MASTERARBEIT

Adaptive Compression for the Zettabyte File System

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik

Autor:	Florian Ehmke
Studiengang:	Informatik
Matrikelnummer:	6053142
E-Mail:	8ehmke@informatik.uni-hamburg.de
Erstgutachter:	Prof. Dr. Thomas Ludwig
Zweitgutachter:	Prof. Dr. Norbert Ritter
Betreuer:	Michael Kuhn

Pinneberg, 24. Februar 2015

Abstract

Although many file systems nowadays support compression, lots of data is still written to disks uncompressed. The reason for this is the overhead created when compressing the data, a CPU-intensive task. Storing uncompressed data is expensive as it requires more disks which have to be purchased and subsequently consume more energy. Recent advances in compression algorithms yielded compression algorithms that meet all requirements for a compression-by-default scenario (LZ4, LZJB). The new algorithms are so fast, that it is indeed faster to compress-and-write than to just write data uncompressed. However, algorithms such as gzip still yield much higher compression ratios at the cost of a higher overhead.

In many use cases the compression speed is not as important as saving disk space. On an archive used for backups the (de-)compression speed does not matter as much as in a folder where some calculation stores intermediate results which will be used again in the next iteration of the calculation. Furthermore, algorithms may perform differently when compressing different data. The perfect solution would know what the user wants and choose the best algorithm for every file individually.

The Zettabyte File System (ZFS) is a modern file system with built-in compression support. It supports four different compression algorithms by default (LZ4, LZJB, gzip and ZLE). ZFS already offers some flexibility regarding compression as different algorithms can be selected for different datasets (mountable, nested file systems).

The major purpose of this thesis is to demonstrate how adaptive compression in the file system can be used to benefit from strong compression algorithms like gzip while avoiding, if possible, the performance penalties it brings along.

Therefore, in the course of this thesis ZFS's compression capabilities will be extended to allow more flexibility when selecting a compression algorithm. The user will be able to choose a use case for a dataset such as *archive, performance* or *energy*. In addition to that two features will be implemented. The first feature will allow the user to select a compression algorithm for a specific file type and use case. File types will be identified by the extension of the file name. The second feature will regularly test blocks for compressibility with different algorithms. The winning algorithm of that test will be used until the next test is scheduled. Depending on the selected use case, parameters during the tests are weighted differently.

Acknowledgments

First of all, I would like to thank my advisor Michael Kuhn for his help throughout the time of writing this thesis. Whenever I had questions or needed any kind of support he had time for me.

Furthermore, I would like to thank everyone from the research group Scientific Computing. In 2010 I visited the lecture "high-performance computing" from Prof. Dr. Thomas Ludwig which first draw my interest to that topic. Ever since then I attended several other courses and seminars during which I met many nice people.

Contents

Chapter 1

Introduction

This chapter will give a short motivation as to why compression is increasingly relevant today and how it can be used to conserve energy and save money. The following section will discuss the benefits of adaptive compression, where it is used and why it is important. After that the structure and goals of this thesis will be described.

Motivation

Compression reduces the size of data by identifying and removing redundant information. There are two main driving factors that make compression increasingly important.

First of all, the electricity price is steadily increasing. A typical *Hard disk drive* (HDD) nowadays has a capacity of 2 TB and consumes around 5 to 10 W. The average price per 1 kWh in the *European Union* (EU) was 0.2014 € in 2014. Storing data for one year (assuming the drive runs 24 hours per day, as it is the case in data centers) would cost us between 9 and 18 €. Supercomputers or data centers have tens of thousands of HDDs and as such the energy cost of the HDDs alone make up a major portion of the *Total Cost of Ownership* (TCO). The TCO is not only driven up by the electricity cost but also by having to replace broken disks. If we compress our data and subsequently need less disks for the same amount of information, we save at three different spots. First, we need to buy less disks, second, we need to replace less disks and third, the energy cost is lower.

The second driving factor for compression is a big discrepancy in the growth of *Central Processing Unit* (CPU) and HDD speed as well as HDD capacity. The CPU speed and HDD capacity increase at much higher rates than the HDD speed.

Every 10 years the HDD speed increases by a factor of 10 while the HDD capacity increases by a factor of 100 [22, 23]. In the same time CPU speed increased by factor of 500 [14]. This makes writing to disks a bottleneck in many use cases. If we compare the throughput of today's HDDs with that of popular compression algorithms, we can see that some algorithms compress faster than HDDs write. Therefore, we can, to some degree, compensate for the slow throughput of HDDs with the help of compression and a fast CPU.

Name	Ratio	Compression	Decompression
LZ4 (r101)	2.084	422	1820
LZO 2.06	2.106	414	600
QuickLZ 1.5.1b6	2.237	373	420
Snappy 1.1.0	2.091	323	1070
LZF	2.077	270	570
zlib 1.2.8 -1	2.730	65	280
LZ4 HC (r101)	2.720	25	2080
zlib 1.2.8 -6	3.099	21	300

Table 1.1: Ratio, compression and decompression speed (in MB/s) of various compression algorithms on an *Intel Core i5-3340M* with $2.7\,\text{GHz}$ [10].

In Table 1.1 we can see that LZ4, for example, almost reaches 3 times the throughput of a current HDD. Assuming we want to write $1\,\text{GB}$ of data, compressing it first with LZ4 would take $\frac{1024}{422} = 2.42$ seconds. Writing the remaining $\frac{1024}{2.084} = 491\,\text{MB}$ of data at a speed of $150\,\text{MB/s}$ would last another $3.3\,\text{s}$. Writing the $1024\,\text{MB}$ uncompressed would take over one second longer (assuming we reach the compression ratio of 2.084 shown in the table). In reality, compression and writing to disk would not happen sequentially, but instead in parallel, and the difference would be even bigger in favour of LZ4.

Adaptiveness

Table 1.1 also shows us that some algorithms (zlib, LZ4 HC) offer higher compression ratios at the expense of a significantly slower speed. If data is intended to be stored for long amounts of time, the further reduced size conserves energy and thus saves money (if less drives have to be used). But it is not guaranteed that the compression is worthwhile as some data is simply incompressible. When confronted with incompressible data, some algorithms perform significantly worse than others.

Hence the difficult part is to decide when to use which algorithm. This depends on the layer in which compression is conducted. It can be done manually, in applications, during transport or in file systems. Not every layer offers the same flexibility in terms of choice. If it is done manually we can choose the data and the tool used to compress it. We can skip some data that we know is incompressible. When we send the data over network we are limited to the implementation of the used protocol. As it is not common to offer a large selection of compression algorithms there are other means to compensate for that.

One way is to use *adaptive* compression that changes the compression algorithm based on the data that is being compressed. This could be done by determining the file type of the data. For data types that are known-to-be incompressible, the compression can be turned off and for other data we could choose the best algorithm, determined by prior tests. Another way would be to continuously measure the effectiveness of the compression and adapting accordingly. Two solutions that offer adaptive compression in the network layer are *OpenVPN*[1]and *rsync*[2].

In file systems, however, so far no adaptive solutions are available. To compensate for that, fast compression algorithms like LZ4 or *Lempel-Ziv-Oberhumer* (LZO) that perform well under almost all circumstances are used. But this also implies that potential is given away at times, since they almost never reach compression ratios that are as good as, for example, zlib's.

Goals

The goal of this thesis is to bring adaptive compression to the file system. The file system that will be used for this is the *Zettabyte File System* (ZFS). ZFS is a modern file system that has built-in compression. The implemented compression algorithms are LZJB, LZ4, ZLE and gzip. If the user chooses an algorithm, it will be used for all subsequent writes. As this includes all kinds of data, incompressible data regularly is part of the data that gets compressed. If an algorithm like gzip is used for this, severe performance problems are the consequence. LZ4 on the other hand performs very well when confronted with incompressible data. As a consequence LZ4 is the algorithm of choice most of the time and gzip does not get used. This gives away potential as gzip frequently reaches much better compression ratios.

In the course of this thesis, ZFS will be extended to allow more flexibility when compressing data. Two main features will be implemented. The first is a file type specific compression mode. In that mode a specific file type is always compressed with the same algorithm. This allows us to test file types for compressibility and set up a list of rules in accordance.

The second feature is adaptive block compression. That mode is used if no prior knowledge about the current data exists. Some blocks of the current file will be tested for compressibility with different algorithms. The algorithm that works best for our current use case will be used for the remaining blocks.

In addition to that, a new property will be added called compression mode. This further extends the flexibility as it allows us to specify a use case. The compression mode will be used in conjunction with the two features and allows us to choose between *archive*, *performance* and *energy*. Depending on the mode, different parameters

[1]`http://wwww.openvpn.net`
[2]`https://rsync.samba.org/`

are more important than others when conducting adaptive block compression. In case of file type specific compression we can specify different algorithms for one file type, one for each mode. The archive mode is intended for long term storage such as backups, where reducing the data size is the most important goal. The performance mode should be used when other CPU-intensive tasks are running whose performance should not be impacted. Lastly, the energy mode represents a tradeoff where we will sacrifice some performance if, in turn, we get a better compression ratio.

Structure

This thesis is structured as follows. In Chapter 2, background to the topics relevant in the course of this thesis will be provided. This includes areas of application for compression, important compression algorithms and ZFS. Chapter 3 will present the design of the new features for ZFS. In Chapter 4, related work, primarily in other areas than file systems, will be discussed. Chapter 5 will cover the implementation of the features designed in Chapter 3. The results and evaluation of this thesis are presented in Chapter 6, and Chapter 7 concludes this work.

Summary

This chapter gave an introduction to the topic of this thesis. Increasing electricity costs and the discrepancy between growth in speed and capacity make compression increasingly important. As compression is CPU-intensive, often adaptive solutions are used to account for special cases. In file systems, this is not the case and as a consequence algorithms that provide a good tradeoff, but not the best compression ratio, are used.

Chapter 2

Background

This chapter will provide the necessary background on topics that are important for this thesis. Section 2.1 will describe four different areas of application for compression. The benefits and drawbacks of conducting compression in the specific area will be discussed as well as how the areas influence each other. In the following Section 2.2 the most important compression algorithms will be introduced and the difference between lossless and lossy compression discussed. Section 2.3 is about ZFS and will present its architecture. The focus lies on the components most important for this thesis.

2.1 Areas of Application for Compression

There is no doubt that compressing data is an important task and thus it is conducted in a variety of different areas. This section gives an overview of the areas of application for compression. Additionally, the difficulties as well as how the areas affect each other will be discussed. The areas, which to some extent can also be understood as layers, in question are user level (e.g. manual compression with tools like *WinRAR*[1]) or application level (data created by applications gets compressed) as well as compression in file systems or transport (data gets compressed before sent over network). The latter two can be found in lower levels and thus can not be controlled as easily as manual compression, something that has to be accounted for when compression gets applied on multiple layers at once. A not so far fetched scenario which touches the compression mechanisms of all four layers would be an application that allows to export data in a compressed format. That compressed data will be sent to a backup server (which has file system compression enabled) over network protected by a *Virtual Private Network* (VPN) which compresses and encrypts the traffic. This illustrates why it is important to ensure that the layers work together well without introducing too much redundancy or inefficiencies.

Encryption and Compression Where data gets compressed encryption is often not far away. Both require considerable amounts of resources to transform the data

[1] `http://wwww.win-rar.com`

into the desired compressed or encrypted format. It is important to know that when combining the two the order is very important. The data has to be compressed first before it will be encrypted, otherwise the compression will be a worthless effort. As a general rule, when data gets encrypted, it will retain the length. That means a chunk of data that only contains zeroes will result in a same-sized chunk of seemingly random encrypted data. That random data is incompressible, the zeroes however can be compressed extraordinarily well. That means, when looking at the big picture, not only compressing (redundantly) multiple times in several layers is an efficiency pitfall, but also encryption when it is conducted at an unfortunate step.

2.1.1 User Level

Compression on user level is perhaps the most widely applied and accepted of all. Today several compression standards are widely used and are either natively supported by the used operating system or applied using available third party applications. The user will usually manually select the files and folders that should be compressed using a tool of his choice. The files will then be placed inside of a compressed archive.

Popular tools used today are (among others) *WinRAR*, *WinZIP*[2], *7-Zip*[3] and *GNU zip* (gzip)[4]. It is common to identify the used tool and algorithm by extending the compressed file with an extension corresponding to the algorithm (`.rar, .zip, .7z, .gz`). Compression on user level is used for a variety of purposes, among them some that at first seem counter-intuitve.

2.1.1.1 Archive

The most obvious reason to compress data is for archive purposes. Storage space is expensive, so having to store less saves money. In this use case compression and decompression speed do not matter as much and one will gladly accept a longer backup or restore operation for the benefits of saved space. Compressed archives however have some drawbacks. They make tasks like incremental backups harder as the application creating the incremental backups has to support the compression algorithm and archive format. One also has to make sure that the tools for decompression are available in the future and also run on newer hardware that might be used for accessing the backups. Corruption of the archive might also endanger the whole archive rather than just a couple of files. Most tools however allow repairing archives or even decompression from partial archives so this should not be an issue.

