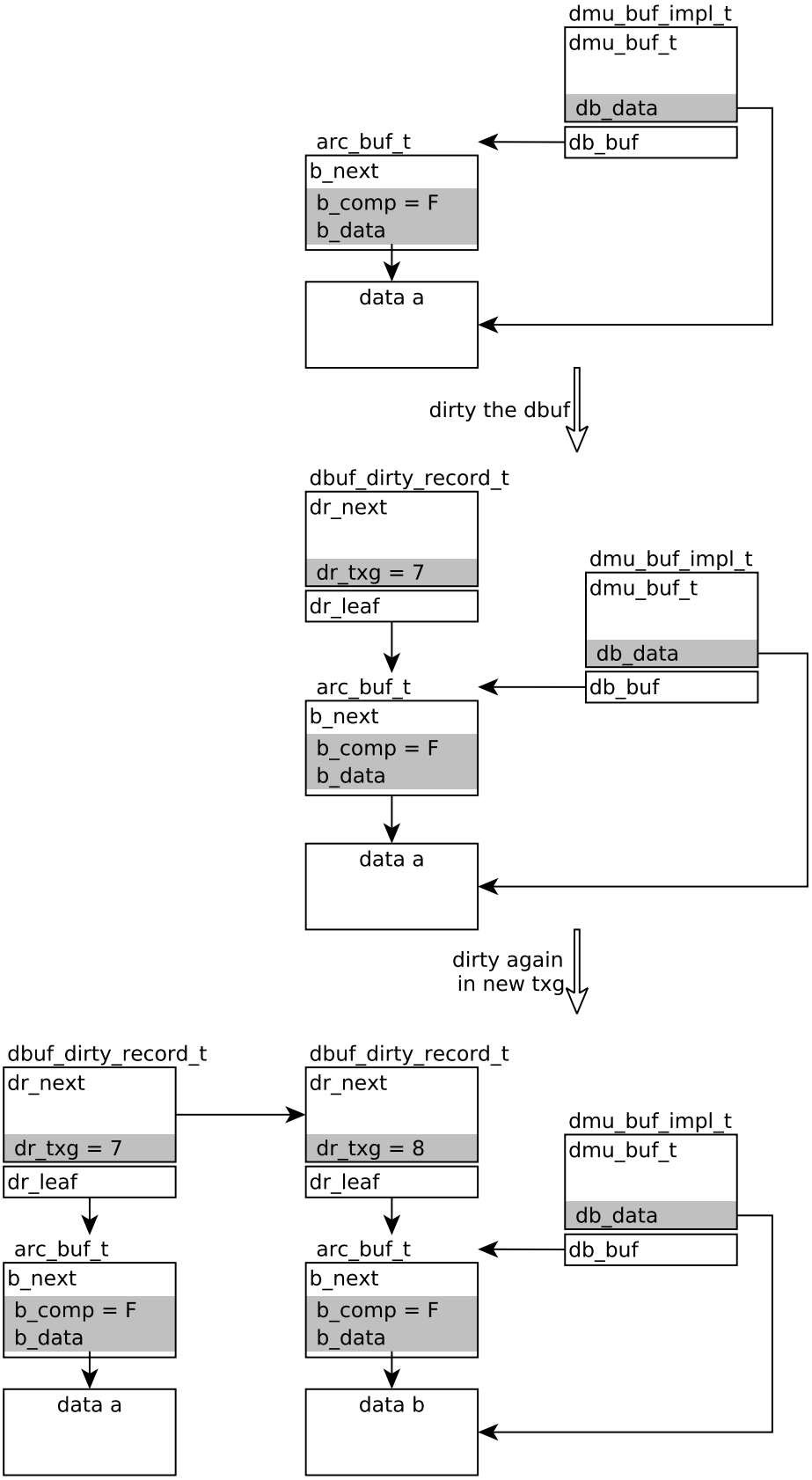

Figure 5.4: Dirtying a dbuf.

Syncing context

In syncing context all dirty data blocks of one TXG is written to disk. Since existing data is never overwritten, modified blocks are written to a new location on disk. If a data block is updated the block pointer in an indirect block also has to be updated. For indirect block goes the same as for data blocks – they are never overwritten. Hence indirect blocks have to be written to a new location. This on the other hand requires that the block in a higher level referencing the just updated block also needs to be modified. This way ZFS walks up the file tree until the dnode is reached and updated. After all files are updated the synchronization operation walks up the ZFS data structure tree until the uberblock is updated and the syncing is done.

The syncing of a file is triggered by the function `dnode_sync`. Subsequently `dbuf_sync_list` will go through the list of all dirty records and either call `dbuf_sync_indirect` for indirect and `dbuf_sync_leaf` for data blocks. Then the function `dbuf_write` will issue an I/O to commit the provided dirty buffer to disk. Here `dmu_write_policy` is called which fills in the `zio_prop_t` structure. It keeps track of properties concerning write operations of a block. These properties are for example the type of the object that is written, the used checksum algorithm and the compression algorithm. Usually the compression algorithm will be set to the algorithm set for the object set (e.g. filesystem). But also an option for overriding the compression algorithm is provided. This is used when a compressed ARC buffer will be written to disk. In this case the same algorithm used for compressing the ARC buffer is adopted.

After the properties are set `arc_write` is called. Mainly this function will just pass the data from the ARC buffer to `zio_write`. Also callbacks are created that will store the physical on-disk compressed or uncompressed data into the header's data buffer after the data was written to disk by the ZIO. As explained after the write to disk the header will be written with the physical on-disk data.

```
1
2     zio = zio_write(pio, spa, txg, bp,
3         abd_get_from_buf(buf->b_data, HDR_GET_LSIZE(hdr)),
4         HDR_GET_LSIZE(hdr), arc_buf_size(buf), zp,
5         arc_write_ready,
6         (children_ready != NULL) ? arc_write_children_ready : NULL,
7         arc_write_physdone, arc_write_done, callback,
8         priority, zio_flags, zb);
```

Listing 5.3: `zio_write` call in `arc_write`

Figure ?? shows how `arc_write` calls `zio_write`. Among others the block pointer `bp` that is written to, the transaction `txg` or pointer to the data `abd_get_from_buf(buf->b_data, HDR_GET_LSIZE(hdr))` are passed. Important variables for compression are highlighted in red. `HDR_GET_LSIZE(hdr)` and `arc_buf_size(buf)` define the logical size and the physical size that is written to the block pointer. `arc_buf_size(buf)` returns the size of the ARC buffer. If the ARC buffer is compressed it returns the

rounded up physical size of the header. zp holds a zio_prop_t where the compression algorithm is set. In zio_flags flags with information about the zio are passed. The ZIO_FLAG_RAW flag marks an I/O operation as "raw" and therefore is neither compressed or decompressed. When writing a compressed ARC buffer the ZIO_FLAG_RAW is always set.

Having the data passed to zio_write this function will fill the zio_t structure by calling zio_create. The structure holds all relevant data to carry out the zio operation. This are for example the physical size, logical size and the stages of the pipeline.
The ZIO works the way that it creates all operations and structures them in a tree. In doing so thousands of zios can be accumulated under a zio_root. A call of zio_wait will enqueue these zios into the pipeline and executes them.

