Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelorarbeit

Efficient handling of compressed data in ZFS

vorgelegt von

Hauke Stieler

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

Studiengang:	Software-System-Entwicklung
Matrikelnummer:	6664494
Erstgutachter:	Dr. Michael Kuhn
Zweitgutachter:	Anna Fuchs
Betreuer:	Dr. Michael Kuhn, Anna Fuchs

Hamburg, January 16, 2019

Abstract

Since the beginning of the computer era, the speed of computation, network and storage as well as the storage capacity, have grown exponentially. This effect is well known as Moore's law and leads to increasing gaps between the performance of computation and storage. High performance computing organizations like the Deutsches Klimarechenzentrum (DKRZ) suffer from these gaps and will benefit from solutions introduced by the IPCC for Lustre projects. To close this cap applications and file systems use compression in order to speed up the storage of data. Even if file systems like ZFS are already able to compress data, distributed file systems like Lustre are not completely able to use these functionality in an efficient way.

A client-side compression implementation done by Anna Fuchs, presented in her thesis "Client-Side Data Transformation in Lustre" ([Fuc16]), reduces the size of the data before it is send over the network to the storage server. First changes to ZFS were presented by Niklas Behrmann in his thesis "Support for external data transformation in ZFS" ([Beh17]) adding support to ZFS for externally compressed data. The interconnection of these two theses was done by Sven Schmidt in his thesis "Efficient interaction between Lustre and ZFS for compression" using the client-side compression with the new API functions of ZFS.

This thesis presents an efficient way to handle externally compressed data in ZFS in order to do fast read and write calls. The code changes and complexity of the solutions were kept at a minimum and are as maintainable as possible. In order to do this, a refactoring of the existing work was done before adding features to the code base. Introducing explicit flags and simplifying code paths enables ZFS to handle the data as efficient as possible without neglecting the quality of the software. A correctness and performance analysis, using the test environment of ZFS, shows the efficiency of the implementation and reveals also tasks to do in future work.

Contents

1. Introduction

This chapter introduces the thesis and its topics of file systems, supercomputing and performance. To illustrate the importance of well performing clusters and file systems, the domain of weather and climate prediction is used. Producing and working with climate simulations is also the task of the DKRZ (Deutsches Klimarechenzentrum). Increasing the efficiency of calculations by improving storage systems is ones task of the working group for scientific computing at the Universität Hamburg, where this thesis was written.

1.1. Historical background

In 1950, the first equations and algorithms for a numerical weather forecast have been created. They were designed and programmed by John and Klara von Neumann for the ENIAC, which produces a 24 hour forecast that took 24 hour of computation. A rather simple barotropic model had been used, which was replaced by alternative multi-level models in 1966 when the National Meteorological Center in Washington began its operation with a six-level model. Other countries began a weather forecast operation as well, e.g. the Deutscher Wetterdienst in Germany. After a few more years, in 1972, they already used ten levels for their models. [Lyn08]

Weather as well as climate models used the computer of that time. The first climate simulation, created by Norman A. Phillips in 1956, also used a multi-layer model, but with only two layers. He did the computation on the MANIAC I, which was also designed by John von Neuman who was hugely impressed by Phillips work [Lyn08]. Until the mid 1980s, only atmospheric dynamics had been used and predictions near ground-level required corrections based on statistics. Also land and sea surfaces were considered since then, both responsible for factors like heat transportation, radiation and humidity. Until the late 1990s the composition of the atmosphere went into the models, which includes the concentration of sulphate aerosols, carbon dioxide and other chemical components. Nowadays even more complex structures like dynamic vegetation are considered in climate models that are currently used [Lud18, p. 38].

1.2. Motivation

All these improvements of the models were only possible because of the increased computational power of computers and supercomputers. The exponential decrease of the time for a calculation or simulation enables the usage of more complex models [Log16, p. 16]. But not only the amount of considered factors, also the level of details increased. Climate models use a 3D grid that is placed upon the earth's surface, where each cell has a specific size. With a smaller cell size, the grid becomes more dense, a simulation produces more data and the results are more detailed.

The Blizzard supercomputer, installed in 2009 at the DKRZ, produced about 10 PetaByte of data per year saving this with a rate of up to 5 GB/s into the archive. Up to this point the supercomputers at the DKRZ showed an exponential growth rate regarding the amount of produced data [DKRc]. In 2016 the new Mistral supercomputer fully replaced the Blizzard offering even more computational power by increasing the amount of floating point operations by a factor of 22 (Blizzard: 158 TFLOPS, Mistral: 3.6 PFLOPS) [DKRa].

The data produced by a simulation has to be stored and will be processed in most cases at a later point in time. Unfortunately there is a gap between the speed of computation, the storage speed and the storage capacity [KKL16, p. 1]. To fill this gap and to keep costs at a minimum, more efficiency is needed. One way to increase efficiency is to compress the data before it gets transferred, for example via the network to a storage server. Also the network can play a role in the overall performance as it can increase the performance when data is compressed before sending it to a server [KKL16, p. 2].

1.3. The IPCC for Lustre project

The Intel® Parallel Computing Centers (IPCC) are institutions modernizing open source code to increase scalability and parallelism. The scientific computing group aims to implement compression support in the Lustre file system at multiple levels.

One part of this is the implementation of client-side compression, where Lustre interacts with ZFS in order to store compressed data. Anna Fuchs implemented a client-side compression mechanism in Lustre [Fuc16] and Niklas Behrmann added in a first approach the support for externally transformed data in ZFS [Beh17]. Both works were then combines by Sven Schmidt, who used the changes made to ZFS within the Lustre server [Sch17].

1.4. Thesis goals

This thesis will add support into ZFS to handle the externally compressed data from Lustre in a most efficient way. Writing and reading externally compressed data efficiently can be done by using existing compression functionality of ZFS. To do so, an additional write and read path is introduced, which makes use of these existing features.

This thesis describes the design decisions and implementation of the additional code paths. An analysis of the correctness and performance will show if the changes are as efficient as desired. Also future and related work on this topic is described.

2. Overview

This chapter describes the fundamental aspects of compression as well as the two file systems ZFS and Lustre. In the section covering compression, multiple algorithms are introduced and described in detail. After that an introduction into ZFS is given, describing its architecture and basics of the internal functioning. The section introducing Lustre describes the architecture as well and shows how these two file systems interact with each other.

2.1. Compression

Compression is a mechanism to encode data producing an output that uses less space then the original data. Therefore one can speak of a physical and a logical size: The physical size is the size of the compressed data on disk whereas the logical size is the size of the decompressed data used by applications. Two major categories of compression algorithms can be determined: Lossless and lossy algorithms.

A lossless algorithm produces compress data that does not loose any information. When this data is decompressed, it equals exactly the original data since no information got lost. Lossy compression on the other hand intentionally removes information in order to reduce the size of the output even more. Therefore does the decompressed data differ from the original one.

This section will introduce some of the most important algorithms and techniques, also with respect to ZFS's compression mechanisms.

2.1.1. Compression ratio and speed

In order to evaluate to quality of a compression algorithm, two criteria were often used as described by [DCG18, p. 6]: The compression ratio and the speed of the algorithm.

The compression ratio $CR(F)$, where F is the file being compressed, is simply the compressed size over the original size. Therefore a smaller value means a better compression of the algorithm.

$$CR(F) = \frac{\mathit{filesize}(F_{orig})}{\mathit{filesize}(F_{comp})}$$

Using the reciprocal of the fraction is also valid, where a higher value means a better compression.

When the compression algorithm is implemented, the speed becomes an important factor. Even though the speed depends on the Hardware and might fluctuate between executions, it can be theoretically determined by the following simple formula. The compression speed is $CS(F)$ and the decompression speed $DS(F)$.

$$CS(F) = \frac{\mathit{filesize}(F_{orig})}{t_{comp}}$$
$$DS(F) = \frac{\mathit{filesize}(F_{orig})}{t_{decomp}}$$

The speed is often expressed as bytes per second, depending of the unit used for *filesize*(F).

2.1.2. Lossy compression

Lossy compression, sometimes referred to as irreversible compression, is a form of compression that looses some of the information from the original data in order to reduce the size of the output even more. Famous examples for lossy compression mechanisms are JPEG, MPEG, the H.264/H.265 codecs and MP3, just to name a few [Wik18b].

Because the decompressed data is not necessarily equal to the original data and precise results might be distorted, lossy compression is not always suitable for data produced by climate algorithms [KKL16, p. 3].

On a file system level particularly lossy compression is not suitable, because the knowledge what kind of data should be compressed is missing. The distortion of data might invalidate it, which makes the data unusable.

2.1.3. Lossless compression

As mentioned earlier, especially lossless compression is relevant for climate data and is implemented in ZFS. This section covers some of the most fundamental and important algorithms for lossless compression.

Run-length encoding (RLE)

The run-length encoding (referred to as RLE) saves the amount of repeating characters instead of the characters themselves. When working on bit-level, a single bit is taken and its amount of occurrences is saved. The amount of bits used for this amount is specified by the largest number of successively arising bits in the given data. [SS13]

For example: The bit string `000001111111110000` (18 bits long) has five zeros at the beginning, nine ones in the middle and four zeros at the end. Therefore the largest amount of occurrences is nine, which takes four bits to encode in binary: `1001`. The different parts can now be encoded as seen in table table 2.1. Putting all together results in `001011100100100`, which is 15 bits long and saves three bits on comparison to the original data.

bit string	bit to encode	amount in binary	result
00000	0	0101	0 0101
111111111	1	1001	1 1001
0000	0	0100	0 0100

Table 2.1.: RLE of different parts of a bit string.

Three bits are not much, but when a whole megabyte of zeros can be stored in only 24 bits (as `011110100001001000000000`), the RLE is very efficient. Compressing a lot of repeating zeros may not seem that useful but is actually a use case in scientific data, which can consist of sparse matrices with lots of zeros in it [Dö16]. ZFS supports the compression of repeating zeros, which they call ZLE (zero-length encoding)[1].

Huffman coding

As of the overview presented by Sharma ([Sha10]), Huffman coding also compresses characters that occur very often, but they do not need to repeatedly occur. Instead, Huffman coding uses fewer bits for very often appearing characters and more bits for characters that do not occur very often.

To do so, the amount of occurrences has to be count. Lets take the word `cocoon` as an example, the amount of occurrences and their probability can be seen in table 2.2. The next step builds up a binary tree, called Huffman tree. First a node is created for every character and a weight is assigned, which can be the amount of occurrences or the probability. The list of nodes needs to be sorted by the weight at any time. A simple algorithm will then create the tree:

character	occurrences	probability
o	3	0.5
c	2	$0.\overline{3}$
n	1	$0.1\overline{6}$

Table 2.2.: Occurrence counts and occurrence probabilities of the word `cocoon`.

1. Take the two nodes with the lowest weight and remove them from the list
2. Create a parent node for these two nodes, assign the sum of the weights to the parent and assign one bit to the edges (e.g. left edge 0, right edge 1)
3. Add the parent to the list
4. Go to step one until there is only one node left

These steps lead to the Huffman tree, which can be seen in figure 2.1.

After the tree had been created, the actual Huffman code can be determined. This can easily be done by searching the letter in the tree starting with the root node (e.g. with a depth-first search). The codes of the example can be seen in table 2.3.

character	binary code
o	0
c	10
n	11

Table 2.3.: Final Huffman coding of the word `cocooon`.

[1] As seen in the source file `module/zfs/zle.c`.

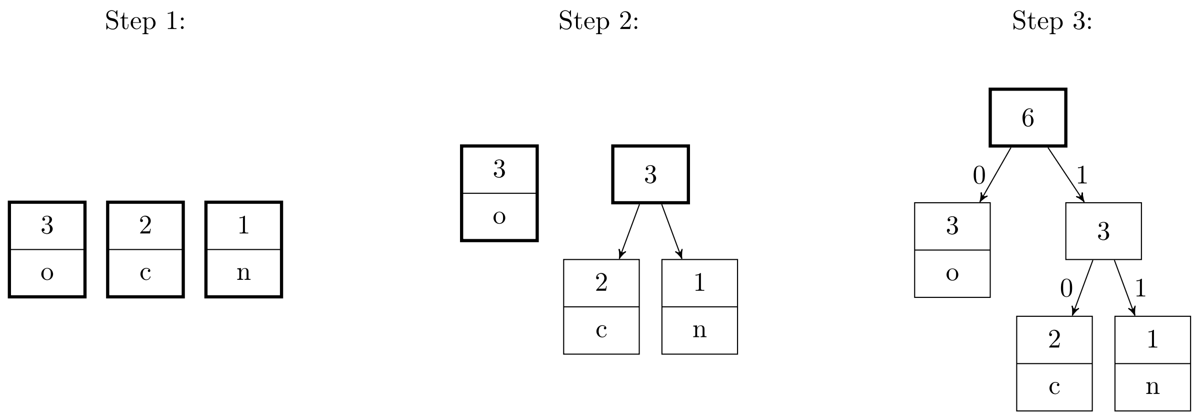


Figure 2.1.: Construction of the Huffman tree. The sorted list of nodes is illustrated by the boxes with a thicker border.

Using 8 bits per character (as the most basic UTF-8 encoding for ASCII characters does), the uncompressed word `cocoon` would be 48 bits long. With Huffman code, the size of the actual data is reduced to only 9 bits: `10 0 10 0 0 11`.

LZ77

LZ77 is part of the Lempel-Ziv algorithm family and very similar to the patented LZ78 algorithm. LZ77 is used in more complex compression algorithms such as DEFLATE [Deu96b], which itself is used in formats like PNG and ZIP [Wik18a]. ZFS supports a member of the LZ algorithm family, called LZ4. Because LZ77 is more frequently used, this section describes its basic functionality as presented by [Mü08].

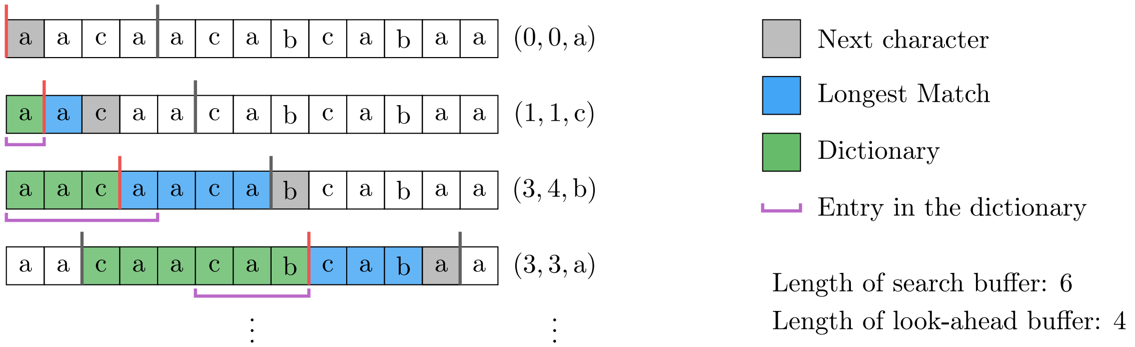

a a c a a c a b c a b a a (0, 0, a)
a a c a a c a b c a b a a (1, 1, c)
a a c a a c a b c a b a a (3, 4, b)
a a c a a c a b c a b a a (3, 3, a)
⋮
⋮
Next character
Longest Match
Dictionary
Entry in the dictionary
Length of search buffer: 6
Length of look-ahead buffer: 4

Figure 2.2.: As the windows move forward multiple matches are found (values in parentheses). [Shi+18, p. 5]

The idea of LZ77 is to find repeating parts in the data, store them in a dictionary and replace the duplicates with a reference to such a dictionary entry. There are two sliding, fixed sized parts of the data (called buffer): One called search buffer (used as dictionary) and one called look-ahead buffer (used as buffer for matches). LZ77 tries to find the longest prefix from the look-ahead buffer that has a match in the dictionary. If it succeeds, it replaces this prefix from the look-ahead buffer of length l with a reference to the dictionary. After that, the windows moves forward by $l + 1$ characters to find the next match.