[2]`http://wwww.winzip.com`
[3]`http://wwww.7-zip.org`
[4]`http://wwww.gzip.org`

2.1.1.2 Sharing

Another reason for compression is to share all kinds of data. There are countless possible ways to share data with another person, where each way has its own quirks. A nuisance still present in many tools of the twenty first century is to allow only one file upload per time. Instead of uploading an entire folder of pictures with one click, each image has to be uploaded at a time, probably even sequentially. A workaround for this is to place files in archives, for example with tools like the old tape archiver *Tar*[5]. It is a logical next step to compress that archive after it has been created. If the size of the archive gets rather large however it could get difficult to share it again. Only one file to upload also means a single point of failure as many transfer protocols have to restart at zero in case of errors. Most tools nowadays allow splitting up the archive into same-sized chunks. This makes big data easier to share while avoiding the hassle of thousands of small files. The only drawbacks of this use case are the need of additional software on the other end, as well as possible cross platform difficulties.

2.1.1.3 Validation

A not-so-obvious advantage of storing files in (compressed) archives is the easier validation of data. It is common to also post a checksum[6] when some kind of archive is published, for example the source code of software. After downloading that archive the user can create a checksum for comparison with the one posted where they downloaded the archive. That comparison can often be done by just looking at the two, if more than one checksum is involved however the need for additional tools arises. This also saves a lot of resources, as computing checksums can be a CPU-intensive task. Of course that happens at the expense of knowledge as a changed checksum only tells us that one or more (if not all) files have changed, but not which ones. For most cases however, that information suffices.

Drawbacks Problems arise when it comes to cross-platform availability of the tools. *WinRAR* for example relies on a proprietary compression standard. Although the application is free of charge for private use the proprietary standard results in *WinRAR* not being available in *Debian*[7] (a stripped down version is available, for all features the non-free repository of *Debian* has to be enabled). Another problem is the amount of competing tools and standards. At some point the user will encounter an unrecognized compression format. If the file extension is correct the right tool should be identified fairly fast, if not, the matter can become problematic. On *Unix*-like systems tools like

[5] `http://wwww.gnu.org/software/tar`

[6] A checksum is a value which allows to verify the integrity of data. If, for example, one bit gets flipped, the checksum will change and that error can be detected.

[7] `http://wwww.debian.org`

Dtrx[8] cope with the difficulty of selecting the right (de-)compression algorithm for a given archive. *Dtrx* stands for "Do the right extraction".

2.1.2 Applications

Compression is also very common for applications that create data in any way. Compared to compression in user level the apparent difference is that the application knows the data. It knows when the data is compressible, when it is incompressible and whether or not it makes sense to compress the data at all. Additionally, the internal application state is known. If there are CPU-heavy calculations happening the application may choose not to compress data to avoid slowing down those calculations. The application may then choose to defer the compression until a later point in time or skip it completely.

Compression can be realized by using external libraries of common compression algorithms or by implementing a custom solution tailored to the specific data. Depending on the choice the created data may not be accessible without the application.

2.1.2.1 Examples

Applications that use compression are, for example, those that get in touch with all kinds of media, be it audio, video or images. Media can be compressed very well and heavy use is made of it across the *World Wide Web* (WWW). The reason for that is that media can be compressed using lossy compression algorithms (see Section 2.2.1.1). Applications that deal with text also make use of compression, with specialized compression algorithms existing for different languages.

2.1.2.2 Databases

Databases are a poor example for applications (or rather a component used by applications) that use compression. That is because they are not generating data, but rather storing what the user wants them to store. This actually entails that the database software does not perfectly know whether or not compression makes sense. *PostgreSQL*[9], for example, wields compression as follows. Rows that contain data types with a variable length, that exceed a certain size, will be compressed in order to fit them into the page size (8 kB). Tuples are not allowed to exceed the page size (there are workarounds however) so this is a necessity. This scheme has two major drawbacks. First of all compressing only tuples (rows) and not redundant data in columns gives away huge potential. Below are three typical log entries of a fictional backup tool to illustrate that case. The **`highlighted`** passages represent data that could be compressed if it was not for the constraint that only data in rows can be compressed.

[8] `http://wwww.brettcsmith.org/2007/dtrx`
[9] `http://wwww.postgresql.org`

```
21-04-14 19:33:27 MyBackupTool:startBackup() "will start backup"
21-04-14 19:33:29 MyBackupTool:doBackup() "backing up xyz..."
21-04-14 19:34:02 MyBackupTool:finishBackup() "backup finished"
```

The second drawback is that a threshold has to be met before compression actually kicks in. As per default that threshold is 2 kB. Other compression methods will attempt to compress no matter the size, so this is expected to further draw down the (theoretical) best compression ratio achievable with this approach.

2.1.3 Transport

Compression of data is a very well established practice in various transport layers. The reason for this is that often it is faster to compress, send and uncompress than to just send uncompressed data. This requires a CPU fast enough to do the compressing and decompressing in reasonable time. Although network bandwidth is steadily increasing, it will probably never be faster to send uncompressed data, given a fast enough CPU, as the performance of CPUs increases at a much higher rate [21]. Additionally, there are further constraints depending on the used network. Wireless networks such as the *Universal Mobile Telecommunications System* (UMTS) raise further constraints as the clients using the network may do so under harsh conditions. The connectivity may be sparse or even interrupted from time to time. In addition transmitting data over a wireless medium draws much more power compared to processing the data on the device [7]. As a consequence it is desirable to send as few bits as possible. Many network solutions have implemented some kind of compression into their protocol, although not all of them are enabled by default.

2.1.3.1 Hypertext Transport Transfer Protocol

Hypertext Transport Transfer Protocol (HTTP) is one of the most important building blocks of the WWW. HTTP is a client-server model that allows request-response communication. The server provides resources such as *Hypertext Markup Language* (HTML) files to the client (response) if they are requested. That communication can be compressed if the compression algorithm of choice is supported on both ends (web server, web browser). If that is not the case, it is sent uncompressed. Given that web servers are often equipped with powerful hardware, whereas the clients may be low end devices such as mobile phones, performing compression is a good measure to increase the effectiveness of the communication.

2.1.3.2 Virtual Private Network

A VPN creates private networks across public networks such as the *Internet*. The main goal is to provide secure network access. This allows, for example, an employee to connect to the company network from a public wireless network without posing a security risk. *OpenVPN* is a VPN solution that allows both encryption and compression of the traffic. An interesting feature is the adaptive compression of all traffic. By default (if compression is turned on) *OpenVPN* will periodically measure the efficiency (in terms of compression ratio) of the compression. If the algorithm detects incompressible data it will turn off the compression until the next check is conducted.

2.1.3.3 Secure Shell

Secure Shell (SSH) is a network protocol for secure data communication. Security is realized by encryption of all traffic. It is best-known for remote shell access to *Unix*-like operating systems. Other than that various file transfer protocols are built on top of SSH. Among them are the well-known *Secure Copy* (SCP) and *Secure File Transfer Protocol* (SFTP). The file synchronization software *rsync* also offers a remote mode that can use SSH. SSH itself already supports compression of the network traffic which is the implementation that both SFTP and SCP use when compression is enabled. On top of that *rsync* implemented a more feature-rich compression system. It allows higher compression ratios (at the cost of CPU utilization) but it also features a skip-compress option that accepts a list of file extensions. If such an extension is encountered during the sync process it will not be compressed. The list can be customized by the user and comes with a list of files known to be incompressible.

2.1.3.4 Multiplayer Games

Multiplayer games often feature complex network solutions as the requirements are highly demanding. In most genres the game state has to be synchronized among clients in real-time to avoid unexpected behaviour. To realize this the data that achieves that synchronization has to be kept as small as possible. This is why it is common to only send delta updates in compressed packets.

2.1.3.5 Video Streams

Streaming videos is very demanding from the network connection. It requires a lot of bandwidth and depending on the source it will take as much of it as it gets. This means that the stream will be affected by other things that might occupy bandwidth and requires a solution that adapts to the current context in order to avoid buffering. A video streaming solution that adapts to different available bandwidths on the fly means that the source has to be available at different bitrates at all times. Therefore the encoding to different bitrates has to be done either preemptively or on demand. Doing so on demand requires an encoder that is capable of outputting multiple bitrates at once and preferably even be able to adapt to the available CPU resources.

2.1.3.6 Voice Chat

In voice chat specialized forms of audio compression are applied to transmit only what is relevant to the human ear. The used compression is different to that used for e.g. music. Speech is much simpler and lot of statistical information is available about he properties of speech. That information is used to compress the audio data as much as possible and transmit only what is needed to retain the intelligibility of the conversation.

Discussion Compression in transport requires highly specialized solutions that are often very resource demanding. No matter whether it is bandwidth or CPU utilization, requiring as much as possible of a resource also creates the need for an adaptive solution to make room for other applications. Other reasons that create the need for adaptiveness are environments or contexts that impose constraints such as mobile networks with varying connectivity or the need for real-time updates. A compressing VPN solution that transfers all kinds data needs to react appropriately when confronted with incompressible data. If this does not happen resources are wasted as data might get compressed more than once without further gains.

2.1.4 File Systems

Conducting compression in the lowest possible software layer, the file system, is a very tempting thing to do. It has several outstanding advantages over the areas of the previous sections but also notable drawbacks.

2.1.4.1 Advantages

The one big advantage of compression in the file system is transparency. To every layer above the file system, compressed files will behave just the same as uncompressed files. They are usable without external tools and report their actual logical file size (as opposed to the physical size). In this manner other tools operating in higher layers will be able to benefit from compression without having to implement it themselves.

2.1.4.2 Problems

Compression in the file system comes with several problems that need to be worked out first. The file system will have to deal with all kinds of data, already compressed files, encrypted files or flat-out incompressible data. This requires a compression algorithm which performs well when confronted with incompressible data.

Another area of concern is performance. Doing additional processing of the files as a file system is subject to stringent requirements as it is not the expected behaviour of a file system to use lots of CPU resources. A suitable algorithm should therefore compress as good as possible while not using too much CPU resources which might disrupt other applications.

Additionally, in a file system the compression algorithms might not work on the whole file, but instead only on the blocks of that file. Typical block sizes are e.g. 128 kB. This gives the compression algorithm a much smaller window to find redundancy and as such it is expected to have worse results compared to compressing in higher layers.

Furthermore, the file system is in a difficult spot as it has almost no knowledge about the current context and the data it writes to disk. It does not know why the data is written, it does not know how long the data will stay and it does not know whether or not it would be acceptable to use more resources to achieve a better compression ratio. By this means it will most of the time make a good effort, but never the best.

2.1.4.3 State of the Art

Many file systems today support transparent compression, among them are *Hierarchical File System Plus* (HFS+), *New Technology File System* (NTFS), ZFS and *B-tree File system* (Btrfs). Other widely used file systems that do not support compression are e.g. *Extend File System* (ext) (all versions) or *XFS*.

The implementations in these file systems allow an all-or-nothing compression of the data. No finer control like per-file type or folder compression are available.

2.2 Differences of Important Compression Algorithms

This section introduces the different compression algorithms important for this thesis. Prior to that, the difference between lossless and lossy algorithms, as well as why lossy ones are not applicable for file systems, will be described.

2.2.1 Lossless and Lossy Compression

The main point when compressing data is always to reduce the size as much as possible. This for a large part depends on the available time for the compression. If given more time, most compression algorithms will be able to squeeze out higher compression ratios. The highest impact on possible compression ratios however has the decision whether or not the compression should be lossless or lossy. A file that has been compressed with a lossless compression algorithm lost no information compared to the original. As a consequence the original file can always be completely restored, bit for bit. Lossless compression algorithms compress by identifying redundant information. As opposed to this lossy compression algorithms compress by removing unnecessary information. What information gets categorized as unnecessary depends on the data to be compressed. The information removed during compression can not be restored, thus lossy compression is irreversible.

2.2.1.1 Usage of Lossy Compression

Lossy compression is very common for all kinds of media. Since it aims to remove unnecessary information the goal is to reduce the size as long as the differences in the resulting file are not noticeable for human senses. Lossless compression of media is also possible, but the difference in compression ratio is significant. Lossless compression of videos for example may achieve a ratio of up to three whereas lossy algorithms reach a ratio between 20 and 200 [20]. Lossy compression algorithms focus on specific kinds of media. There are specialized ones for music, videos and pictures.

2.2.1.2 Usage of Lossless Compression

Other than lossy compression algorithms lossless ones are eligible to compress all kinds of data. Since no information is lost they can be used for file systems or file transport protocols where it is not known beforehand what kind of data gets stored or transferred and whether or not information loss would be acceptable. Nevertheless, there are also lots of lossless compression algorithms specialized for specific file types.

2.2.2 Zero-Length Encoding

Zero-Length Encoding (ZLE) or (more general) *Run-Length Encoding* (RLE) is the simplest form of compression. It is applicable on many different kinds of data and always works on the same principle. The algorithm iterates over the data and looks for sequences of repeating data values. Instead of storing the entire sequence in the compressed form a count and a single data point will be stored.