5.3 Read I/O path

5.3.1 OSD layer

In case of a read Lustre calls the function osd_bufs_get_read. For a read Lustre always uses a zero-copy approach. The function dmu_buf_hold_array_by_bonus retrieves all dbufs directly that hold the wanted data directly from the DMU. Only now in the subsequent for-loop which iterates over the dbufs the buffers are memcopied into Lustre. For each dbuf the data is divided into pages and copied into pages of a niobuf_local structure.

5.3.2 ZFS read I/O path

Figure 5.5 shows a simplified I/O path for a read operation in ZFS.

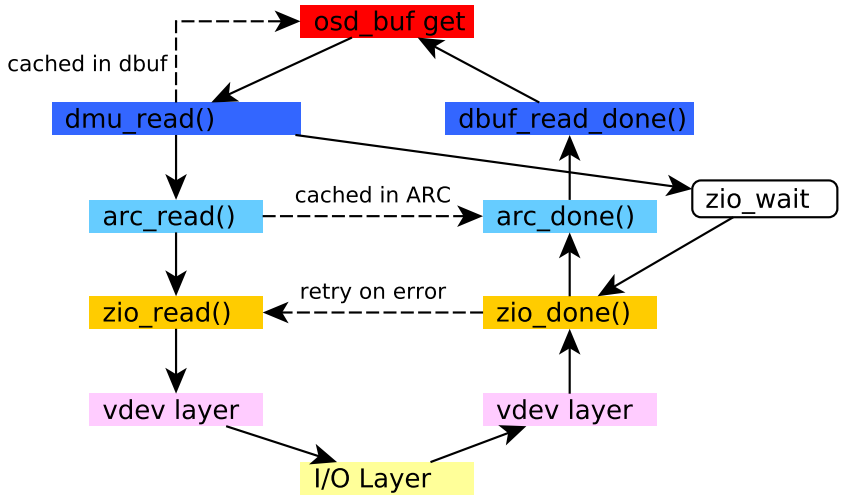

Figure 5.5: Reading a file.

Just like in the write path `dmu_buf_hold_array_by_dnode` will be called which should identify all dbufs that need to be read and return them in an array. But instead of just returning them they are filled with the data of that block. Therefore for every dbuf `dbuf_read` is called. So when a 1 Mbyte should be read in with a block size of 128 Kbyte an array of 8 dbufs will be returned to Lustre [Ahr16].

In `dbuf_read` the dbuf is either in `CACHED`, `UNCACHED` state or the dbuf is currently written or read by another caller. If a dbuf is cached it is already filled and nothing has to be done. Although one special case exists for the compressed `zfs receive`. In this case the data is compressed in the buffer and has to be decompressed to read it.
When another caller currently writes or reads the dbuf will be cached afterwards. So in this case it only has to be waited for the dbuf to change into the cached state. If the dbuf is uncached `dbuf_read_impl` is called which will initiate the read by calling `arc_read`.

`arc_read` first checks in the hash table if the ARC header is in cache for the specified block. If the block is available in L1ARC the data can be returned. Otherwise the data has to be read from disk. In this case a `zio` has to be performed. `zio_read` creates a `zio` and fills in all parameters for reading the block.

After every `dbuf_read` returned to `dmu_buf_hold_array_by_dnode` the prefetcher is notified. Following the function `zio_wait` is called. This function executes all I/Os created for the read operation and waits for them to complete. If all I/Os are done and every dbuf is filled with it's data the data can be returned. The block is not in the cache and the data has to be retrieved from disk via a `zio_read`. If the user data is compressed `ZIO_FLAG_RAW` is set. This triggers a "raw" read and the data is not decompressed in `zio_done`. Hence the data is stored compressed in the buffer of the ARC header (`b_pabd`).
In `arc_read_done` `arc_buf_alloc_impl` is called. This function will allocate an ARC buffer and fill it with data. If the header points to compressed data the data might be uncompressed in another ARC buffer. Whenever this is the case the uncompressed version is copied into the allocated buffer, otherwise the data has to be decompressed. Finally the `dbuf_read_done` sets the dbuf to `CACHED`, if no error occurred.

5.4 Metadata for pre-compressed data

5.4.1 Header

As described in the design all pre-compressed blocks should be stored on-disk with a header containing the used compression algorithm and the physical size of the block in bytes. The desirable approach would be to write and read the header transparent in ZFS. For writing Lustre would pass the compressed data record together with parameters for psize and algorithm to ZFS. Here first ZFS would write both parameters to a buffer.

Then the compressed data record is memcopied with a offset providing enough space for the header into the same buffer. For reading a compressed data record ZFS would read in the header and return both parameters back to Lustre. The data record would be memcopied without the header to the Lustre buffer.

However Lustre is using a zero-copy approach for reading the data and also for writing a zero-copy approach is possible. Zero-copy means in this case that Lustre and ZFS share the same buffer. Thus, the buffer is not memcopied. Therefore the better approach is to let Lustre handle the header.

5.4.2 Alignment shift

The alignment shift referred to as *ashift* is a property of a pool that is set at creation time and is immutable [ash]. 2^{ashift} defines the smallest possible I/O on a vdev meaning that I/O operations will be made aligned to this size. Valid values are from 9 to 13 (512 byte to 8 Kbyte). For optimal performance the value should be equal or greater than the sector size of the child vdevs. Per default it is set to a special value 0 that auto-detects the best ashift by calculating the maximum physical sector size of all child vdevs. But that may change if new vdevs are added to the pool using are bigger physical size sector. So at pool creation it should be taken into account if new vdevs will be added in the future.

The physical size of any block in ZFS is always a multiple of the smallest ashift of a vdev in a pool. Thus, when setting the physical size in ZFS it has to be rounded up to the next 2^{ashift}. So for example when the ashift is set to 12 (4096 Kbyte) and a block with a physical size of 5000 Kbyte should be written to disk the psize stored in the block pointer is set to 8192Kbyte.

5.4.3 Block size

For pre-compressed blocks the logical size should always equal the recordsize. Usually a new file is created the block size is set to the page size in `osd_mkreg`. Here the block size should be set directly to the recordsize. This way `osd_grow_blocksize` becomes obsolete because the maximum size for a block is already set. This also ensures that the zero-copy approach does not fall back to `dmu_write` because the block size always stays the same.