The reference (sometimes also called pointer) is a triplet containing the relative offset to the dictionary entry, the length of the match and the next character that is not in the match: (*offset*, *length*, *character*). An example of the algorithm can be seen in figure 2.2.

If no match has been found, offset and length is 0 and the symbol is the first character of the look-ahead buffer. The windows then moves one character forward.

DEFLATE and gzip

The DEFLATE algorithm is nothing else but the concatenation of LZ77 and Huffman coding [Deu96b]. Gzip is the name for a file format and a compression application as described in the RFC 1952 [Deu96a]. Its compression mechanism is based on the DEFLATE algorithm and the size of the output can be controlled by the level of compression, which is a number from 0 (no compression at all) to 9 (best compression). The Gzip compression standard is mainly used via the library zlib [GA17], which is also used in the Linux Kernel by file systems like SquashFS as described in section 4.1.1.

ZFS has support for all compression levels of gzip, but compression in ZFS is described in section 3.1.2.

2.2. ZFS

ZFS is a 128-bit POSIX compliant file system with support for copy-on-write (COW), transactions, data integrity and many other features [Mic06]. In addition, ZFS combines normal file system features with the ones of a volume manager. This enables ZFS to be used in more sophisticated ways, for example for snapshots or special RAID levels like RAID-Z. Very important for this thesis is the fact that ZFS is already able to compress data using several predefined algorithms (later described in section 3.1).

2.2.1. History

ZFS was originally developed at Sun Microsystems starting in 2001 (announced 2004) as proprietary, closed-source software. With the creation of the OpenSolaris project, ZFS became also open-source and was published under the Common Development and Distribution License (CDDL). A few years later in 2008 the development of a Linux port started, also under the CDDL. Even though the CDDL and the GPL are free and open-source licenses, they are incompatible to each other. That means that ZFS cannot be merged into the Linux kernel, which stays under the GPL, and has to stay in a separate code base.

Because ZFS was ported to multiple platforms and the coordination has become more difficult, the OpenZFS project was founded in 2013. This projects aims to organize the development and offer a centralized platform for interested groups and organizations.

2.2.2. On disk structure

Before the internal code architecture is described, a look at the on disk structure (as described in [Mic06]) is needed to understand how ZFS organizes data.

The uberblock and MOS

ZFS stores everything in a tree of blocks with a root block called uberblock. Every block stores a checksum of all child blocks and block pointers. This builds up a Merkle Tree (as showed in figure 2.3) and ensures data integrity because errors can be detected with the help of the checksums.

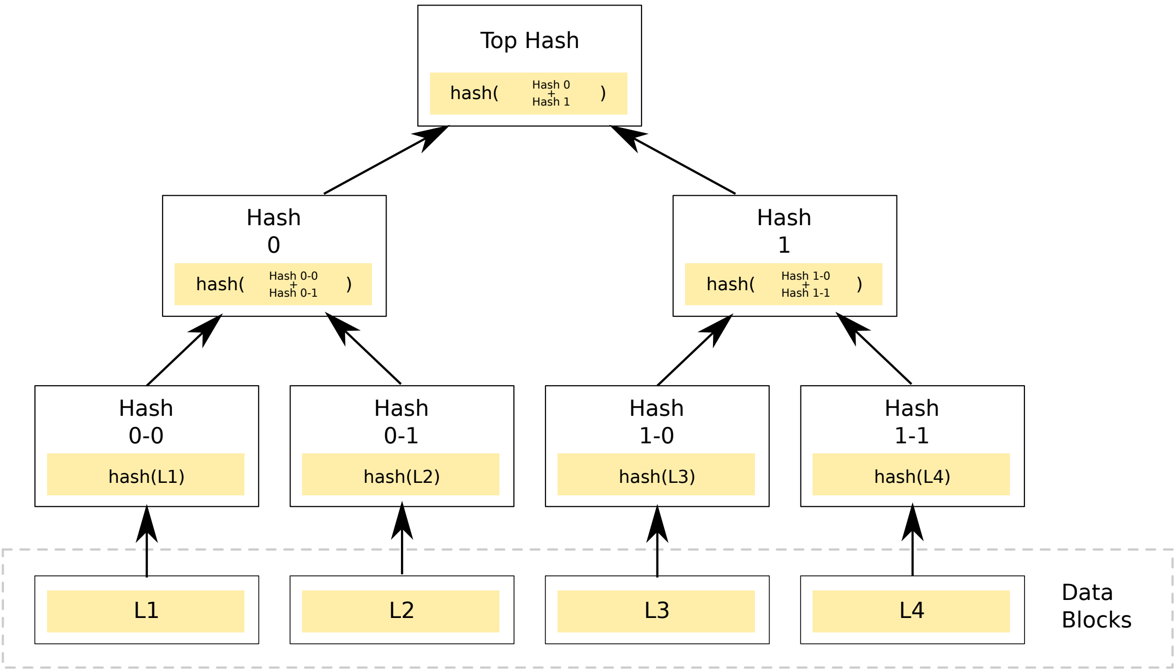


Figure 2.3.: A basic merkle tree with entries containing checksums [Wik19].

The uberblock contains the first block pointer that points to the meta object set (MOS), which is a dataset of type `DMU_OST_META` saving pool wide metadata [Ope13]. The MOS has a reference to the first dnode, which is the beginning of the tree of dnodes and blocks.

The dnode

The dnode structure is comparable to the inode from the UFS or ext file systems and organizes a set of blocks.

In general a dnode describes an object in the object set and is 512 byte large. It saves a lot of metadata like the amount of indirections, the checksum algorithm being used or bonus buffer information. Except for two fields (`dn_blkptr` and `dn_bonus`), all fields have a fixed size. The variable sized field `dn_blkptr` stores up to three block pointers (specified by `dn_nblkptr`) and `dn_bonus` contains a bonus buffer.

DVA

To fully understand the block pointer the data virtual address (DVA) needs to be described. A DVA uniquely identifies the location of a block and consists of two parts. The first part of a DVA is the 32-bit long ID of the VDEV the block contains to. The second part is an offset within the VDEV, which is specified in bytes but as a multiple of 512 because a disk sector is 512 bytes large. A calculation for the offset is therefore done by the following operations:

$$\textit{physical offset} = (\textit{offset}) \ll 9 + 0x400000$$

A bit shift of 9 ($(\textit{offset}) \ll 9$) is done to convert the offset from a sector unit into a byte unit. The 4MB static offset (0x400000) is the sum of two `vdev_labels` (each 256KB) and the boot block (3.5KB).

Often the size of the block a DVA points to is also given when writing a DVA down. Therefore it can be written as `<vdev>:<offset>:<size>`.

Block pointer

A block pointer is 128 byte large and is used to locate, verify and describe a block on disk. This is done by saving up to three DVAs pointing all to a copy of the same data. The size of this data is specified by the three size parameters `psize`, `lsize` and `asize`:

`psize` Physical size on disk (rounded up to the next block size)

`lsize` Logical size, which is the size of the data after decompression or decryption

`asize` The total amount of all blocks holding this data including all additional information (for example metadata or parity blocks for RAID-Z)

One other important part of the block pointer is the 256 bit large checksum, which is an essential part of the write process. ZFS supports several checksumming algorithms like SHA-256, fletcher2 and fletcher4. The role of the checksum is described in section 2.2.4.

The full structure of the block pointer[2] is shown in figure 2.4 with an explanation of each field in table 2.5.

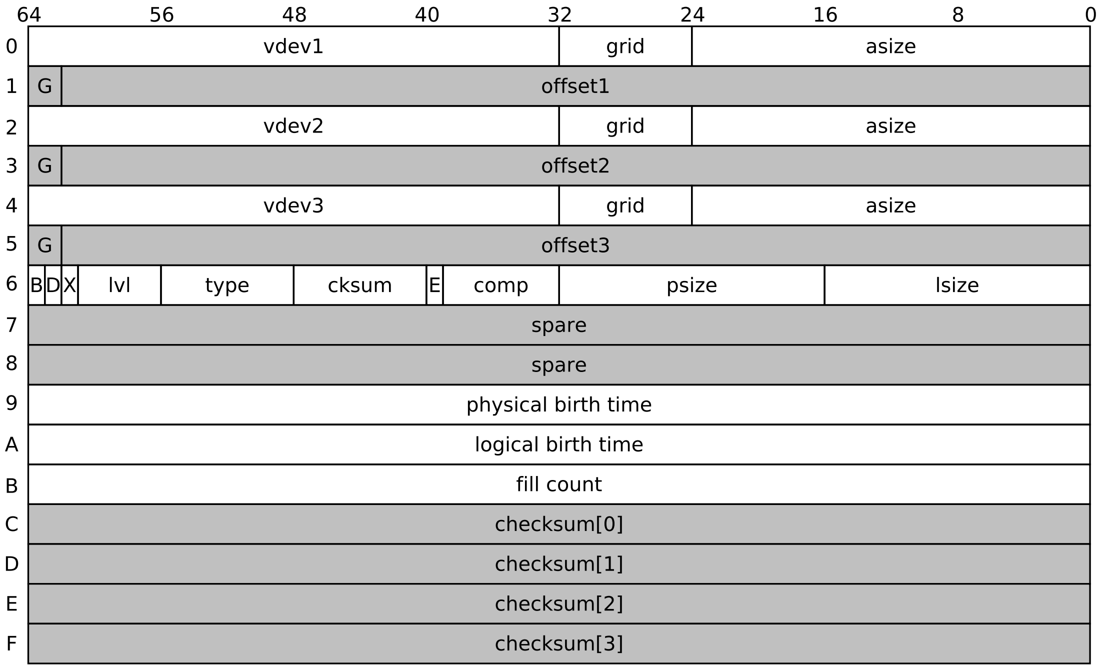
Embedded data

When data that should referenced by a block pointer is not larger than 112 bytes, it is directly stored in the block pointer. Therefore the `E` flag in the block pointer is set to 1 as described in table 2.5. Because a block pointer is only 128 bytes large, the remaining space contains necessary metadata like the logical size, physical size or the level of indirection. However block pointer containing embedded data do not have a checksum or any DVAs.

2.2.3. Internal architecture

ZFS consists of three main layers with several different sub-layers (as illustrated in figure 2.5), which all have different tasks.

[2] Described in `include/sys/spa.h` including a graphic.



<table>
<tr><th></th><th colspan="8">64</th><th colspan="8">56</th><th colspan="8">48</th><th colspan="8">40</th><th colspan="8">32</th><th colspan="8">24</th><th colspan="8">16</th><th colspan="8">8</th><th>0</th></tr>
<tr><td>0</td><td colspan="32">vdev1</td><td colspan="8">grid</td><td colspan="24">asize</td></tr>
<tr><td>1</td><td colspan="2">G</td><td colspan="62">offset1</td></tr>
<tr><td>2</td><td colspan="32">vdev2</td><td colspan="8">grid</td><td colspan="24">asize</td></tr>
<tr><td>3</td><td colspan="2">G</td><td colspan="62">offset2</td></tr>
<tr><td>4</td><td colspan="32">vdev3</td><td colspan="8">grid</td><td colspan="24">asize</td></tr>
<tr><td>5</td><td colspan="2">G</td><td colspan="62">offset3</td></tr>
<tr><td>6</td><td>B</td><td>D</td><td>X</td><td colspan="5">lvl</td><td colspan="8">type</td><td colspan="8">cksum</td><td>E</td><td colspan="7">comp</td><td colspan="16">psize</td><td colspan="16">lsize</td></tr>
<tr><td>7</td><td colspan="64">spare</td></tr>
<tr><td>8</td><td colspan="64">spare</td></tr>
<tr><td>9</td><td colspan="64">physical birth time</td></tr>
<tr><td>A</td><td colspan="64">logical birth time</td></tr>
<tr><td>B</td><td colspan="64">fill count</td></tr>
<tr><td>C</td><td colspan="64">checksum[0]</td></tr>
<tr><td>D</td><td colspan="64">checksum[1]</td></tr>
<tr><td>E</td><td colspan="64">checksum[2]</td></tr>
<tr><td>F</td><td colspan="64">checksum[3]</td></tr>
</table>

Figure 2.4.: Visualization of the several fields of the block pointer [Beh17, p. 23].

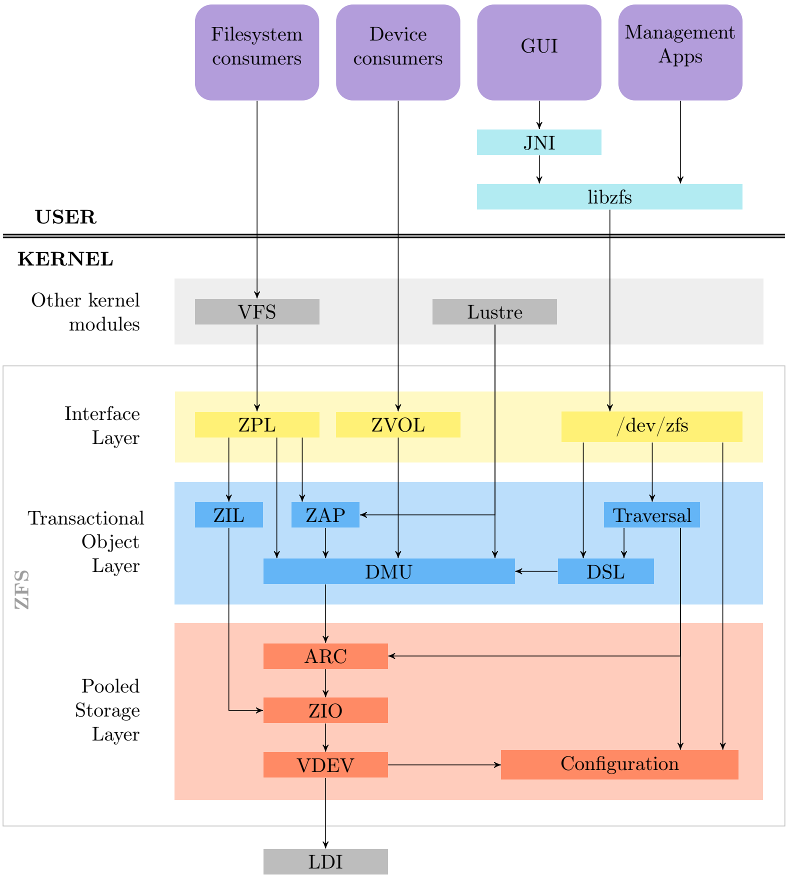


Figure 2.5.: Internal architecture of ZFS (based on [Ora13])

Field	**Description**
vdev$n$	Virtual device ID
grid	RAID-Z layout information (reserved for future use)
asize	Allocated size (including RAID-Z parity and gang block headers)
G	Gang block indicator
offset$n$	Offset into virtual device
B	Byteorder (endianness)
D	Deduplication
X	Encryption
lvl	Level of indirection
type	DMU type of the block (`dmu_object_type_t`)
cksum	Used checksum algorithm (`zio_checksum`)
E	0 if the block does not contain embedded data, 1 otherwise
comp	Used compression algorithm (`zio_compress`)
psize	Physical size on disk
lsize	Logical size, which is the size of the data after decompression or decryption
spare	Some extra space for future use
physical birth time	Transaction group this block was written in
logical birth time	Transaction group this block was born in
fill count	Number of non-zero blocks under this block pointer
checksum[*4*]	256-bit checksum (split into four 64 bit values) of the data this block pointer describes

Table 2.5.: Descriptions for each field of a block pointer.