So, for example, in a picture the algorithm would iterate line by line, pixel by pixel over the entire picture. If ten red pixels occur in a sequence it will store `[10r]`. If the subject is text, it will iterate over the text and count sequences of the same character. When working on bits the procedure is a little different. The algorithm will only store when the bit changes, and not what the actual bit was as that information is already present. A short example:

$$\texttt{1111 0000 1100} \longrightarrow \texttt{4,4,2,2}$$

The algorithm could also be modified to look for repetitions of sequences of characters instead of only single characters. RLE is used today as a preliminary step for other compression algorithms to save other algorithms some work but rarely as a standalone solution.

2.2.2.1 Advantages and Characteristics

The big advantage of RLE is that it only needs one iteration over the entire data for the entire process — both for compression and for decompression. This makes it easy to implement and is also the reason why it is a very fast compression algorithm. In fact it is so simple that the entire implementation of the compression in ZFS is only 22 *Logical Lines of Code* (LLOC) long, the decompression needs another 14 LLOC. If the data consists of constantly changing values, the algorithm could in the worst case increase the size. For that reason only sequences of a certain length (e.g. a minimum of three) will be encoded.

2.2.3 Lempel-Ziv Compression

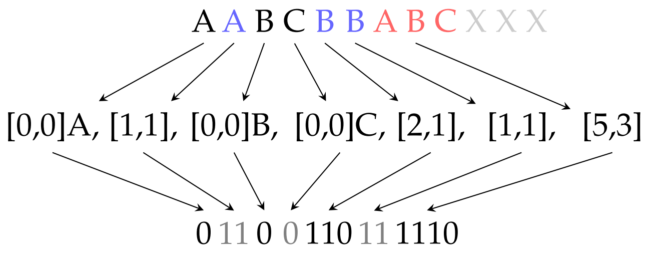


Figure 2.1: *Deflate* compression example. Consists of two steps: LZ77 compression (line 1 → 2) and *Huffman* coding (line 2 → 3). Example taken from [4].

In 1977 Abraham Lempel and Jacob Ziv published a lossless compression algorithm called *Lempel-Ziv Compression* (LZ77) [25]. Many compression algorithms used today are based on LZ77, among them are LZ4 and LZJB which are both available in ZFS. LZ77 searches the text (any byte sequence) that should be compressed for redundant sequences of variable size. The text is searched using a sliding window of a constant size.

In Figure 2.1 such a window can be seen. The first line shows the text sequence with the current window (`AABCBBABC`). The second line shows the output of the LZ77 algorithm. When the algorithm detects a character (or sequence of) that previously occurred inside the sliding window it will output a tuple. That tuple encodes the position and length of the character (sequence). The tuple `[5,3]` can be read like this: "Go back five characters and then read three characters". In order to detect such sequences of multiple characters, the algorithm needs to look ahead. Thus the window can be separated into two areas, a search buffer and a lookahead buffer. If the tuple contains zeroes (`[0,0]`) the character after it is new and therefore encoded as-is. Since the algorithm uses the sliding-window to "look up" things, it can also be thought of as a dictionary. Bigger dictionaries yield more things to look up and therefore better compression ratios, but they also increase the time needed to compress the whole text as more needs to be searched.

Since the size of sliding window, the search buffer and the lookahead buffer can be of arbitrary size the algorithm allows many optimizations. There is no perfect setting for these values as effectiveness for the most part depends on the input data. gzip uses (among other things) LZ77 and allows different "compression levels". These compression levels are in fact the size of the sliding window. Higher levels refer to a bigger window and vice versa.

2.2.3.1 LZJB

The compression algorithm LZJB was developed by Jeff Bonwick who also lead the team that developed ZFS. It was specifically designed for use in ZFS. As compression in a file system is very performance critical it aims for high compression and decompression speeds. LZJB is a variant of *Lempel-Ziv Ross Williams* (LZRW) which in turn is a variant of LZ77.

2.2.3.2 LZ4

LZ4 is another compression algorithm that belongs to the LZ77 family. It was developed by Yann Collet. Similarly to LZJB it was developed with (de-)compression speed in mind. In 2012 LZ4 was added to ZFS as a potential future replacement for LZJB. LZ4 outperforms LZJB significantly both in compression and decompression. It is approximately 50% faster on compressible data and 80% faster during decompression. Additionally, it is over three times faster when operating on incompressible data [9]. The reason for that is a so-called "early abort" undertaken by LZ4 if it detects incompressible data at the beginning of the compression.

2.2.4 Deflate Compression

The *Deflate* compression is actually a combination of two different compression algorithms that work together very well. The first compression is LZ77. The output of LZ77 is then *Huffman* coded. The compression is specified in [2].

Huffman coding is a form of entropy encoding which was developed by David A. Huffman in 1952 [5]. It encodes frequently occurring words to short bit sequences, and uncommon ones to long bit sequences. In the example of Figure 2.1 it can be seen that the tuple `[0,0]`, which occurs the most (three times), gets the shortest bit sequence (`0`) whereas `[5,3]` is encoded to `1110` as it only occurred once.

This secondary compression makes *Deflate* compress considerably better than pure LZ77 but it also requires more CPU time to do so. *Deflate* is, for example, used by gzip.

2.3 The Zettabyte File System

ZFS is a modern file system designed by *Sun Microsystems*. Development started in 2001, the first release was in 2004 as part of the *OpenSolaris* source code. In 2010 *OpenSolaris* was discontinued and further development of ZFS was no longer open source. Also in 2010 *illumos*, an open source derivate of *OpenSolaris*, was founded as its successor which ensured that ZFS continued to exist in the open. ZFS also found its way to other platform as in 2006 development for a *Linux* port started and in 2007 *Apple* started porting it to *Mac OS X* [17].

2.3.1 Characteristics and Classification

ZFS is a general purpose file system, intended to be used on any possible system, be it database servers or desktop computers. The goals during the design of ZFS were to provide simple administration, data integrity and immense capacity [1]. In short, it can be described as a transactional object-based copy-on-write file system. In the following some of the key design principles will be listed.

2.3.1.1 Pooled Storage and Integrated Volume Management

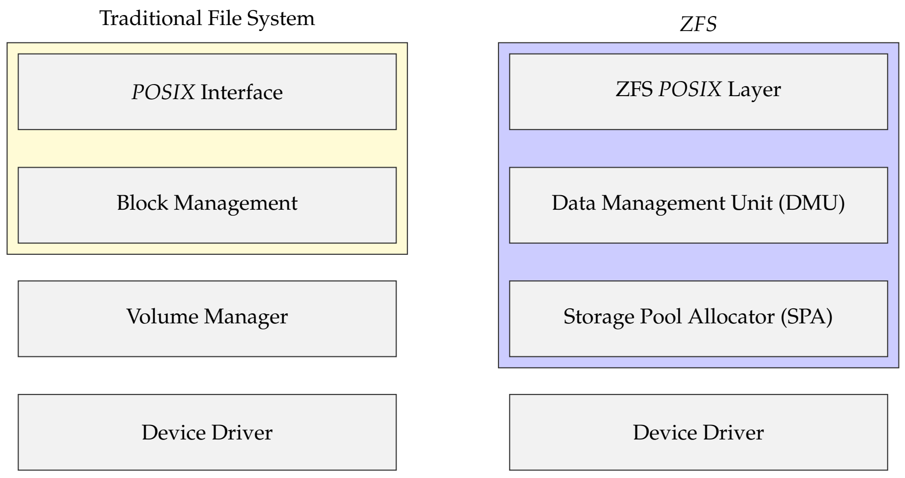


Figure 2.2: Areas of responsibilty for traditional file systems (left) and ZFS (right).

Traditional file systems work on top of a single storage device. That storage device may be a partition or an entire disk. With the help of volume managers like the *Logical Volume Manager* (LVM) of *Linux* it is possible to span a traditional file system over multiple storage devices and add features like mirroring of the data. This keeps the file system code simple and creates an abstraction between device management and actual file system work. But this also means that the volume manager does not know anything about the data it manages. All semantic information about blocks is lost in the interface between the two layers. The consequence of this is that the volume manager has to regard every single block on the disk as allocated and must therefore keep all blocks consistent. To circumvent such drawbacks ZFS has an integrated volume manager that allows pooled storage devices. As shown by Figure 2.2 this means a larger area of responsibility for ZFS.

2.3.1.2 Maintaining Consistency

Maintaining a consistent state of the on-disk data is a very important task of the file system. In case of power outage traditional file systems will encounter an inconsistent state (provided that modifications to the data happened). Different approaches exist to bring the disk back into a consistent state. One way is to use a log. Before modifications to the on-disk data are made the file system will write into the log what will be done. After the reboot log and on-disk state will be compared. Inconsistencies will be removed by replaying the log. A better way to deal with consistency would be to never allow an inconsistent on-disk state in the first place. In ZFS the disk is consistent at all times.

2.3.1.3 File System Capacity

Because the authors of ZFS assumed storage capacity to double every 9 to 12 months using 64 bit addresses seemed not enough. The maximum capacity achievable with a 64 bit file system is 16 exabyte. ZFS prepared for the future as it has an address space of 128 bit. This allows ZFS to store up to 256 quadrillion zettabytes. Allowing file systems of such sizes entails the need to grow (or shrink) them as it is hard to predict the future size of a file system. To solve this ZFS uses dynamically sized file systems that grow or shrink as the users add and delete data.

2.3.1.4 Error Detection and Correction

Bugs in areas ZFS has no control over such has the firmware of disks or *Redundant Array of Independent Disks* (RAID) controllers can cause erroneous data to be written to disk. Another problem is data that can no longer be read from the disk. Such an incident is called *Unrecoverable Read Error* (URE). How often an URE may occur is part of disk specifications. Enterprise disks have an URE error rate of 10^{-16} whereas consumer drives have a rate of 10^{-14} or 10^{-15}. An URE of 10^{-16} means that at most one bit may be erroneous for every 10^{16} read bits. To detect such errors ZFS will validate everything it reads from disk instead of trusting the data. Once such an error is detected it should be corrected (if possible) by writing a correct copy of the corresponding block back to the disk.

2.3.2 ZFS Architecture

The following sections will describe most of the relevant components that can be found in ZFS. The components can be placed (roughly) in three layers: the interface layer, the transactional object layer and the pooled storage layer (see Figure 2.3). The place of a component inside of a layer does not correspond to the internal architecture of ZFS and is purely for aesthetics.

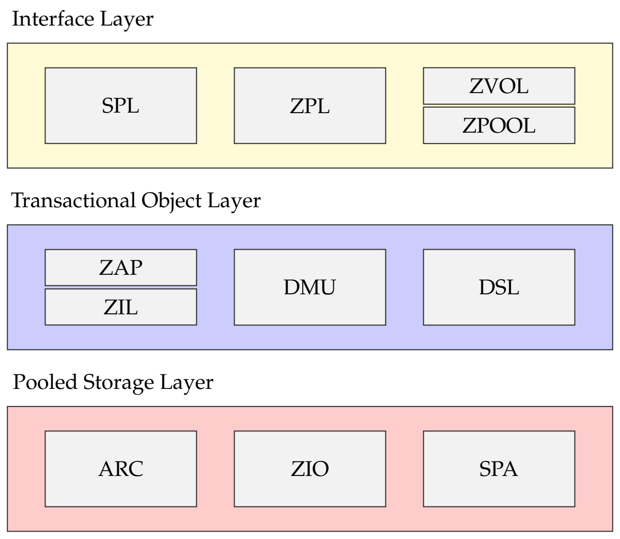


Figure 2.3: Basic overview of the layered ZFS architecture and where its most important components can be found.

2.3.2.1 Interface Layer

The interface layer of ZFS allows administrative tasks such as creating storage pools, volumes or setting up data replication. To conform with standards it implements the necessary functions to achieve *POSIX* compliance. Also part of the interface layer is the *Solaris Porting Layer* (SPL) which is used to bring ZFS to *Linux*.

Solaris Porting Layer As ZFS's origin is *Solaris*, getting it to work on *Linux* required additional work. *Linux* and *Solaris* have a similar kernel *Application Programming Interface* (API), but not all that is available in *Solaris* can be found in *Linux* and vice versa. Thus the SPL was created in an effort to bring ZFS to *Linux*. This makes the port possible with very minimal architecture specific changes to the actual ZFS codebase.

***POSIX* Layer** The *ZFS POSIX Layer* (ZPL) is responsible for compliance with *POSIX* file system aspects. *POSIX* stands for Portable Operating System Interface and is specified by the *IEEE*. *POSIX* specifies many things of *Unix*-like operating systems such as command line utilities and shells. It also has a standard for threads, *Pthreads*. For a file system to be *POSIX* compliant many things have to be taken into account. The goal is to create a uniform behaviour across all file systems to simplify development of applications that make use of the file system. Among other things this means to support:

- Hierarchical file names
- *Unix* permissions
- Three file times:
  - Last modified time
  - Last accessed time
  - Last changed time
- Support for `fsync()` / `dirfsync()`
- Access control lists
- Extended attributes

This is all taken care of by the ZPL. The ZPL sits on top of the *Data Management Unit* (DMU), consumes its objects (ZFS is object based) and makes them look like a *POSIX* file system.