5.4.4 Flag for pre-compressed blocks

On-disk the compression algorithm of every record is stored in the block pointer. It is persisted as a number of one byte length. ZFS uses the enum `zio_compress` to map the algorithms to the numbers stored in the pointer. As explained a new number is needed to identify that the data is pre-compressed by Lustre. So an additional constant called `ZIO_COMPRESS_EXTERNAL` is added. Leading to an updated enum shown in Listing **??**.

```
1 enum zio_compress {
2     ZIO_COMPRESS_INHERIT = 0,
```

```
3       ZIO_COMPRESS_ON,
4       ZIO_COMPRESS_OFF,
5       ZIO_COMPRESS_LZJB,
6       ZIO_COMPRESS_EMPTY,
7       ZIO_COMPRESS_GZIP_1,
8       ZIO_COMPRESS_GZIP_2,
9       ZIO_COMPRESS_GZIP_3,
10      ZIO_COMPRESS_GZIP_4,
11      ZIO_COMPRESS_GZIP_5,
12      ZIO_COMPRESS_GZIP_6,
13      ZIO_COMPRESS_GZIP_7,
14      ZIO_COMPRESS_GZIP_8,
15      ZIO_COMPRESS_GZIP_9,
16      ZIO_COMPRESS_ZLE,
17      ZIO_COMPRESS_LZ4,
18      ZIO_COMPRESS_EXTERNAL,
19      ZIO_COMPRESS_FUNCTIONS
20  };
```

Listing 5.4: `enum zio_compress`

`ZIO_COMPRESS_FUNCTIONS` is used for checking purposes and determine the length of the `zio_compress_table` which maps each specifier to a level and function for compression and decompression. So for example `ZIO_COMPRESS_LZ4` is mapped to the functions `lz4_compress_zfs` and `lz4_decompress_zfs`. For `ZIO_COMPRESS_FUNCTIONS` the compression and decompression function is mapped to `NULL`.

```
1   zio_compress_info_t zio_compress_table[ZIO_COMPRESS_FUNCTIONS] = {
2       {"inherit",     0,  NULL,       NULL},
3       {"on",          0,  NULL,       NULL},
4       {"uncompressed",    0,  NULL,       NULL},
5       {"lzjb",        0,  lzjb_compress,  lzjb_decompress},
6       {"empty",       0,  NULL,       NULL},
7       {"gzip-1",      1,  gzip_compress,  gzip_decompress},
8       {"gzip-2",      2,  gzip_compress,  gzip_decompress},
9       {"gzip-3",      3,  gzip_compress,  gzip_decompress},
10      {"gzip-4",      4,  gzip_compress,  gzip_decompress},
11      {"gzip-5",      5,  gzip_compress,  gzip_decompress},
12      {"gzip-6",      6,  gzip_compress,  gzip_decompress},
13      {"gzip-7",      7,  gzip_compress,  gzip_decompress},
14      {"gzip-8",      8,  gzip_compress,  gzip_decompress},
15      {"gzip-9",      9,  gzip_compress,  gzip_decompress},
16      {"zle",         64, zle_compress,   zle_decompress},
17      {"lz4",         0,  lz4_compress_zfs, lz4_decompress_zfs},
18      {"external",        0,  NULL,       NULL}
19  };
```

Listing 5.5: Mapping of each specifier to a level and algorithms.

5.4.5 Flag for compressed dbufs

To identify that a dbuf currently holds compressed data the publicly visible structure of the dbuf is extended by a boolean called `db_compressed`. Also the extra parameter `boolean_t compressed` passed to `dbuf_create` is added. While writing pre-compressed data this parameter is set to `TRUE` and passed through to `dbuf_create`. This will ultimately set `db_compressed` to `TRUE`. For all other cases while writing `db_compressed` is set to `FALSE`. When a pre-compressed block is read and the dbuf exists the block is already marked as compressed. In the current implementation the externally compressed data cannot be decompressed. So a compressed dbuf is not able to be read decompressed. Thus, an error has to be returned for this operation. If a pre-compressed block is read for which a dbuf does not exist the information can be retrieved from the block pointer. So if the block pointer references a pre-compressed block `ZIO_COMPRESS_EXTERNAL` is set as compression algorithm. If `ZIO_COMPRESS_EXTERNAL` is set `db_compressed` is set to `TRUE`.

```
1 typedef struct dmu_buf {
2     uint64_t db_object;
3     uint64_t db_offset;
4     uint64_t db_size;/
5     boolean_t db_compressed;
6     void *db_data;
7 } dmu_buf_t;
```

Listing 5.6: The extended `dmu_buf` struct

5.5 Writing pre-compressed blocks

The goal is to write a pre-compressed block and store the logical size and the physical size passed by Lustre. Additionally these blocks have to tagged as externally compressed with `ZIO_COMPRESS_EXTERNAL`. Ultimately this information is decided by what `arc_write` passes to `zio_write` as shown in figure **??**. The logical size and physical size is retrieved from the header of the ARC buffer that is written. Additionally the compression algorithm is overriden in `dmu_write_policy` by what compression algorithm is set in the same ARC header.

So the idea is to allocate an compressed ARC buffer with the physical size, logical size and compression algorithm and fill this buffer with the pre-compressed data block. In the I/O path for writing it was shown that two possible ways exist to write blocks through the DMU, either with `dmu_write` or `dmu_assign_arcbuf`. Writing through `dmu_assign_arcbuf` expects to write a full block. When writing a pre-compressed block this block obviously never is a full block. But the block logically represents a full block. So it is known that there can be no other concurrent writes to this block. This makes it possible to also use the zero-copy approach. Both could be used to write pre-compressed blocks.

However, the zero-copy approach fits the purpose of writing the pre-compressed blocks better. As shown in the design for the read-modify-write case the whole block is rewritten. The zero-copy approach does this per default as a new ARC buffer is allocated and filled by Lustre. Subsequently this ARC buffer overwrites the block without reading it first.

`dmu_write` on the other hand is called for every (4 Kbyte) page. If for example a block with a physical size of 12 Kbyte three calls of `dmu_write` would be made. This leads to partial block writes that read the old data from disk first. A lot more changes would have to made for realizing the implementation with this approach. If the block is rewritten in the same transaction with another physical size a new compressed ARC buffer with the new physical size has to be created and the old destroyed. Which would also hold up the write as no new page can be written as long as the ARC is allocated. If the block is rewritten when the block is not cached, the block would be first read from disk. This behaviour ideally also would have to be changed. Moreover for writing with `dmu_write` additional checks whether the block is aligned and only updates one block would have to be made.

Consequently this lead to the decision to use and implement the zero-copy approach with `dmu_assign_arcbuf`. Furthermore the zero-copy approach saves memcopying the data into ZFS, and thus might provide better performance anyway. In the following the implementation for writing the pre-compressed blocks is described.

Zero-copy approach

The idea is to instead of allocating an ARC buffer with `dmu_request_arcbuf` an compressed buffer is allocated using the new function `dmu_request_compressed_arcbuf`. The physical size is handed to the function. It is assumed that the logical size is equal to the recordsize of the current file. The physical size is provided in bytes and thus must be rounded up to the smallest ashift of the device to be stored on disk. The function calls to loan out an compressed ARC buffer with the provided logical size, the rounded up physical size and the compression set to `ZIO_COMPRESS_EXTERNAL`. Loaned out ARC buffers are always anonymous. Thus, not in the cache. Allocating an ARC buffer will first lead to allocating a header. In this header the parameters psize, lsize, and compression are stored. Afterwards for this header the actual buffer is allocated. Since the header only is associated with the newly allocated buffer and both are set to hold compressed data, it is possible to share the data buffer.