Interface layer

The first layer is the interface layer where the ZFS POSIX layer (ZPL) and the ZFS volume layer (ZVOL) are located. This layer is mainly used by applications using ZFS as normal file system, applications that directly use the device files and management applications via the library `libzfs`. Normal applications writing or reading data use ZFS via the system call (syscall) interface with syscalls like `read` or `write`. Each syscall is then passed through the VFS into the ZPL where ZFS decides what to do.

ZFS can also be used via ZVOLs, which are visible to applications as normal block devices. A ZVOL can for example be used to build up a different file system upon ZFS.

Transactional Object Layer

The transactional object layer mainly consists of the data management unit (DMU). This component interacts with the storage pool allocator (SPA, described below) working directly with blocks and provides an interface to other components that deal with transactions. Therefore operations on object performed with this component, like read, write, create and destroy, are always assigned to a transactions passed to the DMU.

Keeping the stored data consistent is he main task of the DMU, which is done by performing a copy-on-write (COW) operation for each written block. This means, that every time a block (or a part of it) changes, the DMU allocates a new block and copies the new data into it. The exact procedure and benefit of this solution are described in section 2.2.4.

Pooled Storage Layer

The pooled storage layer contains another important part of ZFS, the storage pool allocator (SPA). As the name suggests, this layer has the task to allocate space, managing pools and process I/O (input/output) operations. SPA is only the name for this region consisting of three main components [Dil10, p. 5]: The ARC, ZIO and VDEV layer.

The ARC

ARC means adaptive replacement cache and acts as a replacement for usual page caches used in other file systems. It stores data based on the DVA, adapts to the workload of issued I/O operations and manages memory in the SPA [Dil10, p. 25].

ZIO

The ZFS I/O (ZIO) layer contains the ZIO pipeline, which consists of several steps used for example for low level read, write and transformation operations. Whereas the ARC works with DVAs, the ZIO translates them into logical addresses on VDEVs. As mentioned above, this layer is responsible for data transformation (compression and encryption) but also for creation and validation of checksums as well as the handling of duplicate data [Dil10, p. 26].

2.2.4. Writing data

The write process in ZFS is designed to ensure full data consistency, which means that there are only correctly written blocks in the currently used Merkle tree on disk. To detect any kind of error, every block pointer has a checksum of the data it holds.

A checksum does not ensure data integrity, but the write mechanism does, which is implemented in the DMU (as described in [Bon+03]). When any part of a block is written, the DMU allocates a new block and fills it with the new data. Therefore the indirect block holding the location of the old block has to be updated too, in order to point to the new block. To not corrupt any checksums of indirect blocks, each one of them has to be copied as well, which "ripples" up to the uberblock. Copying all these blocks is therefore called copy-on-write (COW).

The uberblock is saved in an array of uberblocks where the entry with the highest transaction group number is "active". Instead of updating an existing uberblock, a new one is created in a round robin manner across all VDEVs. Within one atomic operation, a new active uberblock is created, which directly switches to the new tree of indirect blocks. The figure 2.6 illustrates the tree of blocks while some new data is written.

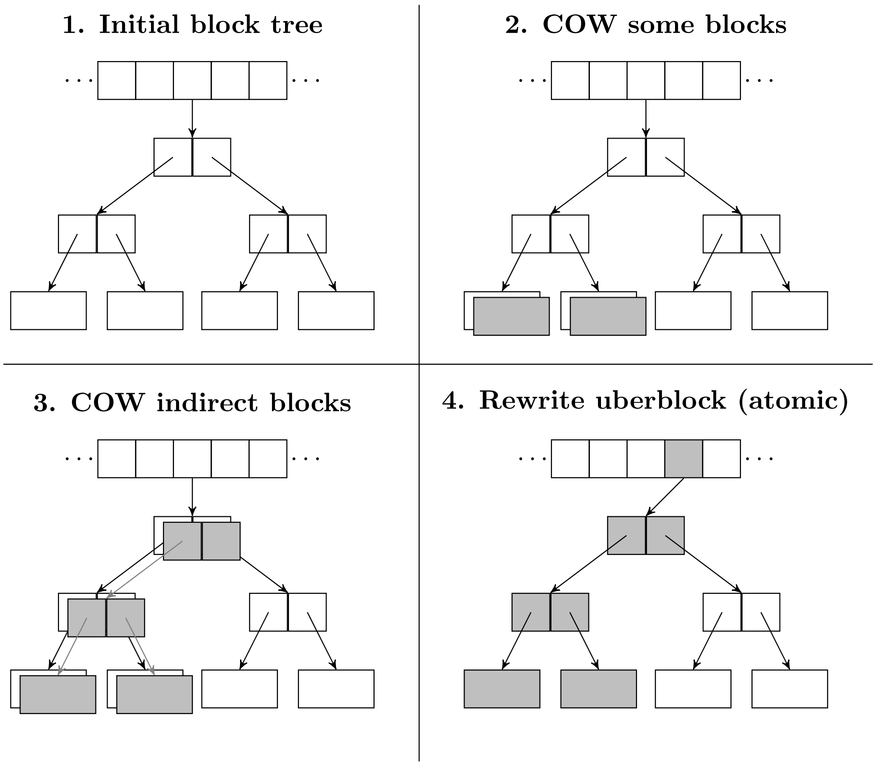


Figure 2.6.: Following the copy-on-write principal creates new blocks before finally overwriting the uberblock in an atomic operation. Based on [Dil10, p. 7].

2.3. Lustre

Lustre is a parallel and distributed file system licensed under the GPL v2. It is mainly used in high performance computing (HPC) clusters, among others at the DKRZ. Lustre is able to handle petabytes of storage data and gigabytes per second of IO throughput, as seen by the new super computer "Mistral" at the DKRZ [DKRb]. Another strength of Lustre is its ability to scale with the amount of servers in a cluster. By adding new storage devices or nodes, the capacity and the I/O performance increases [Ora17].

2.3.1. Architecture

Lustre consists of a whole variety of components where the most important for this thesis are the object storage client (OSC), object storage server (OSS) and object storage target (OST). There are also others like metadata server (MDS), management server (MGS), metadata target (MDT) and management target (MGT). An overview is given by figure 2.7.

The MGT is a storage device storing information about the Lustre file system and is used by the MGS. The MGS provides configuration information about the Lustre file system to clients, which retrieve these information when mounting a drive. Every Lustre component registers itself at the MGS and provides necessary information.

One of these components is the MDS, which serves metadata for Lustre. Besides storing namespace information and usual metadata, like owners and groups (as seen by `stat()`), the MDS determines where files can be stored in Lustre. Clients rely on the MDS, because Lustre is unavailable if it is offline.

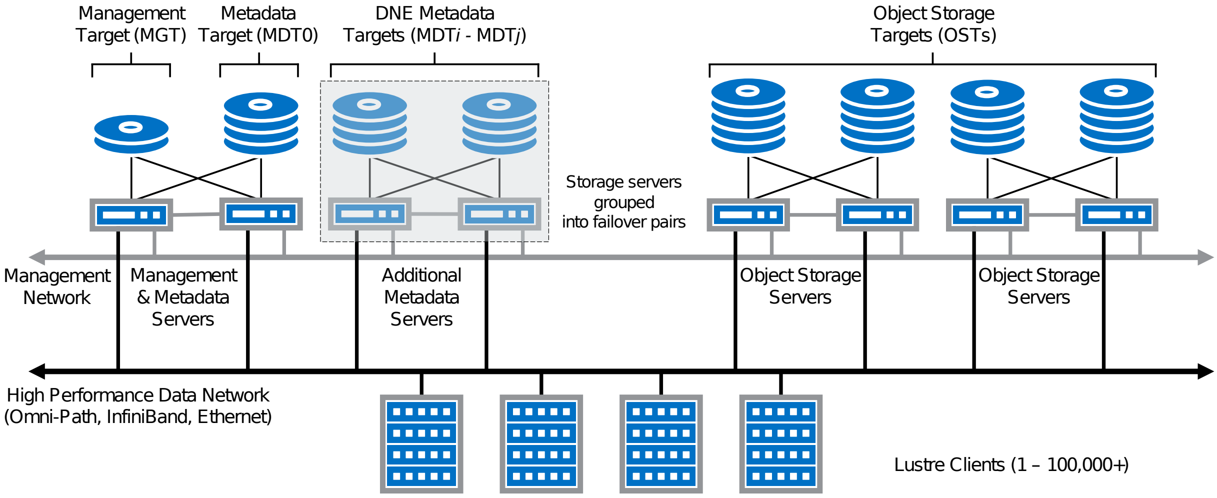

Management Target (MGT)
Metadata Target (MDT0)
DNE Metadata Targets (MDTi - MDTj)
Object Storage Targets (OSTs)
Storage servers grouped into failover pairs
Management Network
Management & Metadata Servers
Additional Metadata Servers
Object Storage Servers
Object Storage Servers
High Performance Data Network (Omni-Path, InfiniBand, Ethernet)
Lustre Clients (1 – 100,000+)

Figure 2.7.: Lustre server architecture [Lus17, p. 3].

The third server component is the OSS, which handles the actual read and write requests. An OSS uses multiple OSTs, which are the devices that store the data. Usually a RAID 6 or similar parity-based RAID levels, like RAID 5 or RAID-Z, are used to form one OST, as seen in figure 2.8.

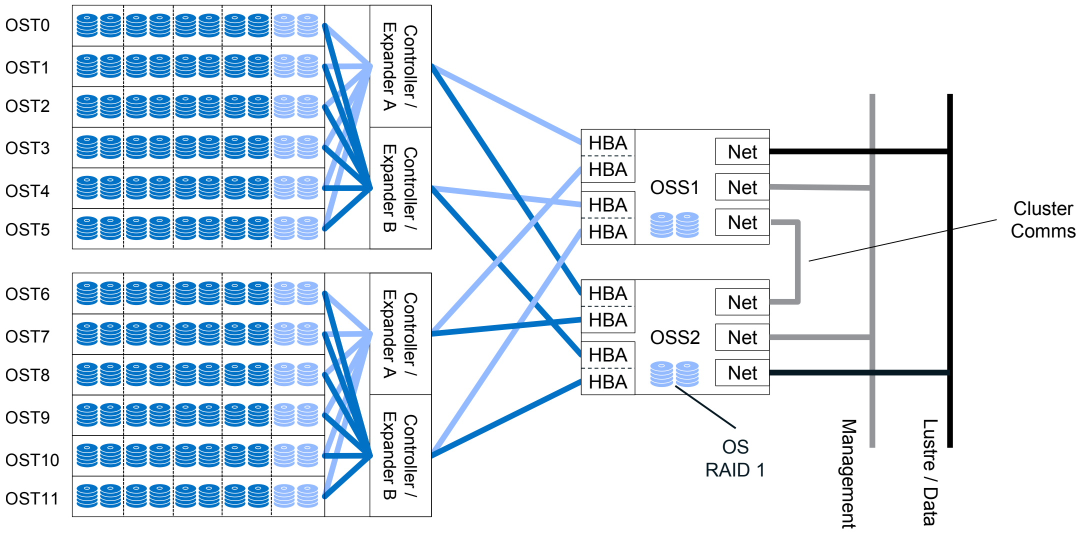


Figure 2.8.: An OST usually use a RAID volume. Here an RAID 6 with eight data and two parity disks is used [Lus17, p. 21].

All these components are used by a Lustre client, which is a kernel-based software and installed on every node in order to mount and use the Lustre file system. Thanks to the implementation of the POSIX standard, a mounted Lustre file system appears to applications like any other file system. The Lustre client is structured as follows: The MGC communicating with the MGS, the MDC with the MDS and the OSC with the OSS. All requests are remote procedure calls (RPCs) specifying the operations that should be performed. For example, when reading a file a RPC is send from the OSC to the MDS to lock a file and to look up the location. The MDS returns the requested metadata containing the OSTs, which hold the file's data [Lus17].

2.3.2. Client side compression

To save space in a transparent, efficient and practical way, data should be compressed on the client side and then sent over the network to be finally stored to disk via ZFS. This also bypasses the possible bottleneck of relatively low network throughput.

A first approach could be to let the user space application use tools like `zip` for compression. Obviously this will work but has a crucial disadvantage: Every application using the data has to know that the data is compressed and what algorithm had been used. Another disadvantage is the lack of support for partial writes, which is useful when only a small part of a large file has changed.

Another solution, which has been implemented by Anna Fuchs, is a client side compression using Lustre itself. This approach cuts the data into stripes and compresses each stripe separately before sending it to the Lustre server [Fuc16]. The server then saves the data using a new API introduced by Niklas Behrmann [Beh17].

In order to be able to use all features of ZFS (especially correct readaheads), the data ZFS receives from Lustre should be treated as if it was compressed by ZFS itself. This approach is presented in chapter 3 below.

2.3.3. Usage of ZFS by Lustre

The OSD component of the Lustre server implements two ways of persisting data on actual disks. The first one is an enhanced version of the ext4 file system called ldiskfs, the second one is ZFS as the back-end file system. Unlike normal user space applications, Lustre does not use the ZPL but directly calls functions from the DMU. This is described in [Sch17] and summarized in the following sections.

Write path

When Lustre uses ZFS to write data, it makes use of the zero-copy approach. In this case zero-copy means that no data is copied between different buffers, which is done by sharing a buffer between Lustre and ZFS. To do so, a buffer from the ARC of ZFS is borrowed, meaning it is created by ZFS but Lustre writes its data into it. It is then directly passed back to ZFS in order to write the data to disk. This approach does not cause an additional creation of a copy and is therefore more performant.

Buffers can be loaned via the `dmu_request_arcbuf` function, which allocates an anonymous buffer from the ARC (called arcbuf). This happens in the function `arc_buf_alloc_impl`, called indirectly by `dmu_request_arcbuf`. Lustre then splits the arcbuf into pages but uses the whole arcbuf for the zero-copy approach when it is assigned back to ZFS.

Lustre has a separate function to commit a write request in which the arcbuf is given back to ZFS. Assigning an arcbuf means that ZFS assigns the content of the arcbuf to a given buffer from the DMU (called dbuf). The dbuf is then marked as dirty, which leads to I/O operations by the synchronization mechanism of ZFS.

Read path

Reading data in Lustre is done by `osd_bufs_get_read`, which calls `dmu_buf_hold_array_by_bonus` to let ZFS fill the empty dbufs passed to that function. Also the amount of dbufs is determined, which used by Lustre to iterate over all dbufs to convert them into pages.

3. Design

This chapter describes the basic functionality for the support of externally transformed data in ZFS. To fully understand the design decisions, an introduction into the ZIO pipeline is given, which is part of the write and read process. After that, the write path is covered by describing the mechanism to mark data as externally transformed. Finally, the read path will be described by introducing flags to prevent ZFS from decompressing the data.

3.1. Compressing data in ZFS

ZFS already has native compression support, which is strongly integrated in the existing I/O infrastructure. Using this to implement an efficient way of storing externally transformed data needs a deeper understanding of this infrastructure that ZFS already has.