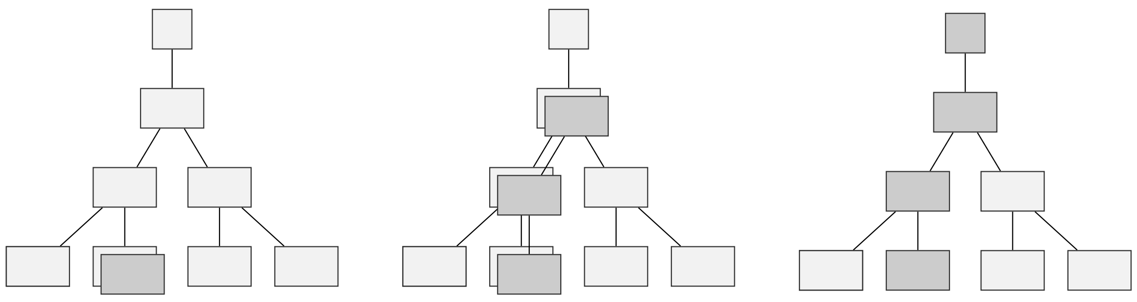
Storage and Volume Management As ZFS is more complex than a traditional file system it has to provide further (non *POSIX*) interfaces. These interfaces provide access to managing (pooled) storage devices and volumes.

With the command `zpool` it is possible to create storage pools that may span multiple devices and include data replication like mirroring or RAID. The ZVOL layer allows to create volumes inside of such pools. A ZFS volume is a virtual block device. Since this volume resides inside of ZFS pools it is backed by the ZFS *Storage Pool Allocator* (SPA) and can therefore benefit from the data replication methods. A ZFS volume works just like any other virtual (or raw) block device and as such other file systems like ext4 can be installed on it.

2.3.2.2 Transactional Object Layer

The middle layer of ZFS is the transactional object layer. The most import component of that layer is the DMU. Apart from that it contains a logging facility that ensures no data is lost in case of hardware failures, a component that makes it possible to take snapshots of file systems and an attribute processor which is used to create hash tables that are used for many things in ZFS.

Data Management Unit ZFS is an object based file system. The DMU is the heart of ZFS. It sits on top of the SPA and consumes blocks. These blocks will be grouped



Copy-on-write the data block (leaf).

Copy-on-write all intermediate indirect blocks.

Finally, rewrite the überblock in place.

Figure 2.4: Copy-on-write process of writing a new block.

into into logical units — objects. Objects are contained in an object set which is also managed by the DMU. Almost every entity inside of ZFS is represented by a DMU object. They are kept apart by their distinct object type. Some example DMU object types are (there are many more):

`DMU_OT_PLAIN_FILE` A plain ZPL file.

`DMU_OT_DIRECTORY_CONTENTS` A ZPL directory.

`DMU_OT_OBJSET` Collection of objects.

`DMU_OT_ZVOL` ZFS volume (zvol).

The DMU is in charge of maintaining consistency. A DMU object is identified by a 64 bit identifier. They can can contain 2^{64} bytes of data. Every change to a DMU object is assigned to a transaction. A transaction is an all-or-nothing operation — either the change finishes successfully, or no change is made at all. This transactional behaviour is achieved by treating every block as copy-on-write. [1, 15]

All data in a pool is represented as a tree of blocks. The root of that tree is the so-called *überblock*. The leafs are the actual data blocks, all other blocks (in-between the leafs and the überblock) are *indirect* blocks. In Figure 2.4 we can see how data blocks are modified using copy-on-write in ZFS.

When a file is changed, that means the corresponding DMU object has to be changed. A new transaction will be created (to clarify: ZFS groups changes into transactions, not every change gets its own transaction) responsible of writing the new, modified blocks that will represent the changed DMU object. The original blocks will be kept. A new block will be allocated and the entire modified block will be copied into it (copy-on-write, leaving two versions of the block). After that the tree will have to be updated, the changes will walk up the tree, starting at the leaves modifying all indirect blocks until the überblock is reached. The überblock is the only block that will be changed in place and not in a copy-on-write manner (it has backups in case of failure). Only when the überblock is successfully changed, the old copies of all changed blocks will be discarded. Once this is the case, the transaction is finished.

ZFS Attribute Processor The *ZFS Attribute Processor* (ZAP) uses the DMU to create ZAP objects. ZAP objects are hash tables used for various management tasks across ZFS. There are two versions of the ZAP, depending on how much data needs to be stored. A *FatZAP* and a *MicroZAP*. They are, for example, used for directories to provide fast entry lookup, insertion and deletion. The ZAP will store the name of files as keys and the corresponding DMU object identifiers as values.

ZFS Intent Log As the DMU likes to group multiple operations into transactions there are delays between write and write-to-disk. If the system crashes before a transaction commits, writes could be lost (but the file system would still remain consistent). To solve this problem, ZFS maintains the *ZFS Intent Log* (ZIL), one for each file system. The ZIL contains enough information about the transaction to be able to replay it when the system comes back online.

Dataset and Snapshot Layer The *Dataset and Snapshot Layer* (DSL) is responsible for managing datasets. A dataset in ZFS can be any of the following:

File system Stores and organizes objects.

Clone Initially identical clone of a file system.

Snapshot Read-only snapshot of a file system or clone at a specific time.

Volume A logical volume (block device).

The DSL manages the interdependencies between datasets and is capable of creating snapshots and clones. Creating these is computationally free because of the copy-on-write properties. The DSL only has to create a new *überblock*, pointing to the same tree. Then, whenever a copy-on-write in the original tree happens it keeps the original block which otherwise would be freed. A clone can further diverge from the original tree, as it is not read-only. These "free" snapshots are a very handy feature for things like system upgrades since one can easily roll-back to a snapshot at any point in time.

2.3.2.3 Pooled Storage Layer

The pooled storage layer of ZFS is the lowest layer of all. It works directly with disk drivers and provides abstractions to those by combining multiple physical disks to a single storage pool. Additionally, data replication techniques can be used to increase the reliability of the storage pool. The performance of reading blocks from such a pool is improved by leveraging a special caching algorithm.

Adaptive Replacement Cache The *Adaptive Replacement Cache* (ARC) is used by ZFS to speed up access to recently, or frequently, used blocks by caching them. The ARC is based on a development of *IBM* [12] and was further extended for ZFS.

Traditionally a *Least Recently Used* (LRU) cache would be used for such a case. The LRU works by caching the least recently used blocks. Every block that is read gets marked with an age indicator (age bit), the oldest block will be discarded when a newer one enters the cache. This mechanic is suboptimal when large, sequential reads happen. These will fill up the entire cache and frequently used blocks (not part of the sequential read) will be discarded. *Least Frequently Used* (LFU) caches are different, they will discard newer data that is not read frequently enough.

The ARC is a (more complex) combination of LRU and LFU, that takes the best out of the two worlds. Simply put, the ARC maintains four lists to classify the items currently in the cache [13]:

LRU Block is in cache because it was recently used.

LRU Discarded A discarded block that was least recently used.

LFU Block is in cache because it is frequently used.

LFU Discarded A discarded block that was least frequently used.

These lists do not actually contain the blocks, rather they are used to identify blocks. The adaptiveness of the cache is realized by online changing of the number of allowed LRU or LFU items in the cache. To come to a decision on that matter, the information in the lists maintaining discarded blocks will be used. If many new blocks can be found in the LFU Discarded list, more LFU items will be allowed and vice versa.

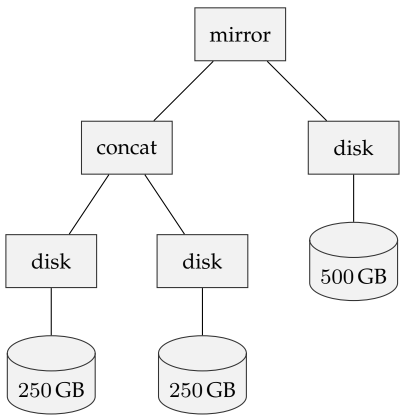


Figure 2.5: An example VDEV that is based on 2 concatenated $250\,\mathrm{GB}$ disks that are mirrored with another $500\,\mathrm{GB}$ disk (example taken from [1]).

Storage Pool Allocator The SPA works on physical disks and exports virtual devices. A virtual device is created using a *Virtual Device Driver* (VDEV). A VDEV may implement a feature like mirroring or just provide regular access to a physical device using its device driver. Virtual devices that are combined using a data replication technique like mirroring are called leaf VDEVs. This is visualized by Figure 2.5 where we can see 5 VDEVs in total. Data can be written to, and read from, VDEVs using a *Data Virtual Address* (DVA). A DVA has to be allocated first at the SPA. The SPA then translates that DVA into a physical location on one or more disks.

ZFS Input/Output The *ZFS Input / Output Framework* (ZIO) processes all blocks prior to (write) or after the disk I/O (read). It works on top of the SPA and translates DVAs into blocks. A block, from the time of its creation until it is written to disk, will go through a multi-stage pipeline (same for reads). Figure 2.6 visualizes that pipeline. It can be seen that, in case of a write, the first stage a block goes through is compression. If the compression algorithm of choice is not able to compress for more than 12.5% it will be disabled. After that the block will be encrypted (if that feature is enabled) and finally the checksum will be generated. Processing of the new block is now "finished" and it can be written to disk. The ZIO will allocate a new DVA from the SPA and subsequently the block will be written to the VDEV to which the new DVA translates. In case of a read, the pipeline looks (as expected) almost like the complete opposite, except that there is no complementary step to the allocation of a DVA.

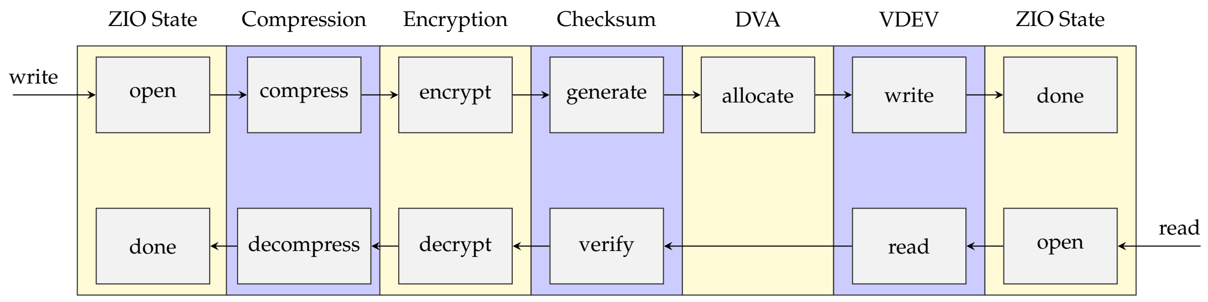


Figure 2.6: Visualization of the ZFS input/output pipeline. For the sake of simplicity, some intermediate stages were omitted. Based on [3].

Summary

This chapter has introduced the different areas of application for compression. The differences, benefits and drawbacks of conducting compression in a specific layer have been discussed. It has also presented the most important compression algorithms used in the course of this thesis. Lastly it gave an in-depth overview over ZFS.

Chapter 3

Design

This chapter presents the planned changes to ZFS. In the first section we will describe the current behaviour of ZFS regarding compression. The drawbacks of the current decision making will be discussed and a newly developed approach that is subject to several design principles is going to be introduced. The new design consists of two main components. The first, file type specific compression, will be presented in Section 3.2. The second, adaptive block compression, will be introduced in Section 3.3.

3.1 Aims and Design Principles

This section will discuss reasons for, and changes to, the decision making of ZFS compression when a new file is created. First the current behaviour and its drawbacks will be described. After that the design principles, that all changes will be subject to, are introduced. The following part will present the desired behaviour of ZFS in the event of a newly created file.

3.1.1 Current Behaviour

By default, compression in ZFS can be controlled with one property (`compression`) that can be set on a per-file-system basis. The property can be set on the fly. If done so, the new value will be used for all subsequently created files. Already written files will not be compressed again with the new algorithm. ZFS allows nested file systems that by default inherit properties from the parent file system. Changing the property in a child file system allows some flexibility as shown in Table 3.1.

In this example a media file system exists with compression turned off, as media is usually already compressed with some lossy compression algorithm. Trying to compress media with a lossless compression algorithm is a worthless undertaking. The "archive" will be compressed with gzip which yields higher compression ratios than LZ4. It could be used, for example, for backups. The home(s) of the users inherit the LZ4 compression from the pool they are part of. As a user's home directory typically contains a diverse set of different file types the compression algorithm needs to offer a good tradeoff between performance and compression.

Name	Compression	Mountpoint
pool	LZ4	/pool
pool/archive	gzip	/export/zfs/archive
pool/media	off	/export/zfs/media
pool/home	inherit (LZ4)	/export/zfs/home
pool/home/meier	inherit (LZ4)	/export/zfs/home/meier

Table 3.1: Example file system scheme.