After allocating the data buffer is still empty. This buffer will be filled by Lustre. Leaving a compressed ARC buffer like shown in 5.6. In this example the logical blocksize is 128 Kbyte and the physical 80 Kbyte. In the header the compression algorithm is set to `ZIO_COMPRESS_EXTERNAL`. In the ARC buffer also compression is set to `TRUE`.

But to write this buffer to disk in the syncing context, the buffer has to be associated with a dirty dbuf. This is done in a function `dmu_assign_compressed_arcbuf` that is introduced and will replace `dmu_assign_arcbuf`. Because of the padding of the physical

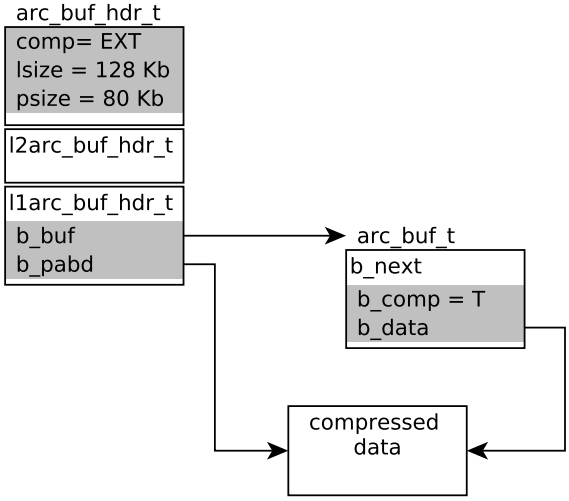

Figure 5.6: The compressed ARC buffer after requesting.

size the remainder between the rounded up psize and the exact psize is zeroed out explicitly. Otherwise the function is able to work mainly analogous to it's counterpart for uncompressed ARC buffers. The dbuf that is retrieved by `dbuf_hold` has to be tagged as compressed. Following that `dbuf_assign_arcbuf` assigns the retrieved dbuf to the ARC buffer. If the dbuf is cached it is already assigned to another ARC buffer this buffer has to be destroyed. If the old buffer created in the same transaction and already tagged dirty, this buffer should not be written to disk. Thus, also the dirty dbuf is overwritten with the new ARC buffer. Then finally the allocated ARC buffer can be assigned to the dbuf and dirtied.

This way after `dbuf_assign_arcbuf` there will be a new allocated ARC buffer containing the compressed data. This ARC buffer has the correct physical size passed to `dmu_assign_compressed_arcbuf`. In syncing context `arc_write` passes the logical and physical size of the buffer to `zio_write`. Furthermore for every writing a compressed ARC buffer `ZIO_FLAG_RAW` is set which will ensure the data is not compressed again before written to disk.

Synchronous write

The write operations called by Lustre are asynchronous but in ZFS these writes might still be executed synchronous. For example when the pool is exported while writing data to disk. In this case `dmu_sync` is called. This function is provided with a single dbuf that should be written. Analogous to a asynchronous write it has to be made sure that the compression algorithm is set to `ZIO_COMRPESS_EXTERNAL` and `ZIO_FLAG_RAW` is set. With `db_compressed` it is possible to check whether the dbuf contains pre-compressed data. If so `dmu_write_policy` is called with `ZIO_COMRPESS_EXTERNAL` to override the inherited algorithm. And the "raw" flag is passed to the subsequent to a call of `arc_write`.

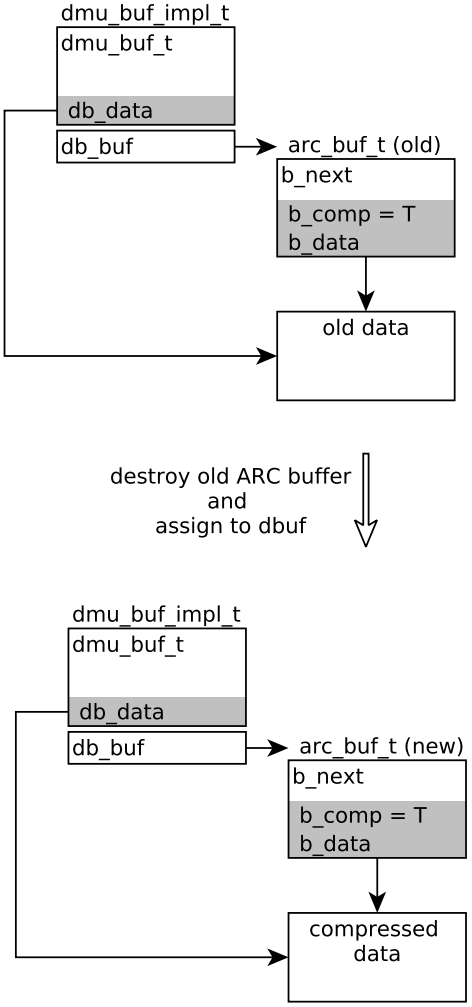

Figure 5.7: Assign allocated ARC buffer to the dbuf.

5.6 Reading pre-compressed blocks

A read operation of a pre-compressed block should return the compressed block as well as providing information about the physical size of the block and the compression algorithm. As the physical size and the compression algorithm is stored in the header the main goal is to suppress the decompression and return information back to Lustre for blocks that are compressed.

`dmu_buf_hold_array_by_bonus` retrieves all dbufs filled with its data and returns them back to Lustre. For reading the data compressed this function is replaced with `dmu_buf_hold_array_by_bonus_compressed`. This function works analogous to the original function except that `dmu_buf_hold_array_by_dnode` is called with a flag indicating that the dbufs should be returned in a compressed state.

It has to be made sure when reading a record marked as `ZIO_COMPRESS_EXTERNAL` is not decompressed. Here `arc_read` has to be called with a `ZIO_FLAG_RAW` flag. This triggers a compressed read.

`ZIO_FLAG_RAW` sets `acb_compressed` to true which leads to filling the buffer with compressed data in `arc_buf_alloc_impl`.

Notify the prefechter

The problem here is that the prefetcher will only read the data into the ARC buffer without creating a dbuf, and thus does not set `db_compressed` and ultimately `ZIO_FLAG_RAW`.

When reading ZFS will try to prefetch data. Either indirect blocks or data blocks can be prefetched. Figure [**?**] shows the call sequence for prefetching a data block. In `dmu_buf_hold_array_by_dnode` the function `dmu_zfetch` is called. There a for-loop iterates over a calculated number of blocks and calls `dbuf_prefetch` for every block. Subsequently for data blocks the function `dbuf_issue_final_prefetch` is called. This function calls `arc_read` for the specified block. To ensure the ARC buffer is not decompressed the block a boolean is passed down the call sequence that, if true, sets `ZIO_FLAG_RAW`.

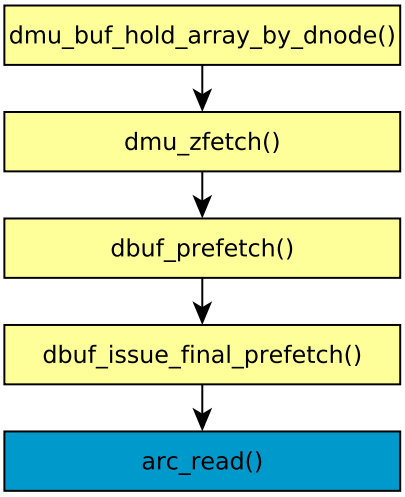

Figure 5.8: Call sequence for prefetching a data block.