3.1.1. I/O Pipeline

I/O operations in ZFS (therefore often referred to as ZIO) are handled by a pipeline called the ZFS I/O pipeline or ZIO pipeline for short. This pipeline executes several steps sequentially before doing the actual I/O operations on a VDEV. Each ZIO is wrapped by a struct called `zio_t`, which contains among others the current pipeline stage and the transformation stack. The pipeline is located in the SPA, which is described in the section *Pooled Storage Layer* in 2.2.3.

Execution of a ZIO is done by calling the function `__zio_execute`. This can happen by a direct call via e.g. `zio_wait`, which executes a synchronous I/O operation, or indirect via a synchronization operation of the SPA. A synchronization of the SPA is done by `spa_sync`, which syncs all datasets via `dsl_pool_sync`.

Each step of the pipeline is specified by the array `zio_pipeline` in `module/zfs/zio.c` containing entries of type `zio_pipe_stage_t`, which is a function pointer taking a `zio_t` as parameter and also returning one[1]. Each step is executed sequentially in `__zio_execute` until `NULL` is returned. Therefore each function has to check if anything should be done, like `zio_encrypt` does as seen in listing 3.1.

```
if (!(zp->zp_encrypt || BP_IS_ENCRYPTED(bp))) {
    BP_SET_CRYPT(bp, B_FALSE);
    return (zio);
}
```

Listing 3.1: The `zio_encrypt` function checks if the data has to be encrypted at all.

[1] Declared in `include/sys/zio.h` with `typedef zio_t *zio_pipe_stage_t(zio_t *zio);`

3.1.2. Compression in ZFS

ZFS supports multiple ways and multiple algorithms for compression. By default, ZFS compresses metadata with its own LZ4 algorithm, which cannot be turned off in OpenZFS[2].

The supported algorithms of ZFS are LZJB, GZIP-N where N is a number from 0 (fastest compression) to 9 (best compression ratio), ZLE and LZ4. All compression properties are specified by the struct `zio_compress` in `include/sys/zio_compress.h`. Each algorithm has its own enum entry, for example `ZIO_COMPRESS_ZLE` for `ZLE` or `ZIO_COMPRESS_GZIP6` for GZIP-6.

The actual algorithm is specified by the table `zio_compress_table`, which is basically a mapping from algorithm to the actual compression/decompression function. Some compression properties do not have an associated algorithm, like `ZIO_COMPRESS_OFF` or `ZIO_COMPRESS_INHERIT`.

Compression, as well as encryption, is a form or transformation and is handled by a stack of transformation steps. Therefore the function `zio_push_transform` adds a transformation to the stack and `zio_pop_transform` removes it from the stack and applies it to the data.

Compressing data

The actual compression of data when writing is done in the `zio_write_compress` function, which calls `zio_compress_data` if the data should be compressed. `zio_compress_data` executes the associated function from `zio_compress_table` and returns the size of the compressed data. Back in `zio_compress_data` this return value is used as `psize` and assigned to the block pointer together with the compression algorithm that had been used. The pipeline stage of compression is now done.

The process and function call stack can be seen in figure 3.1.

Decompressing data

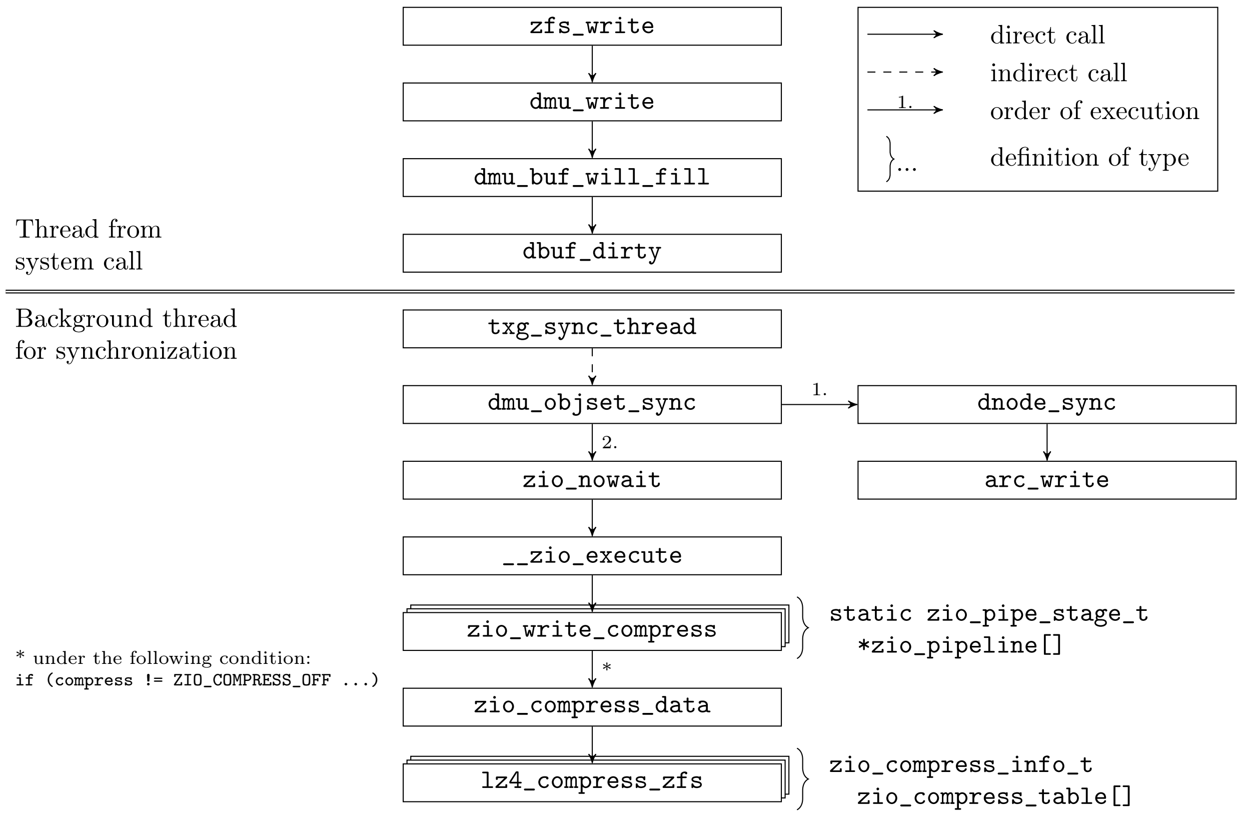
The first step in the pipeline is `zio_read_bp_init`, which prepares the ZIO by settings correct flags and also pushing transformations onto the stack. When reading, compressed data is usually decompressed in `zio_done` by executing `zio_pop_transforms`. This pops operations, goes over every transformation and executes the popped function. When it comes to compression, `zio_decompress` is executed whose control flow ends up in `zio_decompress_data_buf` executing the associated function from the `zio_compress_table`.

The process and function call stack for reading compressed data can be seen in figure 3.2.

3.2. Mark data as externally transformed

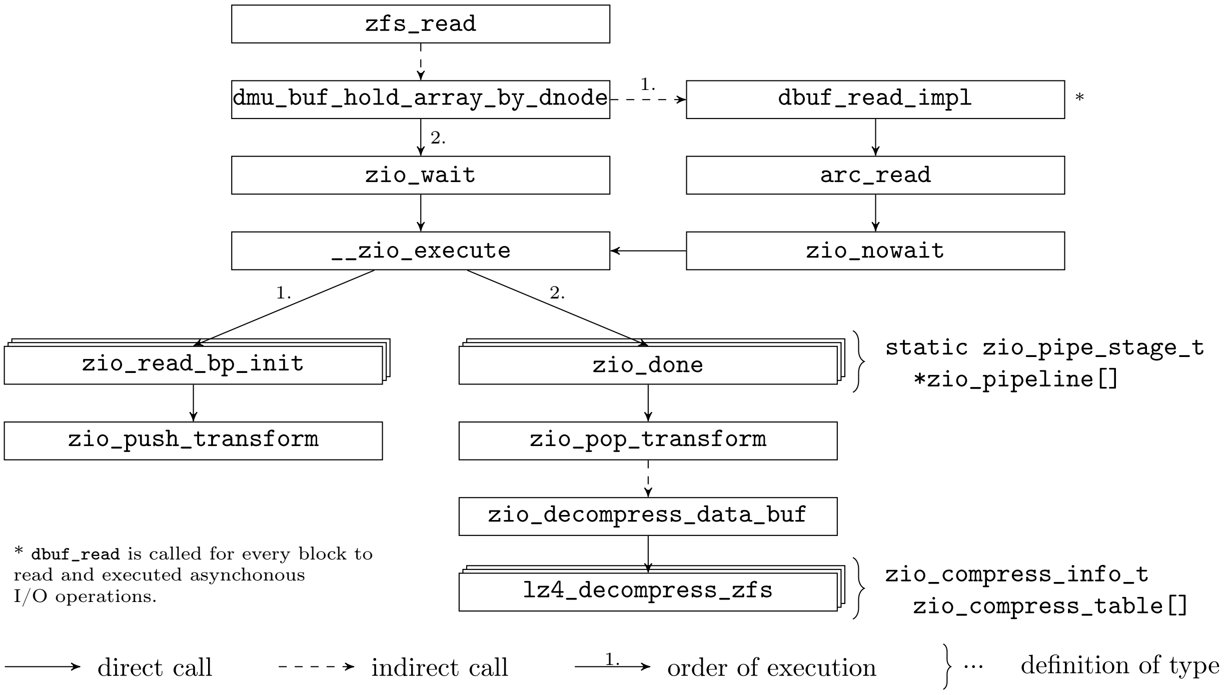
Writes to disk happen either with untransformed data or transformed data (compressed or encrypted). Somehow transformed data always has a physical and a logical size, which are not necessarily equal. When data should be stored compressed, the `zio_write_compress` step in the ZIO pipeline compresses the data.

[2] As mentioned by Tom Caputi in commit `b1d217338a51b025b802ebf6a759f45dcd8e3b4c` on February 21, 2018.



* under the following condition:
`if (compress != ZIO_COMPRESS_OFF ...)`

Figure 3.1.: Code path for compressing data on a write call executed from the ZPL using the LZ4 compression algorithm as an example.



* `dbuf_read` is called for every block to read and executed asynchonous I/O operations.

Figure 3.2.: Code path for decompressing data on a read call executed from the ZPL using the LZ4 compression algorithm as an example.

As described earlier in section 2.3.2, Lustre sends already compressed data to ZFS. In order to store the data Lustre sent to ZFS, the write-process needs to be modified to prevent ZFS from compressing the data again and to mark the saved data as externally compressed. Also the logical and physical sizes must be stored correctly so that efficient readaheads can be done.

The function `zio_write_compress` is called from `__zio_execute` and sets the basic properties of the block pointer. Withing `zio_write_compress` there is already a handling for compressed and uncompressed writes, where externally compressed data can be handled as well.

A first approach using an additional compression mechanism called `ZIO_COMPRESS_EXTERNAL` already exists in the thesis [Beh17] by Niklas Behrmann. This thesis also introduced additional compression flags in the DMU layer structs `dmu_buf`, `dbuf_hold_impl_data` and `dbuf_prefetch_arg` as well as flags for the function `dbuf_read`. The new flags were then passed through several functions to be finally used when writing or reading data. To extend these changes for an efficient and tested solution within ZFS, a refactoring needs to be done in order to add a simple and easy to maintain write and read path.

3.2.1. Efficient readaheads

A compressed block has a physical and logical size where the logical is usually larger then the physical. When writing data without considering the difference between physical and logical size, holes appear on disk. This leads to inefficient readaheads, because these holes can be ignored, but cost resources for reading. An example of this effect can be seen in figure 3.3. However, this is a minor issue, because ZFS is already able to deal with different physical and logical sizes.

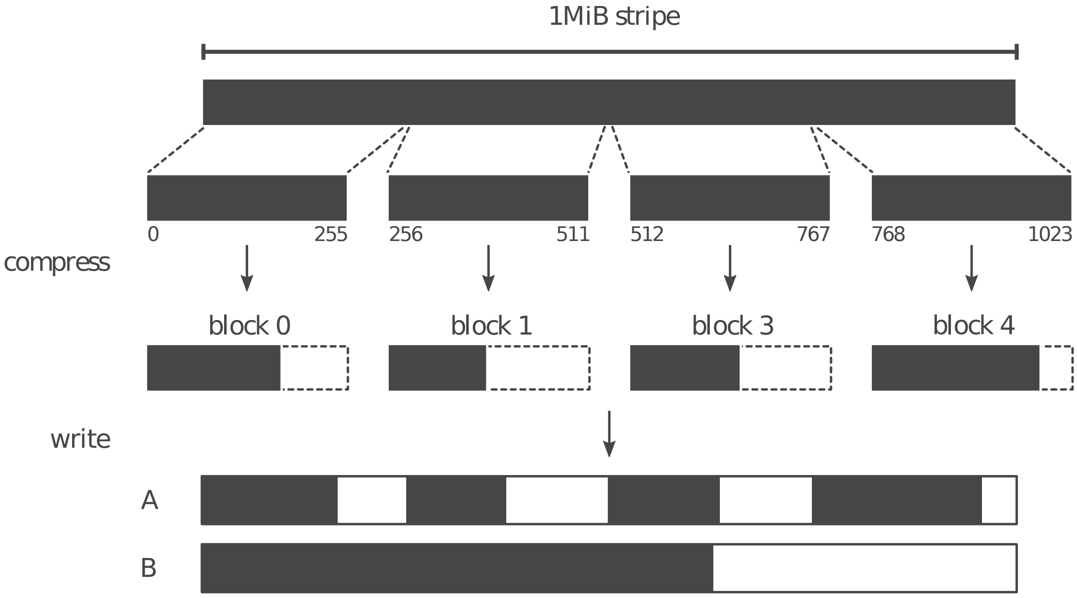


Figure 3.3.: This example shows the effect of writing compressed stripes with and without considering physical and logical size [Fuc16, p. 52].

When the compressed blocks are written correctly and both sizes are stored within the block pointer, efficient readaheads can be made. To do so, ZFS must map the wanted size of the prefetch (logical size) to the corresponding physical size. Only then, the exact amount of compressed data can be read without reading additional, unnecessary data.

Lustre supports next to ZFS also the ext4 variant ldiskfs, which does not support transparent compression. In order to store compressed blocks efficiently, gaps between blocks have to be resolved manually by Lustre before writing data using ldiskfs.

3.3. Prevent decompression

Normally decompression takes place in the ZIO pipeline and is initiated by the function `zio_read_bp_init`, which pushes the decompression step onto the transformation stack. When the I/O operation is done, `zio_done` executes the previously pushed transformation and decompressed the data. This mechanism can be seen in figure 3.2.

Preventing `zio_read_bp_init` from pushing the decompression step onto the stack, won't cause a decompression. The decision if anything is pushed onto the stack or not depends on multiple conditions:

1. The block pointer's compression algorithm must be set.

2. The child ZIO has to be a logical child (and not e.g. a ZIO of a VDEV or deduplicated block).

3. The read must not be a raw compressed read. Raw reads do not transform the data in any form.

Using the last conditions prevents ZFS from decompression, if reading externally compressed data would be a raw read.

The function `dbuf_read_impl` is already setting the `ZIO_FLAG_RAW` flag at the end, when the data should not be decrypted. For the read function `dbuf_read`, which calls `dbuf_read_impl` for uncached buffers, several flags like `DB_RF_NO_DECRYPT` exist[3]. Because the ZIO layer should not depend on DMU specific flags, all `DB_RF_...` flags are then used to set ZIO specific flags and properties.