3.1.1.1 Drawbacks

While the flexibility offered by creating multiple file systems with different selected compression algorithms is certainly a good thing it is not enough for many cases. Often it is not known in advance what kind of data will be stored in a file system. An archive containing backups compressed with gzip makes sense. Backups of the media file system however would need a separate file system for best efficiency. This still seems manageable, but trying to optimize the compression scheme of a user's home directory using multiple file systems with different compression algorithms quickly becomes very complex. To get the best possible results we could use a separate file system for countless directories of the system. This would require us to know for every directory what kind of files it would store in advance, as well as the best compression algorithm for that kind of file.

Despite it being a hassle and complicated, it is not impossible. It is however not possible (in the current version of ZFS) to get flexibility beyond directories. Setting different compression algorithms for different file types is not possible.

To deal with shortcomings like that compression algorithms like LZ4 are designed to perform well under all conditions. The tradeoff for the versatility of such algorithms is that they do not reach as high compression ratios as algorithms such as gzip.

3.1.2 Design Principles

The lack of flexibility and the huge potential of algorithms like gzip motivate changes to ZFS's handling of compression. After this part the newly developed, more flexible behaviour will be presented. This part will discuss the principles that led to the newly developed design.

3.1.2.1 Flexibility

As pointed out in Section 3.1.1 the current decision making of ZFS in terms of compression lacks flexibility. We can only distinguish between directories with different compression algorithms, not files.

The need for flexibility on a per-file-type basis is nothing new in the area of compression. It is common to provide an exclude option for applications that deal with compression. File types that are part of that exclude list will not be compressed. The file type will usually be identified by the file extension. This approach is first and foremost to filter out known-to-be-incompressible files.

Similar to the idea of an exclude list would be a more general list, that maps file types to compression algorithms. A list like that would allow excludes (mapping to `compression=off`) but also further optimizations. Log files (highly redundant, very good to compress) could, for example, be mapped to gzip.

To extend the flexibility in ZFS a solution based on these ideas should be implemented. It should be aware of the file type and decide how the file should be compressed at the moment of its creation.

3.1.2.2 Adaptiveness

Adding flexibility to ZFS in terms of file type awareness is not enough. Detecting the file type has the same disadvantage that per-file-system compression has. The file types, and how to compress them, have to be known in advance. Something that is not always possible. A simple reason for not knowing the file type is a being confronted with a file that lacks an extension, which frequently happens.

This raises the need not only for a flexible, but also adaptive solution. That solution needs to be able to react when having to deal with data about which no prior knowledge exists. A simple adaptive solution is for example applied by *OpenVPN* (see Section 2.1.3.2). *OpenVPN* adaptively compresses traffic by regularly checking whether or not the compression is effective at the moment. If it is not, it turns off the compression until the next scheduled check.

Turning off the compression when ineffective is only one side of the coin. It would be best if the solution would offer other reactions as well. For instance, a different algorithm could be tried, maybe it can compress better and the compression does not have to be turned off altogether.

3.1.2.3 Performance

The performance of the file system is a major part of the overall system performance. As such, when file systems are designed performance is always a key aspect. Good performance is no longer a feature of file systems, it is regarded as a requirement [1]. Therefore, when designing new features it is very important not to introduce new bottlenecks or to lower the overall performance by notable amounts.

Time	t	Time needed to compress
Energy	e	Energy consumed during compression
Ratio	r	Achieved compression ratio

Table 3.2: Important compression parameters.

3.1.3 Desired Behaviour

The current compression behaviour of ZFS is entirely controlled by one property: `compression`. Whenever blocks of any file within a file system are written they are compressed with the compression algorithm corresponding to the value of the `compression` property at the time the file was created. This solution is neither aware of the file type nor adaptive in any kind. Changing this requires more fine-grained control over compression in ZFS as well as an engine capable of making adaptive decisions.

3.1.3.1 Different Modes for Different Use Cases

As described in Section 2.1.4.2, file systems are not aware of the current context. They have little to no knowledge about the actual reasons as to why files are written, or how important performance is right now.

This represents a big problem as this knowledge is a very important ingredient for making the right decisions. If, for example, an application is generating lots of data as a result of a CPU heavy calculation, then performance is likely very important. Slowing down the write performance might cause a bottleneck which would slow down the calculation. However, if we are writing a backup, performance might not be so important as there is no calculation in the background that could be influenced. To the file system both undertakings look the same — lots of files are written. Being able to distinguish such use cases is very important.

Since the file system is not able to do this on its own it has to get the information from the user. In the following we will refer to such use cases as *modes*. When a compression is conducted there are many interesting parameters that can be interpreted. The three most import ones are listed in Table 3.2. Time, energy and compression ratio are of different significance in different modes. We identified three different modes which place value on different parameters. Certainly there are more specific use cases, but those would in turn need additional parameters. The three modes are: *performance*, *energy* and *archive*.

Performance The performance mode is one where the time t needed to write a block to disk is more important than the disk space it requires. The compression ratio r is also important as in some cases it is faster to compress-and-write than to write uncompressed. This is often true when the CPU is idle as this means there are plenty of resources available to compress the data. If the system is not idle, there may not be enough CPU time available to beat writing uncompressed in time.

Archive The archive mode is for long term storage of data. For example, important backups or scientific data that needs to be preserved for the future. If we know that the data will be stored for weeks, months or even years, parameters like the energy cost e to compress the data become irrelevant compared to the cost of storing the data. As a result when in archive mode the compression ratio r is of the highest significance.

Energy The energy mode represents a tradeoff between performance and archive. The energy consumed during compression is important but so is the resulting compression ratio. The energy mode will accept longer compression times than the performance mode but does not place the compression ratio above all other parameters as the archive mode does.

These modes can be used to make advanced decisions. For instance, if we know that a certain algorithm creates a huge overhead when the system is under heavy load we could choose not use that algorithm in performance mode. Similarly, if we know that the overhead results in significant gains in compression ratio we could choose that same algorithm for the archive mode. Using modes that correlate to different contexts we can come to different decisions for one and the same file.

3.1.3.2 Compression Mode Property

In order to make use of the modes introduced in Section 3.1.3.1 a new property needs to be added to ZFS. The property will be called `compression_mode` and works in conjunction with the normal `compression` property. The `compression_mode` property will be set on a file system basis, just like the `compression` property.

Thus far, with only the `compression` property, the decision making was rather simple. When a new file was created, the corresponding object got assigned a write policy by the DMU. That write policy includes the compression algorithm to be used and will be inherited from the object set (the file system). When the blocks of that file are written the value of that property is checked, and the blocks are compressed (or not) accordingly.

With the introduction of modes we have more information at hand and can decide about whether or not to compress more precisely and at different stages during the process of writing the new file to disk. Two additional mechanics will be added. The first one is, determining the compression algorithm based on the file type of the new file and the current mode (see Section 3.2). The second mechanic is to test how the blocks of the file can be compressed and decide which algorithm to use based on the parameters (see Table 3.2) that result from the compression.

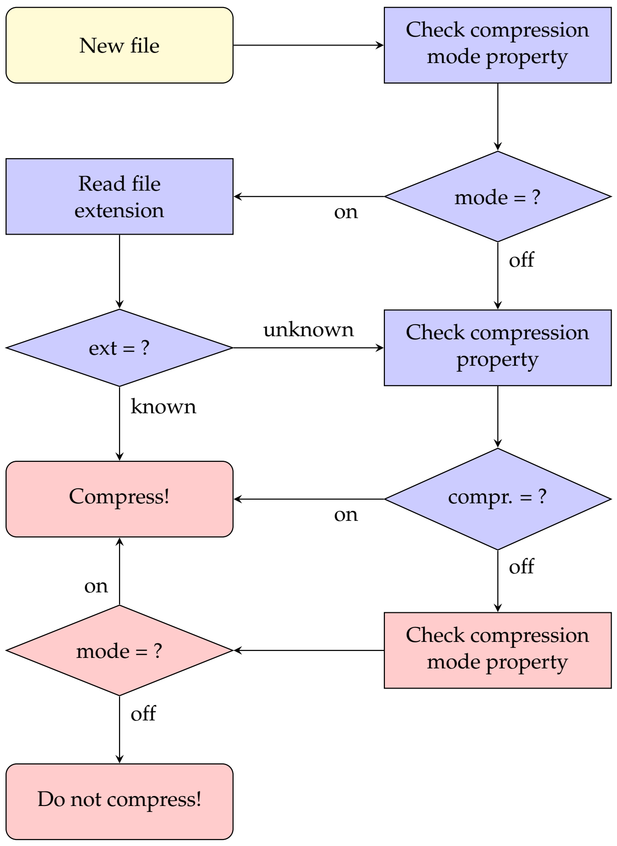


Figure 3.1: How ZFS should decide whether or not to compress a newly created file. The colors of the components refer to the layers of ZFS as seen in Figure 2.3 (yellow=interface layer, blue=object layer, red=storage layer). The "on" values of `compression` and `compression_mode` refer to any compression algorithm or a valid mode respectively.

In Figure 3.1 we can see a flowchart that represents the new behaviour and shows how the two compression properties `compression` and `compression_mode` work in conjunction. If `compression_mode` is set to off, everything will behave just as if no changes were made and the decision depends solely on the value of `compression`. If it is set to any of the three modes, two different cases (depending on the value of `compression`) can be distinguished.

`compression=on` If the `compression` property is set to any of the implemented compression algorithms and not off, then every file that is not known (unknown file extension) will be compressed with that algorithm. Otherwise (known file extension) the file will be compressed with the algorithm corresponding to mode and file type. Both of these decisions are made in the object layer of ZFS.

`compression=off` In this case every unknown file will be tested in the ZFS I/O pipeline and compressed with an algorithm depending on the compression parameters and the mode. Files with a known extension will be compressed depending on the mode.

At first it may seem counterintuitive to check the compression mode property twice, once in the object layer and once in the storage layer. However, the checks happen at two different events in ZFS. In the object layer we check the property when the file is created. In the storage layer we check it when actual blocks are written. In between those events arbitrary time spans can pass. Log files, for example, are usually created once and then data is written to them for an indeterminate amount of time.

Summary This section introduced a new property that will extend the configurability of ZFS. Additionally, two new mechanics were briefly introduced. The first one makes ZFS aware of the file type by reading the extension of a newly created file. It is done in ZFS's object layer and will be described in Section 3.2. The second mechanic evaluates how the blocks of the file in question can be compressed by weighting some compression parameters more than others, depending on the current mode. This is done in the pooled storage layer of ZFS and will be explained in Section 3.3.

3.2 File Type Specific Compression

Different types of files have different degrees of compressibility. A media file that is already compressed with a lossy compression algorithm can not be compressed again. A simple text file can be compressed very well. Some packed binary data may be good or bad to compress, it has to be determined by empirical analysis. If we gather lots of information about the compressibility of different file types we can optimize our compression system by always choosing the best algorithm for the file type depending on the context. In order to do this in ZFS we have to be able to determine the file type when a new file is created. The only feasible way to do so is by reading the file extension even though this is no perfect solution.

3.2.1 The File Name Extension

It is a de facto standard to provide an extension to file names, the file name extension. The extension is separated from the rest of the file name with a dot. The usage of file extension varies between operating systems and is not standardized. On *Windows* the extension is used to determine which application should be used to open it. *Unix* like operating systems have additional measures to determine the right application such as reading the header of the file (the first bytes). On *Unix* systems the file extension is also used for other purposes such as displaying the version number of a library. It is also common to put together multiple extensions to indicate an "inner" and an "outer" file type.

File name	Extension	Actual meaning
notes.txt	txt	Text file
data.nc	nc	*NetCDF* data
backup.tar.gz	gz	Compressed *tar* archive
libxcb.so.1.1.0	0	Version 1.1.0 of libxcb.so

Table 3.3: Example file extensions.

Table 3.3 shows that the usage of file name extensions is not consistent at all and in some cases hard to parse reliably. For the human eye it is easy to see that we are talking about version 1.1.0 of libxcb. Since the version number contains additional dots it becomes hard to parse though. Furthermore, some files lack an extension entirely. This is the result of file name extensions not being standardized.

Nevertheless, using the file extension works for most files and thus determining the file type by interpreting the extension remains a viable way, even though it is not perfect.

3.2.2 Extension Lookup Table

Once the file type is determined the information has to be correlated with the current `compression_mode` in ZFS. We choose to do so using a lookup table as shown in Table 3.4. This should be done in the object layer of ZFS as the name of the file is still at hand in that layer. If the extension is known, a compression algorithm will be looked up that may be different to the one of the file system. Therefore, when the write policy of the file is set the algorithm should already be determined. As extensions that contain dots in itself are hard to parse the table will only support the simplest format. It will however work with extensions like `tar.gz` which will refer to `gz` in the table.

3.2.3 Adding Entries to the Lookup Table

There are multiple possible ways to feed data into ZFS that can be considered to build the lookup table. The simplest way would be to compile the table into the ZFS kernel module. Another way would be to use extended attributes. Extended attributes are a feature of many file systems that allow the user to set additional meta data on files and folders. Finally, ZFS allows to set custom properties if they contain a colon in the name. New entries could be added like this:

```
zfs set compress:archive:nc gzip-1
```

Extension	Archive	Energy	Performance
txt	gzip-6	gzip-1	LZ4
log	gzip-9	gzip-6	gzip-1
nc	gzip-1	LZ4	LZ4
gz	off	off	off

Table 3.4: Example extension lookup table.