6 Evaluation

This chapter presents performance tests that were conducted and subsequently the results are evaluated. First the infrastructure on which the tests are made is described. Afterwards the preparatory work that had to be made to realize the tests is shown. Finally the results are evaluated.

6.1 Infrastructure

The tests are done on a *nehalem* node at the cluster of the Scientific Computing group at the DKRZ [neh]. The node has 8 *Intel Xeon X5560* CPUs with 2.80 GHz and 12 GB RAM of main memory. The operating system on the node is *Ubuntu 16.04.2 LTS*. A 500GB partition of a 2TB HDD is used for the ZFS filesystem. All tests were repeated five times and the results averaged.

6.2 Measurements

6.2.1 Test data

The goal of this evaluation is to test the read performance of the implemented feature. For testing the default recordsize of 128 KB is used. 100000 blocks are written for each file which are filled 25%, 50%, 75% and 100%. So four files are tested which have 3.28 GB, 6.55 GB, 9.83 GB and 13.11 GB of data. Figure 6.1 illustrates both cases for the tested data. These cases should represent the two approaches shown in Figure 3.1 for storing the compressed data. The data written without gaps is tested with the default version of ZFS. The files with gaps in between the blocks for aligned access are tested with the modified version. First the ARC is emptied with `echo 3 > /proc/sys/vm/drop_caches` so that it is ensured the data is read from disk.

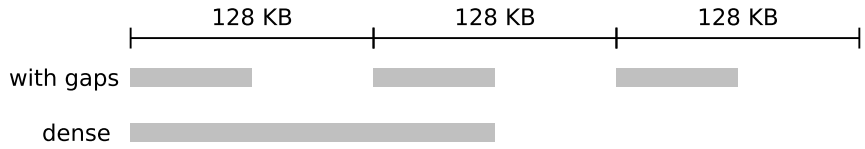

Figure 6.1: The two test cases. The data id written with or without gaps.

6.2.2 Preparatory work

Because the integration of the implemented feature in Lustre was not finished during the work on this thesis a workaround using a modified ZPL layer is used to write and read the data. This was also used for basic functionality tests. The ZPL layer is modified in such a way that it simulates the behaviour of Lustre's OSD layer for the pre-compressed data. The data is written to und read from ZFS like Lustre would do it for the pre-compressed blocks. Therefore it is possible to use ZFS's infrastructure for the data in the intended way. The results should work as a first indication for the performance of the implemented feature.

Writing transformed blocks

`zfs_write` is modified in a way that writes are forced through `dmu_assign_compressed_arcbuf`. First the block size is set to the recordsize (128 KB) with `zfs_grow_blocksize`. This equals the disabled variable block size. Afterwards with `dmu_request_compressed_arcbuf` a compressed ARC buffer is allocated and subsequently written to the DMU with `dmu_assign_compressed_arcbuf`. Just like the calls Lustre should make in the OSD layer when writing pre-compressed blocks.

The files are written with a simple C program. This program writes a given amount of blocks of a block size with random data. The blocks written should resemble the pre-compressed blocks written by Lustre. After a block is written a seek to the next multiple of the recordsize is executed. If for example three blocks with a size of 50 KB are written, the first block is written to the file from offset 0 KB to 50KB, the second block from 128 KB to 178KB and the third from 256KB to 306KB.

Reading transformed blocks

The function `zfs_read` iterates over the data and calls `dmu_read_uio_dbuf` until the whole data is read. For reading files `dmu_read_uio_dbuf` has to be modified. This function is the equivalent to Lustre's `osd_bufs_get_read`. It copies the dbuf's data into a structure called `uio`.

Usually `dmu_read_uio_dbuf` would return uncompressed blocks. If the user wants to read 100KB, 100KB of data are returned. But in this case the "compressed" block is returned, and thus less bytes. So if the user wants to read a 128 KB block, the function returns the compressed block that for example only has 50 KB. This is the same behaviour wanted when reading compressed data to Lustre.

But if just 50 KB instead of 128 KB are read, the offset returned to `zfs_read` also only increases by 50 KB. The next read would start at offset 50 KB. Therefore the offset is rounded up manually to the full block size of 128 KB.

Because of the issue with the offset reading the file is problematic. Tools like `dd` do not seem to work. Maybe when `dd` expects a certain number of bytes, but fewer bytes are returned, it repeats the read. When 128KB should be read in but only 50KB are returned `dd` tries to read the missing 78KB again, but at this point there are no bytes and nothing is returned. This might then just lead to another repeat. However, the `read`

system call does work. So for testing the read performance a C program is used. It calls `read` for a specified amount of blocks.

Apart from being an ideal solution for testing it still allows to utilize the implemented feature like Lustre would, and thus testing the read performance for pre-compressed blocks.

6.2.3 Results

Figure 6.1 shows the comparison of both test cases. The grey bars depict the time for reading the files with full blocks. The green bars show the times for reading the files with partial blocks. Table 6.1 shows the growth of the times with the modified ZFS.

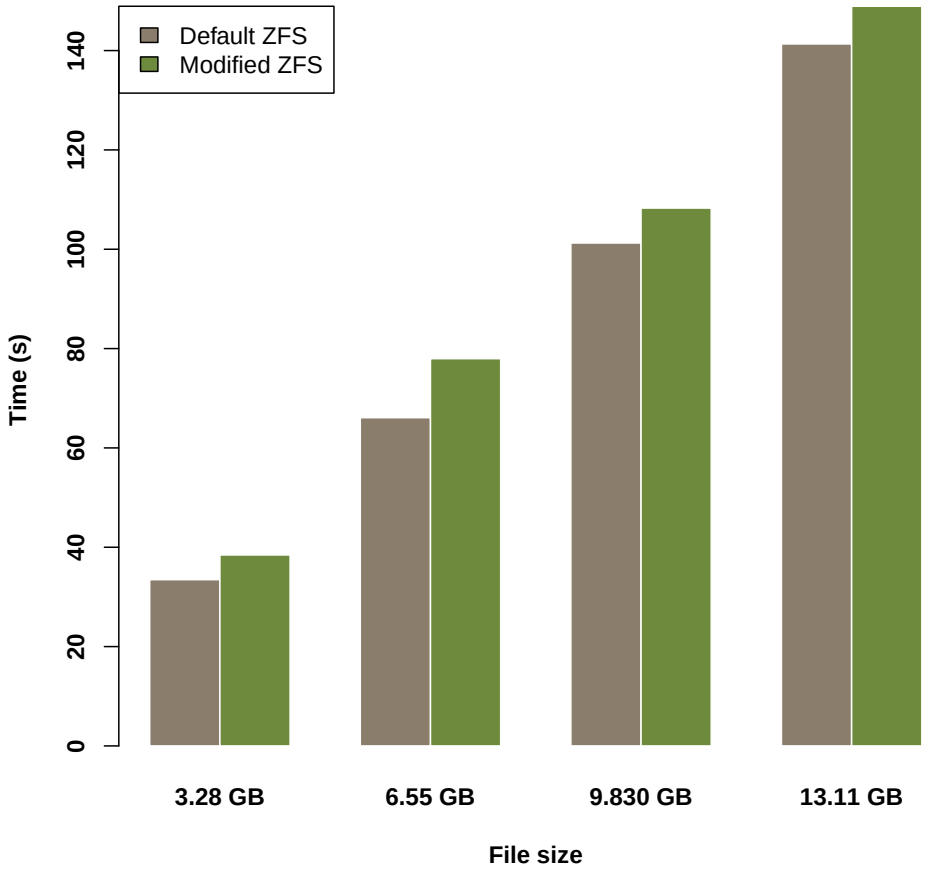

Figure 6.2: Comparison of times for reading the densely written files with the sparsely written files.