Using the exact same mechanism for data that should not be decompressed will prevent `zio_read_bp_init` from pushing onto the stack and therefore `zio_done` from decompression. To achieve this, an additional `DB_RF_NO_DECOMPRESS` flag would be able to indicate that data should not be decompressed when reading externally compressed data.

Alternative approach: Implicit prevention of decompression

An alternative approach is the usage of the algorithm set in `zio_write_compress`. When reading the block, this new compression mechanism `ZIO_COMPRESS_EXTERNAL` can either be used to prevent the decompression in `zio_done` or in a function called by `zio_done`.

Not handling `ZIO_COMPRESS_EXTERNAL` in `zio_read_bp_init` causes the call of `zio_push_transform`, which pushes `zio_decompress` onto the stack. Because the `ZIO_COMPRESS_EXTERNAL` mechanism for externally compressed data does not have any decompression function associated, `NULL` will be used in `zio_decompress_data_buf`. However this

[3] Defined in the `include/sys/dbuf.h`

would fail due to an invalid value (in this case NULL) and therefore would return the error EINVAL[4] back to zio_decompress_data.

To not go into this situation, preventing ZFS from popping zio_decompress would be a solution. This requires zio_done to know about zio_decompress and the new ZIO_-COMPRESS_EXTERNAL mechanism. Nevertheless this is not a feasible solution as zio_done should not modify the behavior of the transformation stack. When a transformation should not be executed, it should not be pushed onto the stack.

The second and also not feasible approach is to pop the decompression step from the stack and prevent zio_decompress from doing anything. However this might fail later, when other steps rely on pre- or post-conditions like lsize == psize, because this should be the case after the decompression of data.

3.4. Code paths from Lustre to ZFS

To sum the above design considerations up, this section shows the overall paths for writing and reading data. As Anna Fuchs [Fuc16], Niklas Behrmann [Beh17] and Sven Schmidt [Sch17] showed in their theses, there are new code paths for reading and writing data. The most important function calls are illustrated by figure 3.4.

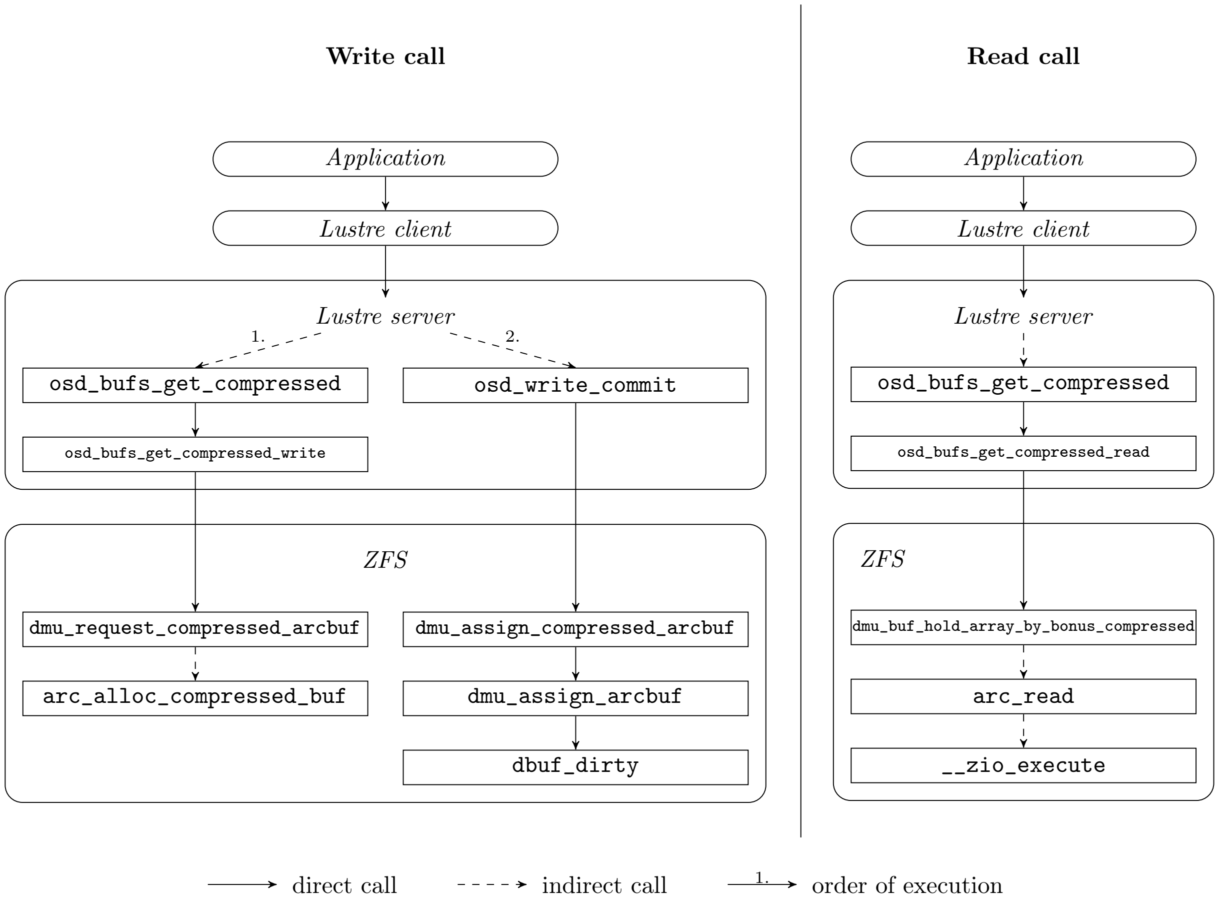


Figure 3.4.: Read and write paths from Lustre into ZFS.

[4] Defined in /usr/include/asm/errno.h with the value 22_{10} (16_{16})

Because the Lustre part is already implemented and uses the new API, the interesting parts for this thesis are the functions and data structures in ZFS. However there needs to be a refactoring before the new design decisions can be implemented. More details and the actual changes are presented in chapter 5.

4. Related work

This chapter shows related work regarding file systems and compression. Two local and distributed file systems with different ways of compression are presented showing alternative approaches in one point or another.

4.1. Local file systems

Even though not all frequently used file systems have support for transparent compression, some important do have this feature. To give a small view into other local file systems and their compression strategies, this section will describe SquashFS and NTFS.

4.1.1. SquashFS

The Linux file system SquashFS is an explicitly read-only file system supporting various compression methods and is used for archival purposes and embedded systems [Squ14]. Aiming to reduce overhead as much as possible, this is done among others by supporting different compression algorithms and using different inode types for each file type (directory, regular file, device file and others). This efficiency is used by many embedded systems and Linux Live CDs like Ubuntu[1].

Next to normal compression of the data, SquashFS also compresses metadata saved in the blocks for inodes. Data is not compressed if the option `-noI` is given or if the compressed data is larger than the original data.

When data is compressed, one of the methods zlib, lz4, lzo or xz is used. Each block is compressed separately and it is from 4KiB up to 1MiB large (default is 128KiB).

4.1.2. NTFS

Not only Linux and UNIX specific file systems, but also the Windows file system NTFS support compression of files [Wik18c]. NTFS uses the LZ77 variant LZNT1 for compression by splitting a file into 16 4KiB large clusters resulting in a 64KiB large chunk. Compression is not available when the cluster size is larger than 4KiB.

Next to compression NTFS also supports efficient writes for sparse files, for example a file with large amounts of zeros in it. Only the relevant content is stored on disk, in this example only the non-zero content.

The free space of compressed files is considered like a sparse file and therefore not written to disk. For example when a 64KiB large file is compressed to 42KiB, five 4KiB clusters are now empty and not written.

[1] As in the file `casper/filesystem.squashfs` packed in the Ubuntu 18.10 image

4.2. Distributed file systems

Distributed file systems often use a local file system underneath, which may support compression. Therefore client-side compression is not always in the focus of the development of distributed file systems. Nonetheless there is some ongoing work in the area of client-side compression as presented by this section.

4.2.1. Lustre

The distributed file system Lustre was not able to compress data until Anna Fuchs presented a mechanism for client-side compression [Fuc16]. In this approach, Lustre does not compress a whole file, but splits the data into stripes. A stripe is then cut into smaller units called chunks, which are then compressed individually. Although there is already a smaller unit called page, this in turn is too small for good compression. Each chunk is then compressed and stored to disk, as illustrated earlier by figure 3.3.

An integration of this client-side compression feature in Lustre with a new API of ZFS was presented by Sven Schmidt [Sch17].

4.2.2. MRC integration in FusionFS

Another distributed file system is FusionFS, which fuses client and storage server into one. Zhao and Raicu integrated a new compression mechanism into FusionFS called multiple reference compression (MRC) [ZR13].

MRC splits the data into blocks and compresses each block. In addition of storing the blocks, equidistant original blocks are stored as well, appended to the decompressed data. These blocks are called references and allow future reads to only decompress requested subsequences of the compressed data. The part between the original data entries is called partition. For example, when the data consists of eight blocks D_0 to D_7 with a partition distance of four, the references would be D_0 and D_4. When data from D_6 is requested, only the second partition with the blocks from D_4 to D_6 needs to be read and decompressed.

Next to compression, the implementation done in [ZR13] introduces additional performance improvements. Data is only written locally on a node, saving time by not transferring data over the network. Read calls from a job requesting data that is stored on another node are just rescheduled, so that the job is executed on the node where the data is stored. Again, this saves a lot of time by not transferring data over the network.

These improvements combined with the MRC mechanism showed a two times faster I/O throughput compared to normal, uncompressed operations [ZR13, p. 10].

5. Implementation

As the design showed, there are two parts to work with: The refactoring and enhancement of the write-path and the implementation of the read-path. First, this chapter describes the refactoring of the write path and shows why it was needed. After that, the changes in the data structures and code paths for writing data are described. The last part deals with the read-path with has been implemented using the new flags introduced in section 3.3.

5.1. Write path

Writing data to ZFS consists of several steps. First a bonus buffer, containing meta information about the requested data, is read from disk using `dmu_bonus_hold`. Then an empty buffer from the ARC is created using `dmu_request_compressed_arcbuf` and the data is filled into this buffer. At the end this filled buffer is written to disk using `dmu_assign_compressed_arcbuf`. How each function had been implemented is described in this section, whereas the read path follows in section 5.2.

5.1.1. Refactoring

Motivation

Before the new features for reading externally compressed blocks had been implemented, a refactoring of the current write path was done. The reason for this is the simplification of the code. In the solution Niklas Behrmann presented, the information if a block is already compressed or not was passed via boolean parameters to several functions. Following this approach would cause many unnecessary changes to the source code introducing a mechanism that is already usable via existing flags.

Data structures

In addition to the introduced parameters, the compression state (compressed or uncompressed) was also added to two data structures. The `dbuf_hold_impl_data_t`[1] structure contained the field `dh_compressed_dbuf` and the `dmu_buf_t` structure `db_compressed`. Both fields were of type `boolean_t`, the exact usage is presented below.

The `dh_compressed_dbuf` flag

This flag was only used once to be passed via the new introduced parameter `compressed_dbuf` to `dbuf_create`. In `dbuf_create` just the `db_compressed` property of the newly created dbuf was set with the value of the new parameter. As listing 5.1 shows, it was

[1] `dbuf_hold_impl_data_t` was renamed to `dbuf_hold_arg_t` in ZFS 0.8.0-rc1

```
static dmu_buf_impl_t *
dbuf_create(dnode_t *dn, uint8_t level, uint64_t blkid,
    dmu_buf_impl_t *parent, blkptr_t *blkptr, boolean_t compressed_dbuf)
{
...
    db->db.db_compressed =  (blkptr != NULL &&
        BP_GET_COMPRESS(blkptr) == ZIO_COMPRESS_EXTERNAL) ?
        TRUE : compressed_dbuf;
...
}
```

Listing 5.1: The new parameter (highlighted in red) was only used for the db_compressed flag

set to TRUE when the block pointer's compression was set to ZIO_COMPRESS_EXTERNAL. If the data was not externally compressed, the value of the newly added parameter of dbuf_create was assigned.

In the case that externally compressed data is not handled by ZFS, FALSE was used as actual parameter by the caller of dbuf_create.

Why this parameter could be removed without loss of functionality is shown below in section *Removing paramters*.

The db_compressed flag

Removing this flag was a larger part of the refactoring because it was more often used. For the write-path the main impact was in the functions dmu_sync and __dbuf_hold_impl[2].

The dmu_sync function stores a single dbuf on disk and is called in special situations like the period of time while a pool gets exported [Beh17, p. 66].

More important was the __dbuf_hold_impl function that also called dbuf_create passing the value of dh_compressed_dbuf into it.

Removing paramters

Because the __dbuf_hold_impl function was called by dbuf_hold_impl, which in turn was called by a variety of functions, each of them got a new parameter, which was just passed through until it reached __dbuf_hold_impl.

When the dbuf created in __dbuf_hold_impl had the db_compressed flag set to TRUE, the allocation of an ARC buffer was done via arc_alloc_compressed_buf. This allocation function also received ZIO_COMPRESS_EXTERNAL as parameter. However the logic to do so was already implemented in a generalized way in dbuf_hold_copy.

The dbuf_hold_copy function uses the ARC compression property via arc_get_compression to determine if the newly allocated arcbuf should contain compressed data or not. Only using this function and therefore removing the manual logic described above, simplified the code a lot. Because there was no need to pass a boolean value regarding

[2] __dbuf_hold_impl was renamed to dbuf_hold_impl_arg in ZFS 0.8.0-rc1

compression to __dbuf_hold_impl after deletion, most of the new parameters could be removed as well.

5.1.2. ARC flags for RAW I/O

The ARC has not been changed but also contains some very important flags for writing externally compressed data. Because a usage of ZIO_COMPRESS_EXTERNAL, to prevent ZFS from transforming the data, would introduce new code paths, the existing infrastructure of raw I/O operations has been used. As mentioned earlier in section 3.3, a raw operation does not change the data by compression, encryption or any other kind of transformation.

To do a raw read or write, the ARC uses specific flags from the ZIO layer. There are three flags, ZIO_FLAG_RAW_COMPRESS for compression, ZIO_FLAG_RAW_ENCRYPT for encryption and ZIO_FLAG_RAW combining both.

The approach presented below makes use of the ZIO_FLAG_RAW_COMPRESS flag to ensure that the data is not compressed again before writing it to disk.

5.1.3. New compression mechanism

The zio_compress data structure contains all compression mechanism known to ZFS. A mapping from a compression mechanism enumeration value to a specific compression and decompression functions, called zio_compress_table, is used to transform the actual data (as described earlier in section 3.1.2). Because the externally compressed data is not compressed in ZFS but within Lustre (or any other application using this feature), there is no compression and decompression function attached to the ZIO_COMPRESS_EXTERNAL value presented by Niklas Behrmann.

5.1.4. Creating buffers

During and especially after the refactoring, the creation of buffers in ZFS had to be adjusted. Because ZFS now has to rely on properly set flags, which already existed in ZFS, setting these needed to be implemented.

Buffer structure of ZFS

ZFS knows different kinds of buffers. The layers DMU and ARC have their own structures to deal with buffers.

The DMU buffer (or dbuf for short) is of type dmu_buf_t and consists only of the actual data, the size of it, the offset inside the object and the object it is associated with. It is a multi purpose buffer, which for example can contain data read from disk or a whole buffer from the ARC.