Using ZFS custom properties is certainly the nicest way on the paper, but it has a disadvantage in common with extended attributes. Both are designed to be used for user space tasks and not to change the internal state of ZFS. ZFS simply does not know what to do with them beyond persisting them on the disk. Therefore both will very likely require considerable code changes.

3.3 Adaptive Block Compression

If we are confronted with a file that lacks an extension, or the extension is simply unknown, and the compression property is not set to off, we would use the chosen compression algorithm and hope for the best. To still benefit from a more intelligent compression we will test the file for compressibility by compressing some of its blocks and evaluating the resulting parameters depending on the chosen compression mode. As shown by Figure 3.1 this is done in the lowest layer of ZFS if previously no known extension was detected and the compression property is set to off.

3.3.1 Testing Blocks of a File

The goal is to test some blocks of a file and evaluate the resulting parameters depending on the current mode. If we are in performance mode and one algorithm is significantly faster than the rest, then that algorithm will be used for the rest of the file. In archive mode the amount that we compressed matters most and in energy a compromise between compression time and compression ratio is what causes an algorithm to win. The winning algorithm should then be used until the next test is conducted.

The biggest raised question when testing blocks is how often it should be done. Testing every block with multiple algorithms is hardly feasible. At the same time, testing too infrequently may render the results useless as the characteristics of the blocks may have already changed.

A file in ZFS is represented by an object inside the DMU — a *dnode*. The dnode will be written to disk in blocks of a size between 512 B and 128 kB depending on the chosen block size. If the file is bigger than 128 kB it will be structured as a tree, where indirect blocks hold pointers to the actual data blocks. The pointer to a block is 128 B large. The biggest indirect block (128 kB) can therefore hold pointers to 1024 blocks. That means per indirect block we can store 128 MB of data. However, the default size for indirect blocks in ZFS is 16 kB and as such the maximum size an indirect block can store is 16 MB of data. Therefore, if we conduct the test at the first data block of an indirect block, the result will be used for the next (up to) 127 blocks (if the file is that large).

3.3.2 Difficulties

Since ZFS is heavily threaded the blocks are written in parallel. Every block will go through the I/O pipeline (see Figure 2.6), but it is not guaranteed that the data blocks of an indirect block will go through it sequentially. Hence some problems need to be worked out in order for this approach to work. In the following we will refer to the blocks after the first block as secondary blocks. The important key points that need to be solved are:

Identifying the first block When a new block enters the compression routine we need to be able to identify it. Is it the first block or one of the secondary blocks? If it is the first block, the tests should be conducted.

Secondary blocks outpacing the first block If blocks enter the compression routine while the first block is still testing compression algorithms the blocks will have to wait for the first block to finish. Waiting is always performance critical. *Busy-waiting*[1] should be avoided to prevent wasting of energy. Instead the threads should wait (if necessary repeatedly) for a fixed amount of time (wait until timestamp). This implies a tradeoff as threads might wait longer than actually needed.

Notifiying secondary blocks of the result Once the first block has finished the tests the secondary blocks should be able to retrieve the results.

3.3.3 Cost Function

When a block is tested we have several parameters that can be used to interpret the compression. The most important ones are listed in Table 3.2. It hast to be noted that the compression ratio has to be calculated from the physical size (*psize*, compressed size — actual used disk space) and the logical size (*lsize*, uncompressed size). These parameters (ratio, time and power) can be used to calculate the *cost* of the compression. For this we use the cost function 3.1. The cost function is defined as follows:

[1]Repeatedly checking whether or not the wait-condition is still active.

$$c = \frac{(t^{m_t} \cdot p)}{(n_l - n_p)^{m_r}} \tag{3.1}$$

where:

c	$=$ compression cost
t	$=$ compression time
p	$=$ power consumption
m_t	$=$ time multiplier
m_r	$=$ ratio multiplier
n_l	$=$ logical size
n_p	$=$ physical size

The algorithm with the lowest cost under the current circumstances is the winner. In order to get different results for the different modes we weight the parameters accordingly. When in performance mode, the time is exponentially weighted with a time multiplier and when in archive mode the compression ratio is exponentially weighted with a ratio multiplier.

Summary

This chapter presented the planned changes to ZFS. To extend flexibility of ZFS a new compression mode property was introduced. The new property can be used in conjunction with the default compression property of ZFS to control two new features. The first, file type specific compression, chooses a compression algorithm based on the extension of a newly created file and the current compression mode. The second feature, adaptive block compression, tests blocks of a file for compressibility with different algorithms and makes a choice based on the compression mode and the resulting parameters.

Chapter 4

Related Work

This chapter gives an overview of existing and related work in the field of adaptive compression and file systems. Section 4.1 will present adaptive compression solutions which are primarily used in the transport layer. In Section 4.2, the compression algorithm LZ4HC will be introduced. Furthermore, in Section 4.3, Btrfs, its differences to ZFS and why it was not suited as well as ZFS for the purpose of this thesis will be discussed.

4.1 Adaptive Compression Solutions

This section presents solutions that adaptively compress data to meet external constraints or to avoid weaknesses of used algorithms by filtering out data that is known-to-be incompressible.

OpenVPN and rsync

As already mentioned in Chapter 2, both *OpenVPN* and *rsync* use adaptive methods in their protocols. This is primarily motivated by the fact that arbitrary data can be transferred by both, which includes incompressible data. In order to circumvent performance problems which the compression of that data might cause, the tools follow different approaches.

OpenVPN regularly measures the efficiency of the compression. If it detects that the compression fails to reach a minimum compression ratio it will turn it off until the next check is scheduled. *OpenVPN* uses LZO for this procedure. Both the minimum amount that has to be compressed and the frequency are hard coded and cannot be changed by the user (if he does not want to recompile). By default, *OpenVPN* will measure the efficiency for a two second period and schedule a test every 60 seconds. The data has to be compressed for at least 5 %, otherwise the compression will be turned off [16].

A different approach has been taken by *rsync*. It relies on identifying incompressible files by using a list of file extensions that belong to known-to-be incompressible files. The extensions can be provided on the command line using the `--skip-compress` option. By default, extensions that correspond to known compression algorithms such as `gz,zip,bz2,7z` are skipped.

The approaches used by *OpenVPN* and *rsync* work well for their specific purposes. Skipping the compression for known-to-be incompressible files as done in *rsync* is similar to our file type specific compression. However, as files may show different degrees of compressibility and not just compressible / incompressible the more fine grained control offered by file type specific compression was needed. The adaptive compression of *OpenVPN* is similar to our adaptive block compression, but as with *rsync*'s skip compress option it only considers compressible and incompressible as options which is not enough for our purpose. Apart from that the checks are far too coarse and as such do not offer the accuracy we have in mind.

Image Compression

Image compression is usually done with lossy compression algorithms. Typically, they offer many parameters that allow the user to control how much quality the algorithm may sacrifice in return for more space savings. One algorithm that offers several parameters to control the quality of the compressed file is the JPEG image compressor. It is heavily used across the WWW to significantly reduce the size of images. This is especially important for mobile devices which have a finite supply of energy and are therefore forced to budget their energy consumption. Both the compression and transmission of data consume significant amounts of energy.

Clark N. Taylor and Sujit Dey [19] present an adaptive solution that varies compression parameters to minimize energy consumption on the device while meeting latency, bandwidth and quality constraints. They use two different parameters of JPEG, *Quantization Level* (QL) and *Virtual Block Size* (VBS), which have different effects on energy consumption, compression ratio and image quality.

Determining the optimal parameters for the current situation consists of three steps. The first two steps are done offline. In those steps they precompute tables in which the radio of the device can lookup compression parameters. Only the third step is done online at run-time. In that step the radio of the device continuously monitors the current image quality, latency and bandwidth constraints and adjusts the compression parameters accordingly. For example, to meet a certain energy constraint the radio may have to decrease the VBS of the compression which lowers its energy consumption but also decreases the image quality. To maintain the image quality (too meet a constraint) the QL has to be decreased which has no influence on the energy consumption but decreases the compression ratio. This however increases the energy consumption of the network interface as more bits have to be transmitted. In order to find the optimal tradeoff between compression and transmission energy cost an algorithm compares parameter combinations until the optimal combination of QL and VBS has been identified.

The adaptive image compression presented in this section works well in its particular domain. However, for our purpose we lack the flexibility in terms of parameters that JPEG offers. The only parameter we possibly have is the compression level of gzip which cannot be compared to the unique interaction of VBS and QL.

Adaptive On-the-Fly Compression

In many cases compressing data before transmitting or writing it is overall faster than sending uncompressed data. However, that is not always the case as the compression performance is dependant on the underlying resources such as the network bandwidth or CPU utilization. In high-speed networks, for example, the situation is different. They reach such high speeds that it is faster to send the data uncompressed. Nevertheless, if the network is fully utilized compression can help to maximize throughput.

The authors of [8] developed the *Adaptive Compression Environment* (ACE) which aims to find and apply the best compression technique for the current circumstances on-the-fly. ACE intercepts program communication and adaptively selects a compression algorithm that suits the current data characteristics, network and CPU performance. ACE uses the *Network Weather Service* (NWS) to predict resource usage on both of the communicating systems [24]. NWS stores time series and when queried returns an accurate short term performance estimate for the requested resource (e.g. CPU or network). Based on that data and statistics about compressibility of previously sent blocks ACE will decide how or if to compress data prior to sending it.

ACE uses adaptive compression in order to react to changes in resource availability and data characteristics. This is similar to our adaptive block testing. The key difference is that ACE has to factor in changes on the receiving end while in our case everything happens on one system. Therefore we do not have to use a prediction engine like NWS.

4.2 LZ4 High Compression

LZ4 is a different version of LZ4 which aims for higher compression ratios. It does so by doing a full search for redundant data in the data to be compressed . Typically, this increases the compression ratio by 20 % while being six to ten times slower [11]. The special thing about LZ4HC is that it shares exactly the same format with LZ4. Therefore, it is possible to use the standard LZ4 decompressor with data that has been compressed with LZ4HC. This makes LZ4HC interesting for scenarios where compression speed is not important as it allows the user to benefit from better compression ratios while maintaining the extremely fast decompression speed of LZ4. For example, data that will only be compressed once but decompressed very often.

LZ4HC is an interesting compression algorithm that would work great in the archive mode of both adaptive block testing and file type specific compression. Unfortunately it is not yet implemented in ZFS although it should be fairly simple as the necessary decompressor is already on board.

4.3 B-tree File System

Btrfs is very similar to ZFS in many aspects. Both are copy-on-write file systems that checksum data, support transparent compression, may span multiple devices and are capable of creating writeable snapshots. However, the ways both file systems implement the aforementioned features are fundamentally different.

ZFS divides disks into blocks which are subsequently consumed by objects of the DMU. Blocks are compressed, checksummed and objects are modified by copy-on-writing the blocks. The default block size in ZFS is 128 kB. When compressing blocks, the compression function operates on the entire block. A typical sliding window (see Section 2.2.3) has a size of 32 kB, and as such the compression has an adequate amount of data to build a dictionary to look up redundant data.

Contrary to that, Btrfs stores data in extents. Extents are continuous runs of on-disk area [18]. Each file gets its own extent and the compression can be stored per-extent. However, the current compression support has a serious drawback as the compression function only compresses 4 kB[1] sized blocks, one at a time. The compressor is not able to reuse state between two blocks and as such the dictionary will be very limited.

Btrfs has two compression algorithms implemented, LZO and zlib. They can be compared to LZ4 and gzip in terms of speed and compression ratio. However, the drawbacks in the way Btrfs compresses data, and the limited choice of algorithms, made Btrfs an inferior choice for the purpose of this thesis.

Work is being done to implement a new compression format in Btrfs that should resolve these issues. The new compression format will allow the compressor to work on bigger blocks. Once that is done it is planned to implement LZ4, LZ4HC and the *Lempel-Ziv-Markov chain algorithm* (LZMA) in Btrfs[2].

Summary

This chapter presented adaptive compression solutions. Due to the nature of the lossy JPEG algorithm the presented adaptive image compression shows a good way to utilize adaptive compression. ACE demonstrates how adaptive compression can be used in distributed environments with changing resource availability. Furthermore, this chapter presented LZ4HC, which at the time of this writing, was not implemented in ZFS. Therefore, we can not make use of it in the archive mode for which it seems very well suited. Lastly, we discussed why Btrfs was an inferior choice for the purpose of this thesis.

[1]*Linux* page size.

[2]`https://btrfs.wiki.kernel.org/index.php/Project_ideas`

Chapter 5

Implementation

This chapter describes the implementation of the features presented in Chapter 3. The first section describes how basic operations like creating a file, or writing to it, work in ZFS. Section 5.2 describes how the new compression mode property was implemented in ZFS. The following sections 5.3 and 5.4 will present the implementation of file type specific compression and adaptive block compression, respectively.