The results show that for the modified ZFS version a roughly constant overhead exists. Even for reading the 13.11GB file for which both ZFS versions should write full blocks.

Here an equal time would be expected since both write without gaps and readaheads should not be a problem anyway. Further investigations have to be done to find the source of this overhead. Maybe some improvements for allocating the blocks is done for uncompressed data which are not done for the modified ZFS because all data is tagged as compressed. A test for reading the 3.11 GB file from the ARC for both versions showed only a slight increase in time with 1.3 s for the unmodified ZFS and 1.6 s for the modified ZFS. For the half-filled blocks with 6.55 GB the difference between the files without gaps and with gaps is the largest. This might relate to the fact that the ARC used in the test is 6 GB large. Here the densely written file might be able to reuse some metadata which is still in the ARC, which cannot be done for the sparsely written file because of the additional overhead for the indirect blocks.

Overall the results show that ZFS is able to do efficient readaheads for the files with partial blocks. For every 3.11 GB that is added the time for reading increases by roughly 40 seconds. Hence the performance is entirely dependent on the bytes which are read.

File size	Full blocks	Partial blocks	Growth
3.28 GB	33 s	38 s	5 s
6.55 GB	66 s	78 s	12 s
9.83 GB	101 s	108 s	7 s
13.11 GB	141 s	149 s	8 s

Table 6.1: Growth of time for reading the data with the modified ZFS.

7 Conclusion and future work

This chapter conludes the work done over the course of this thesis. Moreover some ideas for future work are given.

Conclusion

The overall goal of the IPCC project for enhanced adaptive compression in Lustre is to bring adaptive transparent compression support to Lustre at multiple levels. To bring such a feature to a parallel distributed filesystem like Lustre is a extensive and complex task. But still the performance gains possible with compression are promising. Especially on distributed system where not only I/O is a bottleneck but also the network throughput. On top of this providing storage for petabytes of data is a costly undertaking in terms of storage devices as well as energy consumption by those devices. So reducing the data through compression also is able to reduce costs.

That makes clientside compression in Lustre a long wanted and desired feature as a discussion from 2006 shows [PJB06]. Back then one problem was that the local filesystem could not handle compressed data and blocks growing on rewrites. The layout and allocation functions would need big changes. Today Lustre allows ZFS as a backend filesystem which resolves the mentioned problems. This thesis tried to use this and combine the prototype of compression in Lustre with ZFS' compression functionality.

During the course of this thesis a prototype for writing pre-compressed blocks was implemented. To properly integrate those the compressed and the uncompressed size have to be known to ZFS. Because in ZFS the compressed size of a block is only available in 512 byte steps a header containing the compression algorithm as well as the compressed size in bytes was proposed.

It was described that generally two ways can be used by the OSD layer to write blocks to a ZFS filesystem. One using a zero-copy approach writing a whole block and one writing several pages for a block. Due to it's better suitability the zero-copy approach was chosen.

An existing implementation for sending and receiving compressed ZFS filesystems could be utilized. The implementation includes the ability to create compressed ARC buffer. Using this for every write a new compressed ARC buffer containing the physical size of the block is created. This passes the compressed size down to the I/O operation.

The main task for reading compressed data was essentially to suppress decompression. This was achieved with the help of a flag that is set when calling the introduced interface. This flag makes sure a block is not decompressed but instead returned to Lustre in it's compressed state

It became apparent that integrating pre-compressed data requires more changes to Lustre's OSD layer than expected. These changes were not implemented during the course of this thesis. Because of that for functionality and performance testing the ZPL layer was modified to simulate the wanted behaviour of Lustre's OSD layer.

With the first test results it could be shown that with the modified ZFS efficient readaheads are possible for the sparse data stripes written by Lustre. The tests showed that the time for reading the files decreases linear with higher compression rates. So for example if the compression by Lustre is able to decrease the data by 50% the time for reading the data from ZFS also decreases by almost 50%.

Future work

A working prototype for ZFS could be implemented. However, there is some work to do to in Lustre's OSD layer to make use of this prototype. The evaluation in this thesis is inadequate since it was only conducted through an modified ZPL Layer. To prove the results tests using the implemented feature in collaboration with Lustre have to be made. Additionally further functional and regression tests have to implemented to guarantee that the implementation is functioning.

Having ZFS not know the specific compression algorithm used and thus making the data not decompressible and effectively unreadable by ZFS is a downside of the current implementation. It would be desirable to make ZFS deal with the specified header containing compression algorithm and physical size, and thus making the pre-compressed data decompressible by ZFS. This way the data could be decompressed on the serverside without making changes to the Lustre server. It would be feasible fort Lustre to update small modification of the data via the normal write interface of ZFS. So for a read-modify-write case Lustre sends the data uncompressed to the server. ZFS decompresses the data on the server modifies it and writes it compressed to disk. On the other hand the current approach by storing the necessary information for decompression within the data increases the flexibility. Any transformed data block with a smaller physical size than logical size can be stored. Any compression algorithm could be used without caring if ZFS currently supports this particular algorithm.

Bibliography

[Ahr16] Matthew Ahrens. Lecture on openzfs read and write code paths. https://www.youtube.com/watch?v=ptY6-K78McY, March 2016.

[arc] Zfs 0.7.0-rc3 release code (arc.c). https://github.com/zfsonlinux/zfs/blob/master/module/zfs/arc.c.

[ash] Open zfs performance tuning. http://open-zfs.org/wiki/Performance_tuning#Alignment_Shift_.28ashift.29.

[BB] George Melikov Brian Behlendorf. Building zfs. https://github.com/zfsonlinux/zfs/wiki/Building-ZFS.

[bli10] „blizzard“ - an ibm power6-system. https://www.dkrz.de/pdfs/poster/ISC10_HardwareDKRZ.pdf, 2010.

[Bon05a] Jeff Bonwick. Jeff bonwick's blog: Raid-z. https://blogs.oracle.com/bonwick/entry/raid_z, November 2005.

[Bon05b] Jeff Bonwick. Jeff bonwick's blog: Zfs end-to-end data integrity. https://blogs.oracle.com/bonwick/entry/zfs_end_to_end_data, December 2005.