This rather small and simple structure is used, among many other places, inside the structure dmu_buf_impl_t. In addition to only holding a dmu_buf_t reference, this larger structure also contains meta information about the dbuf. For example, the indirection level, cache state, block pointer or dnode handle are stored in dmu_buf_impl_t.

The ARC has its own buffer called arc_buf_t (or arcbuf for short), which among other properties consists of the actual data, flags and a header. The header for an arcbuf contains metadata about the actual data. Next to the flags and the DVA, also the physical and logical size are stored in the header.

Creation of ARC buffers

The function `arc_buf_alloc_impl` creates an empty buffer. One parameter contains the header of this ARC buffer, which is created via `arc_hdr_alloc` in the function that called `arc_buf_alloc_impl`. There are several other functions using this allocation function, one of them is `arc_alloc_compressed_buf`, which can handle compression modes other than `ZIO_COMPRESS_OFF`. This function in turn is among others called by `dbuf_hold_copy`, whose usage is described above, and by `arc_loan_compressed_buf`.

The `arc_loan_compressed_buf` function is then called by `dmu_request_compressed_arcbuf`, which is one of the new functions introduced by Niklas Behrmann in his new DMU API for Lustre. The described function calls are illustrated in figure 5.1.

Setting properties

As mentioned in section 5.1.2, the ARC works with flags indicating that a buffer should not be compressed/decompressed or encrypted/decrypted. These I/O operations are called "raw" operations.

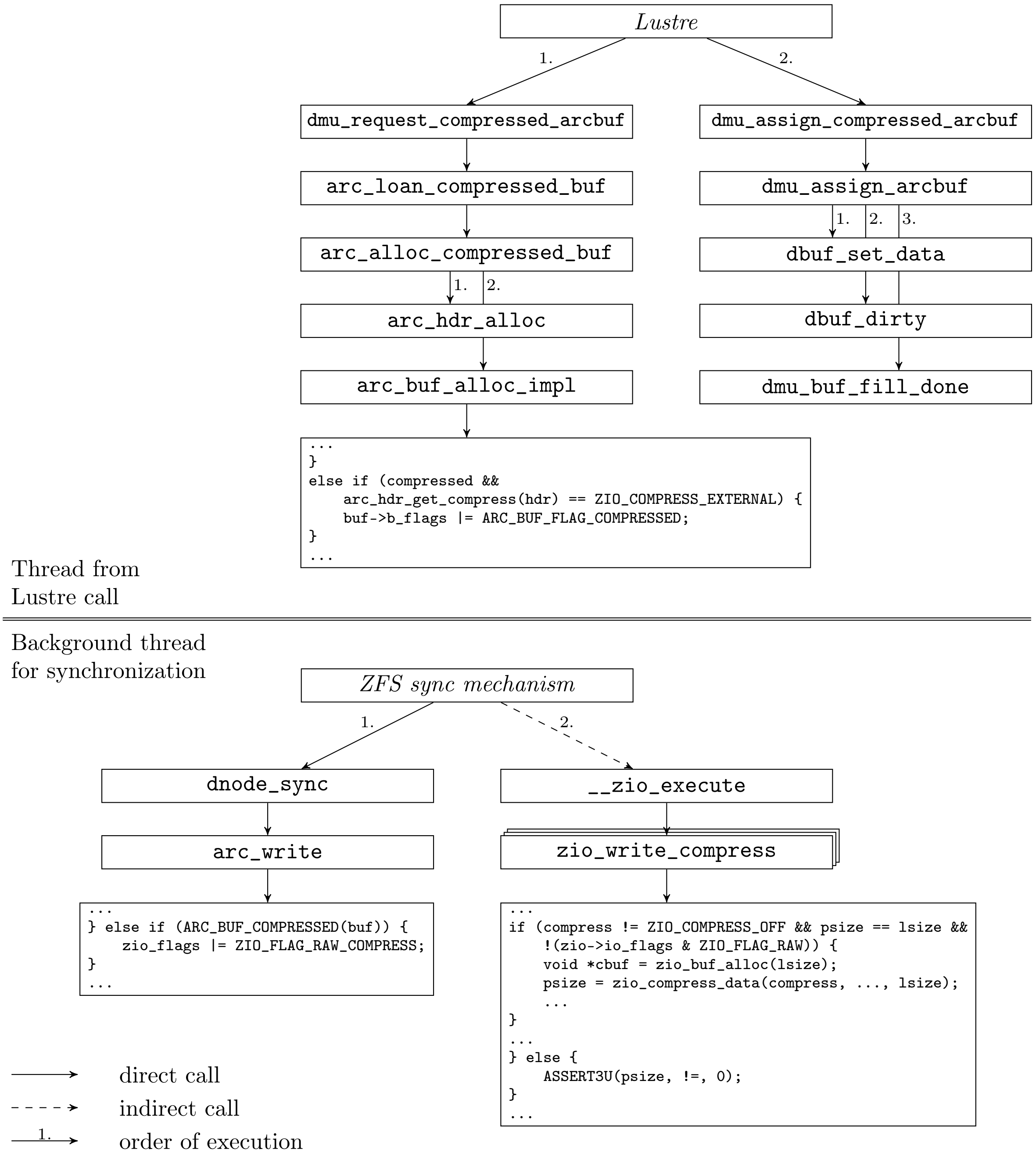
Setting the `ARC_BUF_FLAG_COMPRESSED` flag when allocating an ARC buffer has therefore been added. The allocation function `arc_buf_alloc_impl` sets this flag, if the buffer contains compressed data and if this data is externally compressed. However this flag does not directly cause raw I/O operations, but it is used in `arc_write`, where the corresponding ZIO flag `ZIO_FLAG_RAW_COMPRESS` is set. This will cause a raw I/O operation when the ZIO pipeline starts processing the request. The code execution paths can be seen in figure 5.1.

5.1.5. Assign buffer

After creating the buffer with all properties set correctly, Lustre calls the function `dmu_assign_compressed_arcbuf` introduced by Niklas Behrmann.

This function assigns an arcbuf to a dbuf that should be written to disk. To create a correct dbuf where the actual data should be stored in, a handler dbuf is passed into `dmu_assign_compressed_arcbuf`. This handler dbuf does not necessarily contain the data but can instead contain the metadata of a dnode. Irrespective of the content, this handler dbuf is just used to get the attached dnode we want to write to.

The final step of assigning the dbuf is done in `dbuf_assign_arcbuf` with the most important function calls at its end: `dbuf_set_data`, `dbuf_dirty` and `dmu_buf_fill_done`. Executing the `dbuf_set_data` function follows a zero-copy approach by just setting the arcbuf's data to the data of the dbuf, without copying memory. As shown by figure 3.1, the call of `dbuf_dirty` is one of the last steps in the write process (even though the code path described here differs from figure 3.1). The last call, `dmu_buf_fill_done`, verifies the dbuf, sets the cache-state to `DB_CACHED` and notifies observers waiting for changes on that buffer. The synchronization thread will then write the dirty data to disk as described in section 3.1.1.



```
...
}
else if (compressed &&
    arc_hdr_get_compress(hdr) == ZIO_COMPRESS_EXTERNAL) {
    buf->b_flags |= ARC_BUF_FLAG_COMPRESSED;
}
...
```

```
...
} else if (ARC_BUF_COMPRESSED(buf)) {
    zio_flags |= ZIO_FLAG_RAW_COMPRESS;
}
...
```

```
...
if (compress != ZIO_COMPRESS_OFF && psize == lsize &&
    !(zio->io_flags & ZIO_FLAG_RAW)) {
    void *cbuf = zio_buf_alloc(lsize);
    psize = zio_compress_data(compress, ..., lsize);
    ...
}
...
} else {
    ASSERT3U(psize, !=, 0);
}
...
```

Figure 5.1.: Illustration of the functions setting several flags on a write call from Lustre. The code snippets are simplified part of the actual ZFS code.

5.2. Read path

In contrast to the write path, the structure of the read path is rather simple. This section describes the changed made in order to correctly set the new introduced flags to prevent decompression. Therefore the new code path is presented.

5.2.1. Flags for reading via the DMU

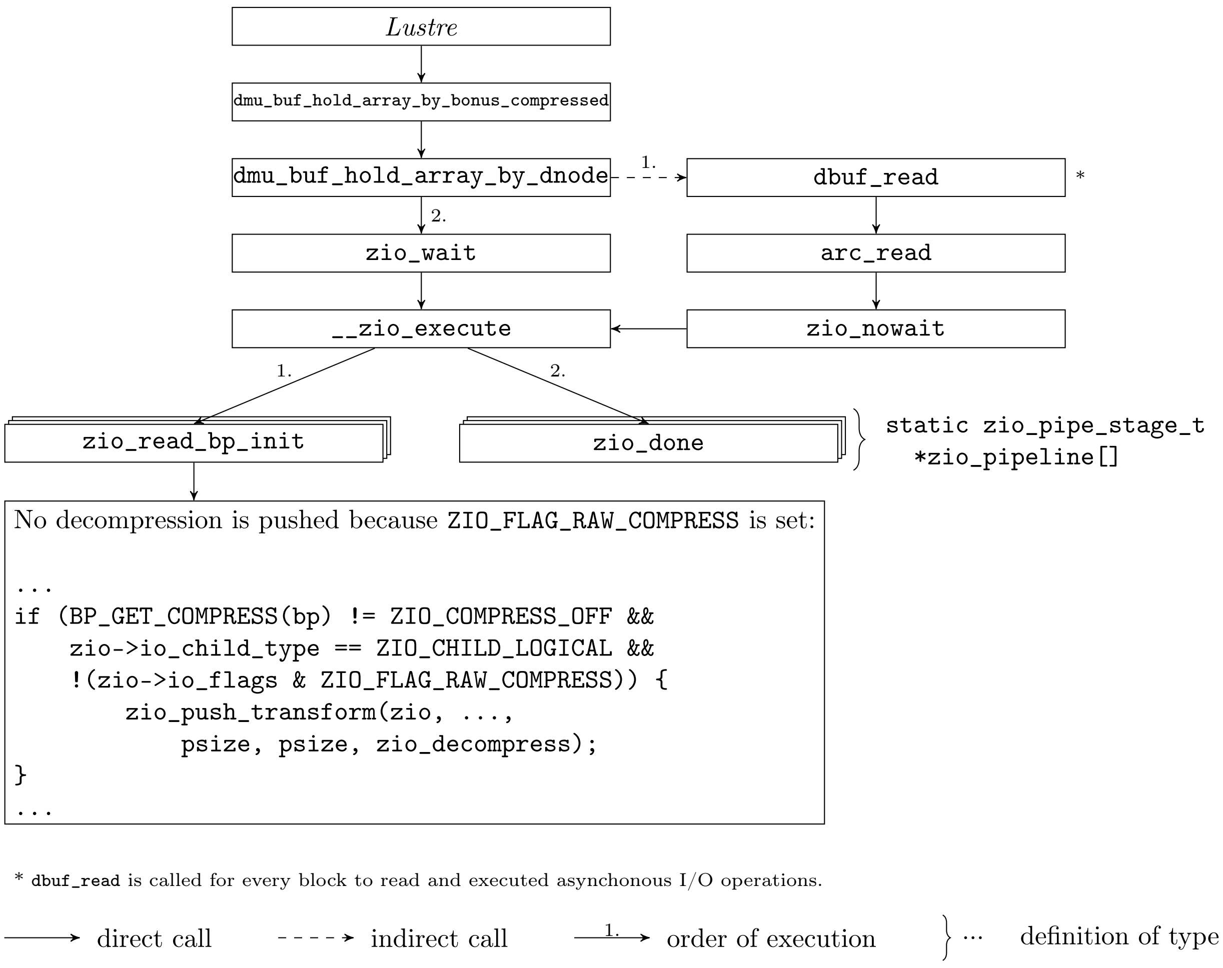
Reading via the DMU makes use of several flags indicating different kinds of procedures that should or should not be executed. One example is the decryption of data, which can be disabled by the `DMU_READ_NO_DECRYPT` flag when it is passed to DMU functions like `dmu_buf_hold`. These `DMU_READ_...` flags are then converted into buffer and I/O specific `DB_RF_...` flags. For decryption this would be the `DB_RF_NO_DECRYPT` flag, which can for example be passed to the `dbuf_read` function.

To use the exact same mechanism, and therefore introduce as little changes as possible, these flags have also been used to prevent ZFS from decompressing the data. The new flags are called `DMU_READ_NO_DECOMPRESS` for the DMU functions and `DB_RF_NO_DECOMPRESS` for `dbuf_read`.

5.2.2. Code path

Reading data using the DMU layer is done by `dmu_buf_hold_array_by_bonus_compressed`, also introduced by Niklas Behrmann. This function requires a handler dbuf (called a "fake" dbuf inside the function) whose only purpose is – as described in section 5.1.5 – to determine the dnode the data should be written to.

More important is the function `dmu_buf_hold_array_by_dnode`, which gets called with the `DMU_READ_NO_DECOMPRESS` flag set. It therefore sets the `DB_RF_NO_DECOMPRESS` flag and issues multiple I/O operations using `dbuf_read` where the flag is passed to. Because the issued I/O operations are all asynchronous, the `dmu_buf_hold_array_by_dnode` function also waits for these operations to finish. After that, the data is set – not copied – to the return parameter. Figure 5.2 illustrates the read-path, which has similarities to the normal read path illustrated by figure 3.2.



No decompression is pushed because ZIO_FLAG_RAW_COMPRESS is set:

```
...
if (BP_GET_COMPRESS(bp) != ZIO_COMPRESS_OFF &&
    zio->io_child_type == ZIO_CHILD_LOGICAL &&
    !(zio->io_flags & ZIO_FLAG_RAW_COMPRESS)) {
        zio_push_transform(zio, ...,
            psize, psize, zio_decompress);
}
...
```

* `dbuf_read` is called for every block to read and executed asynchonous I/O operations.

Figure 5.2.: The read path for externally compressed data that does not push transformations onto the transformation stack.

6. Correctness test and evaluation

This chapter describes the correctness test and performance analysis done for the changes to ZFS presented by this thesis. For testing the correctness of writing and reading data, a new test case had been created which functionality is described in this chapter. An analysis of the performance is done by measuring the execution time of the tested functions. The results are presented at the end of this chapter.

Test environment

All tests ran on a QEMU virtual machine (VM) with an emulated x64 Haswell-noTSX dual core CPU at full 3.4 GHz, 4GB of RAM and a 30GB VirtIO drive. The operating system was a CentOS version `7.5.1804` on a `3.19.0-862.6.3.el7.x86_64` kernel. The code was based on ZFS `0.8.0-rc1` and compiled with GCC version `4.8.5 20150923`.

The VM ran on an Arch Linux computer with the kernel 4.20-arch1-1, an Intel® Xeon® E3-1231 CPU and 16GB of RAM.

6.1. Correctness test

To verify that the changes presented in chapter 5 work as intended, a test has been written that uses the functions that had been changed. In order to test the API and the changes presented by chapter 5, the ztest file was used to create a special test. The internal functionality is described by this section.

6.1.1. ZFS test suite and ztest

The correctness of ZFS is tested by two test environments called `ztest` and ZFS test suite. ZFS' own test suite contains normal bash scripts, which are sequentially executed and perform a variety of operations. This could be increasing the pool size by adding a disk, mounting drives or simple operations like renaming directories. However, these tests do not test API functions of ZFS directly, they are wrapped by the test framework, which sets up all necessary VDEVs, pools and datasets.