5.1 ZFS Create and Write

Before we discuss the implementation of the features outlined in Chapter 3, we will describe how ZFS creates and subsequently writes to files. This is where the compression algorithm is determined and the data gets compressed. Therefore it is of essential importance to understand when and how files are created, and written to, in ZFS.

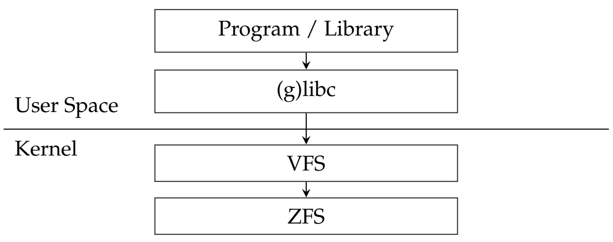


Figure 5.1: Calls from user space to ZFS will go through the VFS.

Everything the user initiates will have to go through the *Virtual File System* (VFS). The VFS is in charge of separating file system code from the rest of the *Linux* kernel. Together with the ZPL this makes two layers of abstraction before user land calls actually reach ZFS. In Figure 5.1 we can see how a call from user space will go through a C library and the VFS, before reaching the *POSIX* layer of ZFS. Inside of ZFS all calls start in the ZPL where they wrap around *vnode* operations. Every object inside of a file system (file / folder) is represented by a vnode. Vnodes are virtual representations of files and folders that have a unique *Identifier* (ID). They are managed by the VFS. Two of these vnode operations are `zfs_create` and `zfs_write`. They are used to create a file, or write to it, respectively. These are the two most important calls for the purpose of this thesis. During `zfs_create`, among other things, the compression algorithm will be chosen, whereas during `zfs_write` actual data will be written and compressed.

In Section 5.1.1 (create) and 5.1.2 (write) the process of these two routines will be described. Section 5.1.3 will talk about the synchronization of buffers since a call to `zfs_write` does not immediately write to disk, but instead to a buffer. The actual compression happens later, when the buffer is synchronized to disk.

5.1.1 ZFS Create

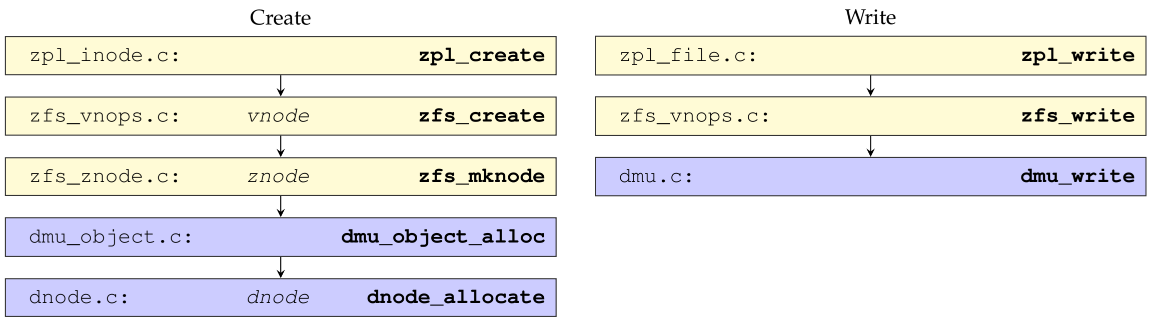

Create
zpl_inode.c: zpl_create
zfs_vnops.c: vnode zfs_create
zfs_znode.c: znode zfs_mknode
dmu_object.c: dmu_object_alloc
dnode.c: dnode dnode_allocate
Write
zpl_file.c: zpl_write
zfs_vnops.c: zfs_write
dmu.c: dmu_write

Figure 5.2: Call sequence to create, or write to, a file in ZFS.

When we create a file with an application or a library, our call will go through five routines before the corresponding ZFS object is created. Figure 5.2 visualizes the sequence of these calls. We begin with the *POSIX* layer, where `zpl_create` corresponds to the `fopen()` call in a C library like *glibc*. Next are the ZFS vnode operations where `zfs_create` gets called and subsequently calls `zfs_mknode`. This call creates a *znode* which is the ZFS equivalent to the vnodes of the VFS. The last interface layer call, `zfs_mknode`, is in charge of creating a DMU object for the newly created file. This operation is wrapped in a DMU transaction. If the transaction succeeds a dnode was created, the internal object representation of a vnode or znode, respectively. When the dnode is allocated in `dnode_allocate`, it gets assigned several properties of ZFS that by default are set to inherit (Listing 5.1). Inherit means, when needed, the value will be taken from the object set rather than the dnode itself. This means the compression algorithm can be stored per-dnode (per-file). Note that, when creating a directory the sequence looks the same, apart from the first two calls being `{zpl/zfs}_mkdir` instead of `create`.

```
dnode_allocate(dnode_t *dn, dmu_object_type_t ot, int blocksize, int ibs,
    dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx)
{
    ...
    dn->dn_checksum = ZIO_CHECKSUM_INHERIT;
    dn->dn_compress = ZIO_COMPRESS_INHERIT;
    ...
}
```

Listing 5.1: Initialization of compression/checksum algorithm per file.

5.1.2 ZFS Write

Once created, we can write to a file (see Figure 5.2). The entry point for this is again the *POSIX* layer of ZFS with the function `zpl_write`. After that, just like in `zfs_create`, a ZFS vnode operation (`zfs_write`) takes care of wrapping everything in a transaction. This operation then calls `dmu_write` where, as the name indicates, the DMU writes the actual data to the internal object representation (dnode) of the file. One would assume that this is where the data is compressed, but that step happens later as the DMU only writes to a buffer. The SPA is responsible of persisting the internal state to disk, and just before data is written to disk, it will be compressed. Therefore we will need to have a look at how ZFS synchronizes buffers onto the disk in order to get a closer look at how, and where, the compression is conducted.

5.1.3 Synchronizing Buffers

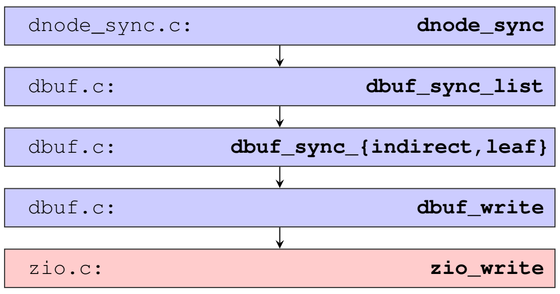


Figure 5.3: Buffer synchronization.

There are several operations that initiate a synchronization of buffers and disk in ZFS. For example, when the user calls `sync`[1], `fsync`[2] or when a ZFS pool is exported. In any case, at some point, `dnode_sync` is called to synchronize a file. If the operation concerns the entire pool this routine is called for every dnode that has "dirty" (unsynchronized) buffers. Since the blocks of a dnode are structured in a tree, ZFS has to walk along that tree and synchronize the blocks which consist of indirect and data blocks (leafs). The routine `dbuf_sync_list` makes sure that all blocks are included. Just before a block enters the ZFS I/O pipeline through a call to `dbuf_sync_{indirect,leaf}`, its write policy is specified. This is where the compression property which was set in `dnode_allocate` is evaluated. The function `dbuf_write` creates two important structs for every block: `zio_prop_t` and `zio_t`. The chosen compression algorithm is stored in the `zio_prop_t` along with other properties. Among them are the chosen checksum algorithm, if the block is replicated somewhere or if it should be considered for deduplication. The other struct, `zio_t`, holds the actual data. With those structs, `dbuf_write` then calls `zio_write`. That call initiates the I/O pipeline where one of the first stages of the block is compression in the function `zio_write_bp_init`.

[1]Synchronizes data on disk with memory.
[2]Synchronize file's state with storage device(s).

```c
if (compress != ZIO_COMPRESS_OFF) {
    void *cbuf = zio_buf_alloc(lsize);
    psize = zio_compress_data(compress, zio->io_data, cbuf, lsize);
    if (psize == 0 || psize == lsize) {
        compress = ZIO_COMPRESS_OFF;
    }
}
```

Listing 5.2: Block compression in `zio_write_bp_init` (simplified).

In Listing 5.2 we can see how the block data gets compressed with the algorithm chosen in `dmu_write_policy` (called by `dbuf_write`). If the compression yields no savings (physical size = logical size) the compression for the block will be set to off.

5.2 Compression Mode Property

As the name suggests, the compression mode property should be implemented as a ZFS property, changeable on the fly. The property should be stored — as the normal compression property — on a per-file-system basis. That means we will have to store it in the object set. To implement this we had to (among other, minor, things) add a lookup table in `zfs_prop.c` and register the property as an "index" type property in the same file (see Listing 5.3). We registered the property as `PROP_INHERIT` which means child object sets (nested file systems) will inherit the property (just as the normal compression property).

```c
static zprop_index_t compress_mode_table[] = {
        { "off",        AC_MODE_OFF },
        { "archive",    AC_MODE_ARCHIVE },
        { "energy",     AC_MODE_ENERGY },
        { "perf",       AC_MODE_PERFORMANCE }}
};

zprop_register_index(ZFS_PROP_COMPRESSION_MODE, "compression_mode",
    AC_MODE_DEFAULT, PROP_INHERIT,
    ZFS_TYPE_FILESYSTEM | ZFS_TYPE_VOLUME,
    "off | archive | energy | perf", "COMPRESSION_MODE",
    compress_mode_table);
```

Listing 5.3: Adding a new property (compression mode) to ZFS.

Additionally, we had to provide a callback that specifies what should happen in case the value changes (see Listing 5.4). We just make sure that it is a valid value and update the entry in the object set. The callback is registered when the object set is opened and the to be opened object set is passed as an argument.

```
static void compression_mode_changed_cb(void *arg, uint64_t newval) {
        objset_t *os = arg;
        ASSERT(newval < AC_NUM_MODES);
        os->os_compress_mode = newval;
}

int dmu_objset_open_impl(spa_t *spa, dsl_dataset_t *ds, blkptr_t *bp,
    objset_t **osp) {
        ...
        err = dsl_prop_register(ds,
            zfs_prop_to_name(ZFS_PROP_COMPRESSION_MODE),
            compression_mode_changed_cb, os);
        ...
}
```

Listing 5.4: Adding a callback for compression mode property changes.

5.3 File Type Specific Compression

As we have seen in Section 5.1, ZFS allows us to store a dedicated compression algorithm for a specific file. However, as shown in Listing 5.1, by default the compression is set to inherit which means it takes the value of the object set. Unfortunately, the compression algorithm is not part of the parameters of `dnode_allocate` and neither is the file name still available inside `dnode_allocate`.

As a consequence, we will have to decide between two modifications. The first one is to pass the file name all the way down into the object layer and evaluate the file extension there. The other one is evaluating the extension in the interface layer, and passing down the chosen compression algorithm down into the object layer. We have opted for the second modification, which means evaluating the extension in `zfs_create` inside `zfs_vnops.c` as this is the last function where the file name is still available. Note that available in this case means "available with reasonable effort". Theoretically the name is available in other layers as well, but retrieving it would require manners that go against the design of ZFS. We would have to search the specific ZAP (a hashtable which maps file names to object ids) by value, something that is not supported and likely not performant. In the end that meant that we had to change the signature of three functions in ZFS:

1. `zfs_mknode(..., `**`compression)`**
2. `dmu_object_alloc(..., `**`compression)`**
3. `dnode_allocate(..., `**`compression)`**

Once this change was done, two things were left to do. In `zfs_create` the file extension had to be evaluated. The extension should then be used together with the compression mode to retrieve a compression algorithm from a lookup table as described in Section 3.2. Before that, though, we had to decide how the lookup table should be maintained and edited. Sections 5.3.1 and 5.3.2 cover the implementation of this. With the implementation of the lookup table and the mapping between file extension and file type, choosing the right compression algorithm in `zfs_write` is simple as can be seen in Listing 5.5.

```
ac_mode = os->os_compress_mode;
if (ac_mode != AC_MODE_OFF) {
        compress = ac_compress_select(ac_mode, name);
} else {
        compress = os->os_compress;
}
zfs_mknode(dzp, vap, tx, cr, 0, &zp, &acl_ids, compress);
```

Listing 5.5: Selecting the compression algorithm for the new file.

5.3.1 Lookup Table

Ideally, the lookup table should be editable by the user online, when the file system is mounted and used. As described in Section 3.2.3, the two considered ways that would make this possible had disadvantages. Therefore we had to use a compiled list of known extensions. Adding entries to the table using extended attributes was implemented but later removed as it had too many drawbacks.

The lookup table consists of an array of entries of type `ac_file_info_t` (see Listing 5.6) which has two members. The first one is the extension and the other is a list of compression algorithms — one for each mode.

```
typedef const struct ac_file_info {
        char        *ext;
        uint8_t     compressions[AC_NUM_MODES];
} ac_file_info_t;
```

Listing 5.6: Mapping an extension to a list of compression algorithms.