[Bon06] Jeff Bonwick. Jeff bonwick's blog: Zfs block allocation. https://blogs.oracle.com/bonwick/en/entry/zfs_block_allocation, November 2006.

[btr] Btrfs. https://btrfs.wiki.kernel.org/index.php/Main_Page.

[Col11] Yann Collet. Lz4 explained. http://fastcompression.blogspot.de/2011/05/lz4-explained.html, May 2011.

[Cor11] Jonathn Corbet. Kernel development: 3.11 merge window part 2. https://lwn.net/Articles/557814/, July 2011.

[dbu] Zfs 0.7.0-rc3 release code (dbuf.c). https://github.com/zfsonlinux/zfs/blob/master/module/zfs/dbuf.c.

[Dil10] Andreas Dilger. Zfs features & concepts toi, 2010.

[dkr] Mistral configuration. https://www.dkrz.de/Nutzerportal-en/doku/mistral/configuration.

[EB14] Andreas Dilger Eric Barton. *High Performance Parallel I/O (Chapman & Hall/CRC Computational Science)*. CRC Press, 2014.

[Ehm15] Florian Ehmke. Adaptive compression for the zettabyte file system. Master's thesis, Universität Hamburg, 2015.

[Eva14] David Evans. Adaptive replacement cache. `https://www.youtube.com/watch?v=_XDHPhdQHMQ`, April 2014.

[Fuc16] Anna Fuchs. Client-side data transformation in lustre. Master's thesis, Universität Hamburg, 2016.

[Geb09] Michael Gebetsroither. Ztest manpage, November 2009.

[Gre12] Brendan Gregg. Brendan's blog: Activity of the zfs arc. `http://dtrace.org/blogs/brendan/2012/01/09/activity-of-the-zfs-arc/`, January 2012.

[hdf15] Improving hdf5 compression performance. Technical report, The HF Group, 2015.

[Hei14] Jan Heichler. An introduction to beegfs. `http://www.beegfs.com/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf`, November 2014.

[his] History. `http://open-zfs.org/wiki/History`.

[huf] Huffman-kodierung: Beispiel. `http://www.binaryessence.de/dct/de000080.htm`.

[ins] Custom packages. `https://github.com/zfsonlinux/zfs/wiki/Custom-Packages`.

[ipc] Ipcc for lustre. `https://wr.informatik.uni-hamburg.de/research/projects/ipcc-l/start`.

[JMK14] Thomas Ludwig Julian M. Kunkel, Michael Kuhn. Exascale storage systems – an analytical study of expenses. *Supercomputing Frontiers and Innovations*, 2014.

[Ker10] Michael Kerrisk. *The linux programming interface*. no starch press, 2010.

[Kim] Dan Kimmel. compressed zfs send / receive. `https://github.com/openzfs/openzfs/pull/192/files`.

[Kis13] Saso Kiselkov. Illumos lz4 compression. `https://wiki.illumos.org/display/illumos/LZ4+Compression`, February 2013.

[KKK10] Aasia Khanum Kush K. Kella. Apcfs: Autonomous and parallel compressed file system. *International Journal of Parallel Programming*, 2010.

[Leo09] Brian Leonard. Zfs compression – a win-win. https://blogs.oracle.com/observatory/entry/zfs_compression_a_win_win, April 2009.

[Lev09] Adam Leventhal. Adam leventhal's weblog: Triple-parity raid-z. https://blogs.oracle.com/ahl/entry/triple_parity_raid_z, July 2009.

[LNB07] Shankar Pasupathy Jiri Schindler Lakshmi N. Bairavasundaram, Garth R. Goodson. An analysis of latent sector errors in disk drives. 2007.

[log] Logical volume management. https://en.wikipedia.org/wiki/Logical_volume_management.

[lusa] Lustre release 2.10.0. http://wiki.lustre.org/Release_2.10.0.

[lusb] Lustre release 2.9.0. http://wiki.lustre.org/Release_2.9.0.

[lusc] Lustre software release 2.x operations manual. http://doc.lustre.org/lustre_manual.xhtml.

[lz7] Binary essence - lempel-ziv-77 (lz77). http://www.binaryessence.de/dct/de000138.htm.

[Mac14] Kip Macy. How writes are scheduled for i/o. http://daemonflux.blogspot.de/2014/10/understanding-how-writes-translate-to.html, October 2014.

[MK16] Thomas Ludwig Michael Kuhn, Julian Kunkel. Data compression for climate data. 2016.

[MKM14] Robert N.M. Watson Marshall Kirk McKusick, George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2014.

[ML15] Michael Kuhn Michael Lautenschlager, Panagiotis Adamidis. *Big Data and High Performance Computing*. IOS Press BV, 2015.

[neh] Scientific computing: Teaching ressourcen. https://wr.informatik.uni-hamburg.de/teaching/ressourcen/start.

[NM03] Dharmendra S. Modha Nimrod Megiddo. Arc: A self-tuning, low overhead replacement cache. 2003.

[Ora] Oracle. Sending and receiving zfs data. https://docs.oracle.com/cd/E18752_01/html/819-5461/gbchx.html.

[PJB06] Goswin von Brederlow Peter J. Braam. Lzw block-level compression for improving lustre read/write speeds? http://lustre-discuss.lustre.narkive.com/H5rxb6ll/lzw-block-level-compression-for-improving-lustre-read-write-speeds, 2006.

[RD09] John Dawson Rahul Deshmukh, Andreas Dilger. End-to-end data integrity design. Technical report, Sun microsystems, 2009.

[Roc08] Ben Rockwood. zdb: Examining zfs at point-blank range, November 2008.

[SI15] Li Xi Shichia Ihara. Osd-btrfs, a novel lustre osd based on btrfs. http://cdn.opensfs.org/wp-content/uploads/2015/04/OSD-Btrfs-a-Novel-Lustre-OSD_Ihara_Xi.pdf, April 2015.

[SM06] Inc. Sun Microsystems. Zfs on-disk specification. 2006.

[TM10] Yannis Klonatos et al. Thanos Makatos. Using transparent compression to improve ssd-based i/o caches. 2010.

[Wika] Adaptive replacement cache. https://en.wikipedia.org/wiki/Adaptive_replacement_cache.

[wikb] Extent (file systems). https://en.wikipedia.org/wiki/Extent_(file_systems).

[wikc] Lz77 and lz78. https://en.wikipedia.org/wiki/LZ77_and_LZ78.

[zdb15] zdb – zfs debugger command examples. http://fibrevillage.com/storage/252-zdb-zfs-debugger-command-examples, June 2015.

[ZFSa] Zfs. https://en.wikipedia.org/wiki/ZFS.

[zfsb] Zfs metadata compression. https://docs.oracle.com/cd/E26505_01/html/E37386/chapterzfs-7.html.