When certain functions of ZFS should be tested, the test file `cmd/ztest/ztest.c` is used. As the test framework does it in the ZFS test suite, this file as well uses a set up environment and performs randomized tests but directly executes ZFS functions. Normally these ZFS functions are only available in the kernel space but ZFS can also be compiled into a library (`libzfs`), which then operates in the user space.

6.1.2. Test case

Creating a test case in the ztest file just needs a function fulfilling a simple interface. After registering the function to the array `ztest_info`, the function will be executed periodically.

The test case itself creates data with a reasonable pattern, in this case the repeating word `moin`. After that the functions Lustre uses are called. First the write functions:

1. `dmu_bonus_hold_impl`, with the `DMU_READ_NO_DECOMPRESS` flag passed to it, creates a dbuf with meta information about the object that should be stored

2. `dmu_request_arcbuf` creates an arcbuf that is filled with the data created earlier

3. `dmu_tx_create` creates a new transaction for the object set

4. `dmu_tx_hold_write` assigns the dnode of the created dbuf to the transaction. If the object itself already exists but is not cached, I/O operations are issued to read the dnode from disk.

5. `dmu_assign_arcbuf` assigns the dbuf to the filled arcbuf so that the arcbuf is correctly written to disk

6. `dmu_tx_commit` detaches the transaction and the dnode from each other

To verify that the data is written correctly, the test directly reads it back. Because the actual write process depends on the background thread and synchronization mechanism, the data might not be stored on disk but comes from the cache. The read functions used are:

1. `dmu_bonus_hold_impl` creates the dbuf as mentioned above

2. `dmu_buf_hold_array_by_bonus_compressed` reads the actual data from disk, fills the creates buffer array and returns the number of buffers used for that

The read data is then compared to the data that has been written. If a single bit is different, the test fails, otherwise the buffers are released and the test succeeds.

6.1.3. Reading data manually

The ztest uses files as virtual disks and therefore the data can be read, after successfully running the ztest, by just opening the file and searching for the pattern `moin`. Because the files are big and there might be plenty of them, the ZFS debugging utility `zdb` makes it a bit easier to manually read and verify the data.

To do so, the DVA of the data has to be determined in order to read exactly that block. Because ZFS always builds `zdb` with it, the `zdb` utility is in `bin/zdb`. Executing `zdb` in combination with the filtering utility `grep` prints out every externally compressed entry including meta data of it. To get the necessary information from `zdb`, the `-vvvv` flags are used to increase verbosity (more `v`'s, more verbosity), the `-U <cache>` parameter specifies the cache file used by ztest and the last argument specifies the pool name. The listing 6.1 shows that the `ZIO_COMPRESS_EXTERNAL` was correctly stored, the data is not

```
~/zfs $ ./bin/zdb -vvvv -U /tmp/zpool.cache ztest | grep external
                ztest_write_external_compressed(0)[0] = 20
loading concrete vdev 0, metaslab 14 of 15 ...
objset 5 object 20 level 0 offset 0x0 DVA[0]=<0:18017400:1800> [L0 other uint64[]]
    ↪ sha256 external unencrypted LE contiguous unique single size=1400L/1000P
    ↪ birth=24L/24P fill=1 cksum=6d32f0502059cbdf:fd5f1ca34f1d73ae:3f52b4568626da20:
    ↪ 263ab8b814921b3a
```

Listing 6.1: Using zdb and grep to get the DVA and meta data of the written data. Important parts are highlighted.

encrypted and that the data has the correct size. The ztest used a physical size of 4096 bytes (1000 in hexadecimal, therefore `1000P` in the output) and a logical size of 5120 bytes (1400 in hexadecimal, therefore `1400L` in the output). The DVA of this example is `0:18017400:1800`.

Using `zdb` with `-R <poolname> <vdev>:<offset>:<size>` to specify the DVA and `-U <cache>` to specify the cache file created by ztest, views the actual data. The listing 6.2 uses also `head -n 6` to display only the first six lines.

```
~/zfs $ ./bin/zdb -R ztest 0:18017400:1800 -U /tmp/zpool.cache | head -n 6

0:18017400:1800
          0 1 2 3 4 5 6 7   8 9 a b c d e f   0123456789abcdef
000000:  6d6f696e6d6f696e  6d6f696e6d6f696e  moinmoinmoinmoin
000010:  6d6f696e6d6f696e  6d6f696e6d6f696e  moinmoinmoinmoin
000020:  6d6f696e6d6f696e  6d6f696e6d6f696e  moinmoinmoinmoin
```

Listing 6.2: Using zdb and grep to get the DVA and meta data of the written data. Important parts are highlighted.

6.2. Performance test

Determining the performance is done by measuring the execution time of each function used for writing and reading as listed in section 6.1.2.

Measuring execution times

To compare the functions used for externally compressed data to the non-compressed version of these functions, a second test case has been created. This used the non-compressing functions that are also called by Lustre for uncompressed data.

Because the execution time for each function depends on a variety of factors, 139 test runs have been made to calculate an average execution time. Figure 6.1 shows the execution times of all functions mentioned in section 6.1.2.

The recorded data had some strong outliers, as figure 6.2 shows for `dmu_bonus_hold`. They are so strong that they have a significant impact on the average showed in figure 6.1. A very simple mechanism removed them, which used the deviation of each data point from

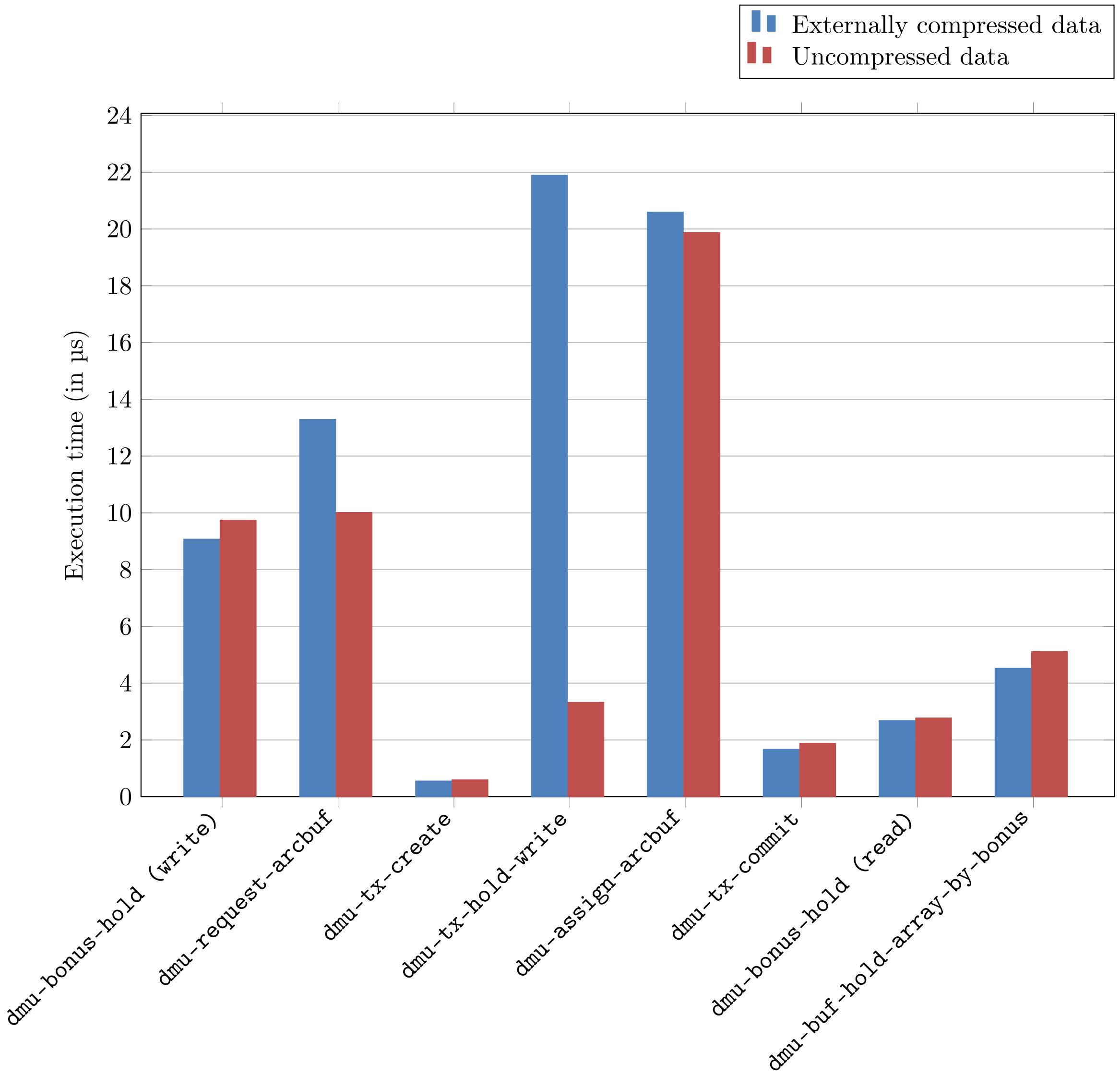


Figure 6.1.: Execution times for several parts of the test case.

the average of the whole data set for the according function. Every data point above a threshold, which was a multiple of the average, had been removed.

Evaluation

As one can clearly see in figure 6.1 the execution time of `dmu_tx_hold_write` is a clear outlier with more then 6 times the execution time of the non-compressing function (21.89µs externally compressed; 3.32µs non-compressing). Because the difference is so high, usual fluctuations on the data are very unlikely. The reason for this significant difference might be a code path that was not covered by the newly introduced conditions and flags. Therefore unnecessary I/O operations, copying of buffers or other instructions might result in a higher execution time.

A second thing that the figure 6.1 shows is that all other functions have rather similar execution times. There is no other function with a difference as high as the one by

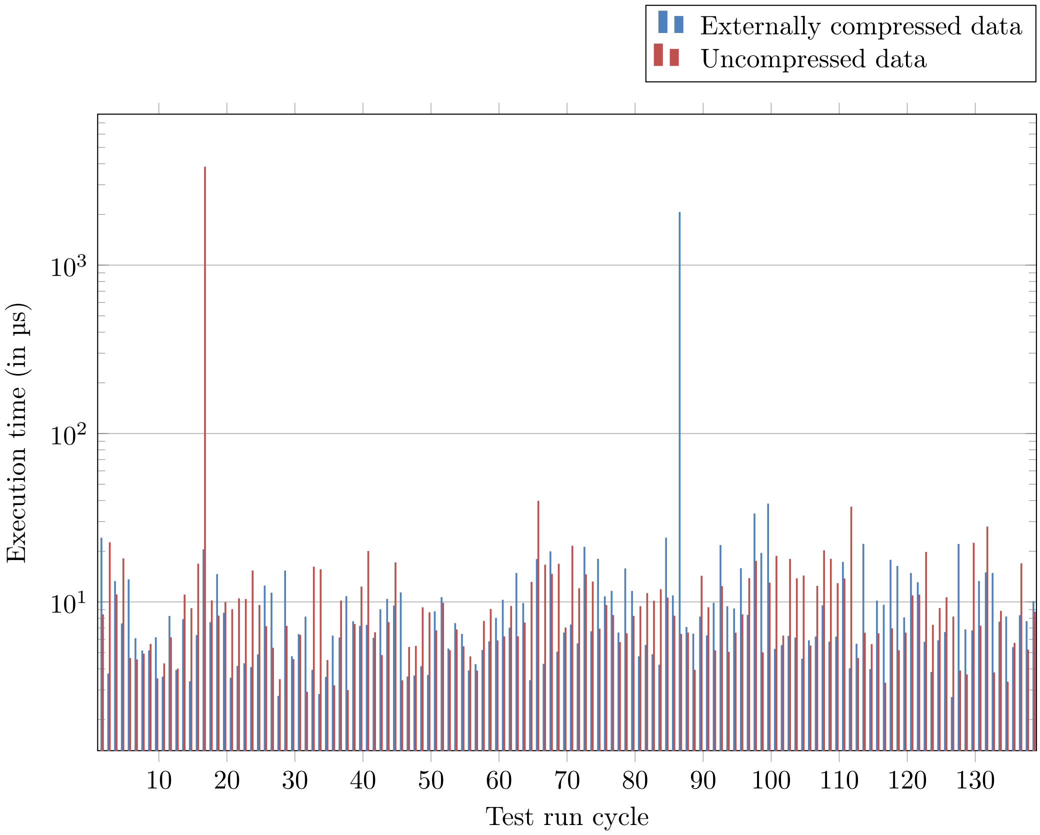


Figure 6.2.: These raw execution times for `dmu_bonus_hold` have two large outliers in it (note the logarithmic scale of the y-axis).

`dmu_tx_hold_write`. Because the changes, which have been made, should be as efficient as possible with as little impact as possible, the lower differences at other functions show that this is the case.

7. Conclusion and future work

7.1. Conclusion

As the IPCC for Lustre project aims to improve existing open source systems, client-side compression inside Lustre and the efficient storage of this compressed data is one part of it. The motivation mentioned that the computational power, network throughput, storage speed and storage capacity increase differently. Closing this building gap between the performance of each component is one reason for client-side data compression with an efficient backend file system, which in this case is ZFS.

7.1.1. Integration of existing theses

As the client-side compression was already implemented by Anna Fuchs in her thesis "Client-Side Data Transformation in Lustre" [Fuc16], some implementation details were described. Her approach splits the data further into stripes, compresses them individually and sends them over the network. Each stripe also contains metadata used for decompression.

A first implementation to support externally compressed data was presented by Niklas Behrmann in his thesis "Support for external data transformation in ZFS" [Beh17], which was used to show the thoughts that had been made and to point out tasks. In his thesis, he presented some new parameters and a new compression mechanism, which is stored on disk to mark data as externally compressed. Also a new API was introduced, which this thesis is based on.

Their both theses were combined by Sven Schmidt in the "Efficient interaction between Lustre and ZFS for compression" [Sch17]. He used the new API added by Niklas Behrmann [Beh17] and used the compression functionality implemented by Anna Fuchs [Fuc16].

7.1.2. This thesis

This thesis presented the design and implementation of an efficient way to write and read externally compressed data in ZFS using additional and yet lightweight code paths. The need for this additional feature in ZFS was presented earlier and made clear that client-side compression inside the Lustre file system needs an efficient storage mechanism in ZFS.

First, the design of a refactoring was described by removing existing parameters and by using more of the existing infrastructure. Secondly, the implementation of the described refactoring as well as the actual new changes and functionalities were shown.

The question if the changes are as efficient as they should be, was answered in the chapter 6 by showing an analysis of the performance. The measurement results of the execution times for the new functions were nearly the same as for the existing and non-compressing functions – with one exception. Even after removing outliers, the `dmu_tx_hold_write` function had a significantly higher execution time when used for externally compressed data. Even though the duration is very small (21.89 µs), it can add up when a lot of small write calls are made. To solve this problem, additional work has to be done.

7.2. Future work

7.2.1. Performance enhancements

As the section 6.2 showed in a performance test, the function `dmu_tx_hold_write` needs a performance enhancement. Unnecessary I/O operations, an unwise use of memory or other unneeded instructions might cause this discrepancy.

7.2.2. Open issues

The described test in section 6.1.2 stores the data correctly, however there are two major issues to solve:

Even though the the data is written correctly, there are certain edge cases where at least the ztest fails. One – maybe only theoretical – case is a physical data size lower than or equal to 512 bytes. In this case the size of the data is rounded up to 512 bytes and therefore stored inside a block pointer as "embedded" data. Apparently this causes ztest to fail and further investigations are needed.

In a special and yet not fully determined state, the `dmu_tx_hold_write` issues an I/O operation to read data from disk. However this read call does not receive the `DMU_READ_NO_DECOMPRESS` flag as it is an internal call of `dmu_read`, issued indirectly by `dmu_tx_hold_write`. Because the data is interpreted as compressed (`!= ZIO_COMPRESS_OFF`), the call of `zio_decompress_data` fails on an `ASSERT` statement. One way to solve this, is the correct setting of the flag `DMU_READ_NO_DECOMPRESS` in the call of `dbuf_read`.

7.2.3. Relaxation of `ZIO_COMPRESS_OFF`

During the development of this thesis, a pull request on the main code base had been made on GitHub[1]. The intention was to get feedback from ZFS developers, who presented alternative solutions regarding the write path of the data, which is currently marked as externally compressed. Using `ZIO_COMPRESS_OFF` instead of the described `ZIO_COMPRESS_EXTERNAL` from section 3.2 needs additional code changes but would not cause problems regarding backward compatibility. New flags for externally compressed data, instead of a new compression mechanism, can be used to run the code path designed earlier.

7.2.4. Decompression inside ZFS

The pull request resulted in another task, which can be implemented in the future. Instead of only marking data as externally compressed, a possible feature is the decompression by ZFS itself. Therefore the data needs to have the same format as if it was generated by ZFS in order to be able to decompress it. This enables a local application to use the data as well as a remote ones like the Lustre client.

[1] `https://github.com/zfsonlinux/zfs/pull/8143`

Bibliography

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

[Bon+03] Jeff Bonwick et al. *The Zettabyte File System*. Tech. rep. 2003. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.184.3704&rep=rep1&type=pdf.

[DCG18] Xavier Delaunay, Aurélie Courtois, and Flavien Gouillon. "Evaluation of lossless and lossy algorithms for the compression of scientific datasets in NetCDF-4 or HDF5 formatted files". In: *Geoscientific Model Development Discussions* 2018 (2018), pp. 1–25. DOI: 10.5194/gmd-2018-250. URL: https://www.geosci-model-dev-discuss.net/gmd-2018-250/.

[DKRa] DKRZ. *Computer History at DKRZ*. URL: https://www.dkrz.de/systems/historie.

[DKRb] DKRZ. *HLRE-3 "Mistral"*. URL: https://www.dkrz.de/systeme/hpc (visited on 11/21/2018).

[DKRc] DKRZ. *Supercomputing and data*. URL: https://www.dkrz.de/mms/pdf/klimadaten/poster/K202037_Infotresen.pdf.

[Deu96a] P. Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. May 1996. URL: https://www.ietf.org/rfc/rfc1951.txt.

[Deu96b] P. Deutsch. *GZIP file format specification version 4.3*. May 1996. URL: https://www.ietf.org/rfc/rfc1952.txt.

[Dil10] Andreas Dilger. *ZFS Features & Concepts TOI*. 2010. URL: http://wiki.lustre.org/images/4/49/Beijing-2010.2-ZFS_overview_3.1_Dilger.pdf (visited on 12/01/2018).

[Dö16] Sebastian Döbel. *Scientific data formats*. May 2016. URL: https://wr.informatik.uni-hamburg.de/_media/teaching/sommersemester_2016/sds16_dobel_data_formates_presentation.pdf (visited on 12/11/2018).

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

[GA17] Jean loup Gailly and Mark Adler. *zlib 1.2.11 Manual - gzip File Access Functions*. Jan. 2017. URL: https://zlib.net/manual.html#Gzip.

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

[Log16] Aleksej Logacjov. “Klimamodellierung und die Entwicklung moderner Computer”. MA thesis. Universität Hamburg, 2016. URL: https://wr.informatik.uni-hamburg.de/_media/teaching/sommersemester_2016/siw-16-seminararbeit.pdf.

[Lud18] Prof. Dr. Thomas Ludwig. *Hochleistungsrechnen - Vorlesung im Wintersemester 2018/19*. Oct. 2018. URL: https://files.wr.informatik.uni-hamburg.de/s/GtwRaaRT3wMYPbA/download?path=%2F&files=hr-1819.pdf (visited on 01/12/2019).

[Lus17] Lustre. *Introduction to Lustre Architecture*. Oct. 2017. URL: http://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf (visited on 11/21/2018).

[Lyn08] Peter Lynch. “The origins of computer weather prediction and climate modeling”. In: *J. Comput. Phys.* 227.7 (2008), pp. 3431–3444. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2007.02.034. URL: https://doi.org/10.1016/j.jcp.2007.02.034.

[Mic06] Sun Microsystems. *ZFS On-Disk Specification*. Sun Microsystems, 2006. URL: http://www.giis.co.in/Zfs_ondiskformat.pdf (visited on 12/01/2018).

[Mü08] Jens Müller. *Data Compression LZ77*. Nov. 2008. URL: http://jens.jm-s.de/comp/LZ77-JensMueller.pdf (visited on 12/13/2018).

[Ope13] OpenZFS. *Documentation/Administrative Commands*. Sept. 2013. URL: http://open-zfs.org/wiki/Documentation/Administrative_Commands (visited on 12/01/2018).

[Ora13] Oracle. *ZFS Source Tour*. Mar. 2013. URL: https://web.archive.org/web/20130316073014/http://hub.opensolaris.org/bin/view/Community+Group+zfs/source (visited on 11/25/2018).

[Ora17] Intel Oracle. *Lustre Software Release 2.x Operations Manual*. 2017. URL: http://doc.lustre.org/lustre_manual.pdf.

[SS13] S. Sarika and S. Srilali. “Improved Run Length Encoding Scheme For Efficient Compression Data Rate”. In: 2013.

[Sch17] Sven Schmidt. *Efficient interaction between Lustre and ZFS for compression*. Online https://wr.informatik.uni-hamburg.de/_media/research:theses:sven_schmidt_efficient_interaction_between_lustre_and_zfs_for_compression.pdf. Bachelor’s Thesis. Aug. 2017.

[Sha10] Mamta Sharma. “Compression Using Huffman Coding”. In: *IJCSNS International Journal of Computer Science and Network Security* 10.5 (May 2010), pp. 133–141. URL: http://paper.ijcsns.org/07_book/201005/20100520.pdf.

[Shi+18] Peng Shi et al. “A knowledge-embedded lossless image compressing method for high- throughput corrosion experiment”. In: *International Journal of Distributed Sensor Networks* 14 (2018). URL: https://journals.sagepub.com/doi/full/10.1177/1550147717750374.

[Squ14] SquashFS. *Documentation suqashfs.txt.* From Linux Kernel "Documentation/-filesystems/squashfs.txt", commit 62421645bb702c077ee5a462815525106cb53bcf. Nov. 2014. URL: https://github.com/torvalds/linux/blob/master/Documentation/filesystems/squashfs.txt (visited on 01/07/2019).

[Wik18a] Wikipedia. *LZ77 and LZ78.* Dec. 2018. URL: https://en.wikipedia.org/wiki/LZ77_and_LZ78 (visited on 12/12/2018).

[Wik18b] Wikipedia. *Lossy compression.* Dec. 2018. URL: https://en.wikipedia.org/wiki/Lossy_compression#Methods (visited on 12/10/2018).

[Wik18c] Wikipedia. *NTFS.* Dec. 2018. URL: https://en.wikipedia.org/wiki/NTFS (visited on 01/07/2019).

[Wik19] Wikipedia. *Merkle tree.* Jan. 2019. URL: https://en.wikipedia.org/wiki/Merkle_tree (visited on 01/04/2019).

[ZR13] Dongfang Zhao and Ioan Raicu. “Exploring Data Compression in Distributed File Systems”. In: (Jan. 2013). URL: https://pdfs.semanticscholar.org/d04d/951536299397c66abb08f157b5f36ae2558a.pdf.

List of Listings

List of Figures

List of Tables

List of Abbreviations

API Application programming interface. 21, 29, 32, 36, 47

ARC Adaptive replacement cache. 18, 21, 33–35

arcbuf ARC buffer. 21, 35, 42

CDDL Common Development and Distribution License. 13

CentOS Community Enterprise Operating System. 41

COW Copy-on-write. 13, 18

CPU Central processing unit. 41

dbuf DMU buffer. 21, 35

DKRZ Deutsches Klimarechenzentrum. 3, 7, 19

DMU Data management unit. 17, 21, 26, 27, 35, 38

dnode DMU node. 14, 35, 36, 38, 42

DVA Data virtual address. 14, 15, 18, 36, 42

ENIAC Electronic Numerical Integrator and Computer. 7

ext Extended file system. 14

ext4 Fourth extended file system. 21, 27

FLOPS Floating-point operations per second. 8

GCC GNU Compiler Collection. 41

GDB GNU debugger. 60

GPL GNU General Public License. 13, 19

HPC High performance computing. 19

I/O Input/output. 18, 23, 36, 48

IPCC Intel® Parallel Computing Centers. 3, 8, 47

JPEG Joint Photographic Experts Group. 10

LZ4 Lempel-Ziv 4. 12, 24

LZ77 Lempel-Ziv 77. 12, 31

LZ78 Lempel-Ziv 78. 12

LZJB Lempel-Ziv-Jeff-Bonwick. 24

LZNT1 Lempel-Ziv New Technology I. 31

MANIAC I Mathematical Analyzer Numerical Integrator And Computer Model I. 7

MDS Metadata server. 19

MDT Metadata target. 19

MGS Management server. 19

MGT Management target. 19

MOS Meta object set. 14

MP3 MPEG-2 Audio Layer III. 10

MPEG Moving Picture Experts Group. 10

MRC Multiple reference compression. 32

NTFS New Technology File System. 31

OSC Object storage client. 19

OSD Objest storage device. 21

OSS Object storage server. 19

OST Objest storage target. 19

PNG Portable network graphic. 12

POSIX Portable Operating System Interface. 13, 20

QEMU Quick emulator. 41

RAID Redundant array of independant disks. 13, 15, 20

RAM Random access memory. 41

RLE Run length encoding. 10, 11

RPC Remote procedure call. 20

RPM Red Hat package manager. 59

SPA Storage pool allocator. 17, 18, 23

SquashFS A read only Linux file system. 13, 31

syscall System call. 17

UFS UNIX file system. 14

VDEV Virtual device. 15, 18, 23, 27, 41

VFS Virtual file system. 17

VirtIO Virtual I/O. 41

VM Virtual machine. 41

zdb ZFS debugger. 42

ZIO ZFS I/O. 18, 23, 24, 27

ZLE Zero length encoding. 11, 24

ZPL ZFS POSIX layer. 17

ZVOL ZFS volume. 17

Appendices

A. Build, test and debug ZFS

Requirements

The following the commands and scripts of this chapter were used on hard- and software described in the section *Test environment* in chapter 6 and require a working environment with all packages installed as recommended by the ZFS wiki[1].

Building

Full build

A full build of ZFS consists of three steps:

```
./autogen.sh
./configure.sh
make -j$(nproc)
```

The `-j#` parameter specifies the amount of threads used by `make`. It is not necessarily needed to build and install packages like RPM packages.

Build with assertions

Building ZFS does not include the evaluation of assertions. All `ASSERT` statements (or similar) are ignored. Adding the parameter `-enable-debug` to the configuration script above enables assertions.

```
./configure.sh --enable-debug
```

Testing

Ztest

The ztest script is located in `./cmd/ztest/ztest.c` and directly calls ZFS functions to test them. Most of the parameters and properties are randomized (for example compression settings or the size of objects).

Build

A simple usage of `make` build ztest:

```
cd cmd/ztest
make
```

[1] `https://github.com/zfsonlinux/zfs/wiki/Building-ZFS`

Run

Running ztest has some default parameters, which are among others the usage of 23 threads and a total run time of 300 seconds. Overriding some parameters might be useful:

```
./cmd/ztest/ztest -VVVVVVVV -t 1 -T 60
```

This prints a lot of output, only uses one thread (which prevents the output from being mixed up) and just runs for 60 seconds (which is enough for most use cases).

ZFS test suite

The ZFS test suite is a collection of scripts testing all kinds of functions. The scripts are located in `./tests/zfs-tests/tests` and have the file extension `.ksh`. However the executable to run the tests is `./scripts/zfs-test.sh`. All tests are registered by a so called runfile, which is a list of `.ksh` files that should be executed. The default is the `./tests/runfiles/linux.run` and can be changed by `-r <runfile>`.

```
./scripts/zfs-test.sh -vx -r foo.run
```

The `-v` flag increases verbosity, `-x` removes files from previous runs and `-r foo.run` specifies the custom run file `foo.run`. The default `linux.run` runfile might take several hours to complete all tests. Therefore a custom run file with the most important tests might be useful.

Debugging

Using a debugger

The first two build steps create all makefiles used to build ZFS. These makefiles specify flags with `CFLAGS = -g -O2` which are adding debugging information via the `-g` flag. However, the optimization `-O2` might cause problems due to heavy usage of compiler optimizations.

Using GDB is possible via executables like ztest.

```
cd ./cmd/ztest
make
./ztest
```

The execution of ztest can be aborted immediately, it was only executed to create the `./cmd/ztest/.libs/lt-ztest` file, which is the actual binary that can be debugged.

```
gdb .libs/lt-ztest
```

Depending on the environment, GDB might need to be executed as root via `sudo`. When GDB has been started, normal breakpoints can be set:

```
b 1234
b ../../module/zfs/dmu.c:1234
```

Even though this might work for most cases, there are many situations during the development of this thesis were GDB was not usable for proper debugging. However, the error was probably not on GDB's side.

Using print functions

Because ztest is running in the user space, the normal `printf` function can be used.

```
#ifndef _KERNEL
    printf("i = %d\n", i);
#endif
```

Writing the check for the `_KERNEL` definition over and over again can be very annoying, but a simple definition of a logging makro helps.

```
#ifdef _KERNEL
    #define hlog(fmt, ...) printk(KERN_DEBUG "LOG %s - " fmt \
    "\n", __FUNCTION__, ##__VA_ARGS__)
#else
    #define dlog(fmt, ...) printf("DEBUG LOG %s - " fmt "\n", \
    __func__, ##__VA_ARGS__)
#endif
```

When running ZFS in the kernel, the normal `printk` can also be used. The output is available via `dmesg`.

```
#ifdef _KERNEL
    printk(KERN_INFO "i = %d\n", i);
#endif
```

Another way is ZFS' own debug messaging mechanism with the function `dprintf`. This must first be enabled and in order to use it a file from the `/proc/` file system must be read.

```
echo 1 > /sys/module/zfs/parameters/zfs_dbgmsg_enable
...
cat /proc/spl/kstat/zfs/dbgmsg
```

B. Measurement data

From figure 6.1

This is the raw data in µs for figure 6.1. The interesting value for the function `dmu_tx_hold_write` is highlighted.

	Externally compressed (µs)	Not compressed (µs)
`dmu_bonus_hold`	9,07	9,74
`dmu_request_arcbuf`	13,29	10,01
`dmu_tx_create`	0,55	0,59
`dmu_tx_hold_write`	**21,89**	**3,32**
`dmu_assign_arcbuf`	20,59	19,87
`dmu_tx_commit`	1,67	1,88
`dmu_bonus_hold`-read	2,68	2,77
`dmu_buf_hold_array_by_bonus`	4,52	5,11

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Software System Entwicklung 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.

Hamburg, den 16. Januar 2019

Hauke Stieler

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, den 16. Januar 2019

Hauke Stieler