Appendices

Acronyms

APCFS The Autonomous and Parallel Compressed File System. 5, 48

ARC Adaptive Replacement Cache. 5, 26, 27, 29, 32, 41, 51–53, 55–57, 59–61, 64–68, 70, 71, 73, 74, 86

ASCII American Standard Code for Information Interchange. 13

btrfs B-tree filesystem. 5, 7, 47, 48

CentOS Community Enterprise Operating System. 15, 83

CERN European Organization for Nuclear Research. 21

DKRZ Deutsches Klimarechenzentrum. 7, 70

DMU Data Management Unit. 23, 26–29, 40, 47, 51, 53–55, 60, 65, 71

DSL Dataset and Snapshot Layer. 30

DVA Data Virtual Address. 24

ext4 Extended File System 4. 17, 21, 28

FLOPS floating-point operations per second. 7

HDF5 Hierarchical Data Format 5. 5, 47, 48

HPC high performance computing. 7

I/O Input/Output. 3, 6–9, 15, 17, 28, 30, 32, 47, 48, 51, 54, 55, 59, 60, 62, 65, 74

IPCC Intel Parallel Computing Centers. 3, 8, 74

L1ARC Level 1 Adaptive Replacement Cache. 27, 51–53, 61

L2ARC Level 2 Adaptive Replacement Cache. 26–28, 51–53, 86

LRU Least Recently Used. 26, 29

LV logical volumes. 19

LZ4 Lempel Ziv 4. 12

LZO Lempel-Ziv-Oberhumer. 47

LZW Lempel-Ziff-Welch. 48

MDS Metadata Server. 15, 16

MFU Most Frequently Used. 26, 27

MGS Management Server. 15, 16

MGT Management Target. 15, 16

MPI-OM Max Planck Institute's Global Ocean/Sea-Ice Model. 14

MRU Most Recently Used. 26, 27

NFS Network File System. 55

NTFS New Technology File System. 5, 48

OSD Object Storage Device. 17

OSS Object Storage Server. 15, 16

OST Object Storage Target. 16, 17

POSIX Portable Operating System Interface. 16, 26, 28, 31, 34, 47

RAID Redundant Array of Independent Disks. 19, 21

RAM Random-access Memory. 27, 70

RHEL Red Hat Enterprise Linux. 15

RLE Run-Length Encoding. 12

SHA Secure Hash Algorithm. 21

SPA Storage Pool Allocator. 27

SSD Solid-state Drive. 27, 47, 48

TXG Transaction Group. 56, 59

UFS Unix File System. 28

VFS Virtual File System. 25, 26, 46, 47

ZAP ZFS Attribute Processor. 26, 31

ZDB ZFS Debugger. 33, 40

ZIL ZFS Intent Log. 26

ZIO ZFS I/O. 30

ZLE Zero Length Encoding. 12, 14, 22, 44

ZPL ZFS Posix Layer. 26, 28, 31, 44, 46, 71, 75

ZVOL ZFS Volume. 26, 28

Building ZFS

This section shows the installation of ZFS on CentOS 7 [ins].

```
1 $ sudo yum groupinstall "Development Tools" parted lsscsi
    ↪ wget ksh
2 $ sudo yum install kernel-devel zlib-devel libattr-devel
    ↪ libuuid-devel libblkid-devel libselinux-devel
    ↪ libudev-devel
```

First the Solaris Porting Layer (SPL) needs to be installed which provides core interfaces required for OpenZFS on Linux.

```
1 $ git clone https://github.com/zfsonlinux/spl.git
2 $ cd spl
3 $ ./autogen.sh
4 $ ./configure --with-spec=redhat
5 $ make pkg-utils pkg-kmod
6 $ sudo yum install *.<arch>.rpm
```

Afterwards ZFS can be installed.

```
1 $ cd ..
2 $ git clone https://github.com/zfsonlinux/zfs.git
3 $ cd zfs
4 $ ./autogen.sh
5 $ ./configure --with-spec=redhat
6 $ make pkg-utils pkg-kmod
7 $ sudo yum install *.<arch>.rpm
```

For additional correctness tests and enabling all asserts ZFS can be configured with `-enble-debug`

Debugging ZFS

Debug messages

For printing debug messages `dprintf()` can be used. The output of those messages has to be explicitly enabled with:

```
1 $ echo 1 > /sys/module/zfs/parameters/zfs_dbgmsg_enable
```

The output can be shown with:

```
1 $ cat /proc/spl/kstat/zfs/dbgmsg
```

ztest

The tool `ztest` is a ZFS unit test. It runs by default 10 minutes and uses files stored in `/tmp` for creating pools. The option `-V` increases the verbosity. Additional `V`s will further increase the output. During the run `ztest` creates `ztest.` files which can be removed after `ztest` finished [Geb09]. Properties of the pools `ztest` creates can be changed form the default value with various options. For example with `-s` the size of each vdev can be changed.

In-tree build

For developing and debugging ZFS can also be installed in-tree [BB]. The steps until after configuring stay the same but ZFS is installed with:

```
1 $ make -s -j$(nproc)
```

For this build additonal helper scripts have to be executed. `zfs-helpers.sh` creates symlinks on the system from the installation location. `zfs.sh` loads the kernel modules and unloads them the option `-u`.

```
1 $ sudo ./scripts/zfs-helpers.sh -i
2 $ sudo ./scripts/zfs.sh
```

`zloop.sh` repeatedly runs `ztest` with randomized options.

```
1 $ sudo ./scripts/zloop.sh
```

ZFS also provides a test suite which does extensive regression and functionality testing. It is executed with:

```
1 $  ./scripts/zfs-tests.sh -vx
```

Fields of `arcstat.py`

```
1  Field definitions are as follows:
2      l2bytes : bytes read per second from the L2ARC
3       l2hits : L2ARC hits per second
4         read : Total ARC accesses per second
5         dmis : Demand misses per second
6          mru : MRU List hits per second
7        mread : Metadata accesses per second
8            c : ARC Target Size
9           ph% : Prefetch hits percentage
10      l2hit% : L2ARC access hit percentage
11         pm% : Prefetch miss percentage
12         mfu : MFU List hits per second
13         mm% : Metadata miss percentage
14       pread : Prefetch accesses per second
15        miss : ARC misses per second
```

```
16        mrug : MRU Ghost List hits per second
17        dhit : Demand hits per second
18        mfug : MFU Ghost List hits per second
19        hits : ARC reads per second
20         dm% : Demand miss percentage
21        miss% : ARC miss percentage
22        mhit : Metadata hits per second
23         dh% : Demand hit percentage
24         mh% : Metadata hit percentage
25        pmis : Prefetch misses per second
26     l2asize : Actual (compressed) size of the L2ARC
27     l2miss% : L2ARC access miss percentage
28      l2miss : L2ARC misses per second
29        mmis : Metadata misses per second
30        phit : Prefetch hits per second
31        hit% : ARC Hit percentage
32       eskip : evict_skip per second
33       arcsz : ARC Size
34        time : Time
35      l2read : Total L2ARC accesses per second
36      l2size : Size of the L2ARC
37      mtxmis : mutex_miss per second
38       dread : Demand accesses per second
```

Listing 1: Overview of fields `arcstat.py` is able to print.

List of Figures

List of Listings

List of Tables

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