Listing 5.7 shows how the lookup table looks like (with example entries). The table is both used to check whether an extension is known and to lookup the compression algorithm if it is the case.

```
const ac_file_info_t ac_lookup_table[AC_NUM_KNOWN_FILES] = {{
        /* PERFORMANCE        ARCHIVE                 ENERGY */
 "txt", {ZIO_COMPRESS_LZ4, ZIO_COMPRESS_GZIP_5, ZIO_COMPRESS_GZIP_1}},{
 "gz",  {ZIO_COMPRESS_OFF, ZIO_COMPRESS_OFF,    ZIO_COMPRESS_OFF}}
}
```

Listing 5.7: Lookup table with example entries.

Looking up an entry, once the type and compression mode are known, is simple as shown in Listing 5.8. Both the type of the file and the selected compression mode are mapped to integers which are subsequently used as array indices to retrieve the appropriate algorithm. If the provided index (file type or compression mode) is unknown, the compression is set to inherit. That means in the following ZFS will behave just as if no changes were made.

```c
uint8_t
ac_lookup_compression(uint8_t ac_mode, uint8_t ac_filetype)
{
        if (ac_mode >= AC_NUM_MODES || ac_filetype >= AC_NUM_KNOWN_FILES)
                return ZIO_COMPRESS_INHERIT;

        return ac_lookup_table[ac_filetype].compressions[ac_mode];
}
```

Listing 5.8: Looking up entries in the table.

5.3.2 Extracting the Extension

Extracting the extension first required us to define what should be considered as the extension. As described in Section 3.2.1, we opted to consider everything past the last dot in the file as the file extension.

```c
unt8_t
ac_get_filetype(char *fname)
{
        char            *ext;
        uint8_t         type;

        if (fname == NULL)
                return AC_FILE_UNKNOWN;

        ext = strrchr(fname, '.');
        if (ext != NULL)
                ext++;

        if (ext && !(ext[0] == '\0')) {
                for (type = 0; type < AC_NUM_KNOWN_FILES; ++type) {
                        if ((strcmp(ext,ac_lookup_table[type].ext) == 0))
                                return type;
                }
        }

        return AC_FILE_UNKNOWN;
}
```

Listing 5.9: Extracting the extension from the file name.

As shown in Listing 5.9 we use strrchr() to obtain a pointer to the last occurrence of the character ". " in the file name. Incrementing this pointer by one leaves us with just the extension without the dot. That pointer is then used to compare the extension with all known ones inside of a loop. The current loop index when a match was found is our file type. If no match is found we return AC_FILE_UNKNOWN which causes ac_lookup_compression to return ZIO_COMPRESS_INHERIT. We considered to use a more efficient way to lookup entries inside the table, other than looping over all of them. But in order for the performance gains by this to matter, the lookup table had to become remarkably large.

5.4 Adaptive Block Compression

This section will describe how the implementation of adaptive block compression (see Section 3.3) was done. We decided to test the first block of every indirect block, which means the decision made there will be used for the next (up to) 127 blocks. As outlined in Section 3.3.2, we need to make sure that we can exchange information between the first and the secondary blocks. Furthermore, we need to make sure that the secondary blocks do not overtake the first block. This is discussed in Section 5.4.1. As we will see in Section 5.4.2, implementing the cost function introduced in Section 3.3.3 proved to be more difficult than assumed.

5.4.1 Synchronizing First and Secondary Blocks

To exchange the necessary information between the different secondary blocks (children), we decided to use the zio_t struct that is part of every I/O operation. We added three members as shown in Listing 5.10.

```
typedef struct {
    ...
    boolean_t  io_first_child;        // Am I the first block?
    boolean_t  io_first_child_ready;  // Is first block ready with testing?
    uint8_t    io_ac_compress;        // Which algorithm won the test?
    ...
} zio_t;
```

Listing 5.10: New members for zio_t which are used to exchange information between the first and secondary blocks.

When a block enters zio_write_bp_init it will check in the zio_t struct of its parent if it is the first block. If so, it will set io_first_child to false (see Listing 5.11). All other blocks that share the same parent with the first block will have to wait until it finishes testing the chosen compression algorithms.

```
zio_t *pio = zio_unique_parent(zio);

mutex_enter(&pio->io_lock);
if (pio->io_first_child) {
        pio->io_first_child = B_FALSE;
}
mutex_exit(&pio->io_lock);
```

Listing 5.11: The first child will set `io_first_child` in the parent `zio_t` to false.

The first child will then start the tests. When it is done, it will set `io_first_child_ready` (see Listing 5.12) to true and if there were waiting secondary blocks they will continue to progress in the I/O pipeline.

```
cbuf = zio_buf_alloc(lsize);
ac_compress = ac_block_compress_select(ac_mode, lsize, zio->io_data,
   ↪ cbuf);

mutex_enter(&pio->io_lock);
pio->io_ac_compress = ac_compress;
pio->io_first_child_ready = B_TRUE;
mutex_exit(&pio->io_lock);
```

Listing 5.12: The first block calls `ac_block_compress_select` to determine which algorithm should be used in the following and stores the result in the parent's `zio_t`.

The secondary blocks are waiting in the beginning of `zio_write_bp_init` (see Listing 5.13) where they call `zio_wait_for_first_child` (shown in Listing 5.14) until the first block is done . If that method returns true, they will call `usleep_range`. This sleep function does not busy wait. Instead, `usleep_range` is called with a *range* (e.g. 10 000-12 000 µs). The documentation states [6]:

> "With the introduction of a range, the scheduler is free to coalesce your wakeup with any other wakeup that may have happened for other reasons, or at the worst case, fire an interrupt for your upper bound."

This behaviour is ideal for our case and should not impact the performance too much. The reason being that usually multiple blocks are waiting at once. They start sleeping at different times but wait for the same thing. This way the blocks will eventually synchronize as the scheduler will use occurring wakeups for multiple blocks at once.

```
if (compress == ZIO_COMPRESS_OFF && ac_mode != AC_MODE_OFF &&
   ↪ !pio->io_first_child) {
        while (zio_wait_for_first_child(zio, pio)) {
                zfs_sleep_until(gethrtime() + AC_WAIT_TIME);
                zio->io_wait_count++;
                continue;
        }
}
```

Listing 5.13: The secondary blocks are waiting for the tests to finish in the beginning of `zio_write_bp_init`.

Note that not every block will wait in the beginning of `zio_write_bp_init` as shown in listing 5.13. Blocks that deal with metadata are not affected by the adaptive compression logic and will work their way through `zio_write_bp_init` ignoring all tests and waiting.

```
static boolean_t
zio_wait_for_first_child(zio_t *pio)
{
        boolean_t waiting = B_FALSE;
        mutex_enter(&pio->io_lock);
        if (pio->io_first_child_ready == B_FALSE) {
                waiting = B_TRUE;
        }
        mutex_exit(&pio->io_lock);
        return (waiting);
}
```

Listing 5.14: Called by secondary blocks to determine if the first block is ready.

5.4.2 Integer Cost Function

The first block tests three different compression algorithms: LZ4, LZJB and gzip-1. The first algorithm tested is always LZ4. Thereby we make use of LZ4's early abort feature, as it is by far the fasted algorithm to detect incompressible data. If the algorithm detects incompressible data, we will not test the other two algorithms to save time.

Implementing the cost function as planned was not trivial (see Listing 5.15). The reason for that was that it is not allowed to use floating point calculations. They are too slow and as such we were restricted to use integers. Since our cost function contains exponentials we needed a *pow* function. We used *exponentiation by squaring* for this. Since we could not use floating point numbers we used the difference between the logical and physical size instead of the compression ratio. In case of the archive mode, we had to use big integer constants to retain as much accuracy as possible while preventing integer overflows and achieving a large exponential growth.

```
uint64_t
ac_block_compress_cost(hrtime_t t, uint64_t lsize, uint64_t psize,
    uint8_t ac_mode, uint8_t compression)
{
    uint8_t     p = ac_cost_info[compression];
    uint64_t    c = 0;
    uint64_t    d = lsize - psize;

    switch (ac_mode) {
        case AC_MODE_ARCHIVE:
            d = d / DIV_ROUND_CLOSEST(lsize, 10);
            c = (t * p * 1e6) / (ipow(d, 16) / 1e6);
            break;
        case AC_MODE_ENERGY:
            c = (t * p) / d;
            break;
        case AC_MODE_PERFORMANCE:
            c = (ipow(t, 2) * p) / d;
            break;
    }

    return c;
}
```

Listing 5.15: Cost function using only unsigned integers.

Initially we wanted to make use of *Intel*s *Running Average Power Limit* (RAPL) which allows us to obtain power measurements of the whole CPU or individual cores. This would have allowed us to measure how much energy a single compression used. Unfortunately we were not able to read the RAPL counters from within a *Linux* kernel module while it worked perfectly fine from user space. As a consequence, we had to fall back to using constant values for every compression algorithm, based on preliminary measurements.

Summary

This chapter described how operations like create and write in ZFS work. Writing to a file does not imply that its contents are immediately written to disk. That synchronization of buffers and disk happens later. With the understanding of such internal behaviour of ZFS the implementation of file type specific compression and adaptive block compression were explained. Implementing a cost function in the Linux kernel was more complicated than first assumed as floating point operations are not available due to performance constraints.

Chapter 6

Results and Evaluation

This chapter presents the results and evaluation of this thesis. In the first section the used infrastructure together with data collection methods and used test data will be introduced. The following Section 6.2 will show preliminary measurements that were made to get a better understanding of performance and behaviour of compression in ZFS. Section 6.3 will show if and how much overhead the modifications added to ZFS. Section 6.4 will evaluate the effectiveness of adaptive block compression.

6.1 Infrastructure

All tests were conducted on consumer grade hardware with the test data presented in Section 6.1.3. The system has a *Intel Core i5 2500* CPU of the *Sandy Bridge* generation and 8 GB of *DDR3 Random-access memory* (RAM). The operating system (*Ubuntu 14.04*) is installed on a 320 GB HDD. All tests were done on a separate 2 TB drive from *Western Digital*.

To see how the compression of the data is influenced by heavy load on the machine we utilized a *Python* script that multiplies random numbers on one process for every core on the machine. All tests were repeated at least three times and the results averaged.

6.1.1 Data Collection Methods

We wanted to have more information about what happens when data gets compressed in ZFS. The resulting compression ratio as well as the time it took to compress the data is easy to obtain. However, to get further insight, we needed more information and also traced the CPU utilization, power consumption as well as various informations for every block that ZFS writes and compresses. The following sections briefly describe how the data was acquired and correlated.

6.1.1.1 ZFS Logs

To analyse ZFS's internal behaviour we used extensive logging for all relevant compression activities. For every block that enters `zfs_write_bp_init` we now get to know whether it was compressed or not. If it was compressed, we log the algorithm and how long the compression took. Logging is done in binary format into a buffer, controlled by the SPL. The buffer has to be dumped regularly to prevent it from overflowing. On the system we used the logging did not introduce significant overhead.

Understanding how the logging works, and how the logs can be accessed, is, to quote Ned Bass from the ZFS developer mailing list[1], "not at all obvious". Logging in ZFS is done with the call `dprintf(message ...)`. In *Solaris* these logs can be accessed using the tool *dtrace*. Unfortunately that tool is not (yet) available on *Linux* and therefore we have to use a `/proc/sys` interface exposed by the SPL. If we write anything to the file `/proc/sys/kernel/spl/debug`, the logs since the last "dump" will be written to a file in `/tmp` named `spl-log.XXX`. That binary log can then be opened using the `spl` command which is part of the SPL sources.

6.1.1.2 CPU Utilization

The CPU utilization was obtained by sampling the data from `/proc/stat` every 10 ms. The file contains statistics about how much time the CPU spent in user mode, low priority user mode (nice), system and idle. Dividing the sum of the first three through the sum of all entries gives us the CPU utilization.

6.1.1.3 Energy Consumption

To get data about the energy consumption of the CPU we used the *Performance Application Programming Interface* (PAPI), which allows us, among various other things, to read data from *Intel*'s RAPL facility. Originally RAPL is designed to limit the power usage of *Intel* CPUs to, for example, meet power constraints in data centers or similar. However, it also provides registers that contain the energy consumption for reading which are used by PAPI. We used PAPI in the same program that also traces the CPU utilization and therefore the energy consumption is also read every 10 ms.

6.1.2 Correlation of Measurements

To correlate the data from ZFS with other data we used timestamps. In ZFS this is done using `gethrtime()`. The function is part of the SPL and uses the *Linux* function `clock_gettime(CLOCK_MONOTONIC)`. Using the same function in the C program that calculates the CPU utilization and reads the RAPL data allowed us to correlate

[1]https://groups.google.com/a/zfsonlinux.org/forum/#!topic/zfs-devel/S7TRlxlRTfg

the data points. The timestamps from the monotonic clock give us the elapsed time from some arbitrary point in the past, in our case since the system booted.

6.1.3 Test Data

We used five different sets of test data to evaluate how the modified ZFS performs. We wanted to cover different areas of compressibility with the used data — incompressible, good and very good compressibility. In addition to that, we wanted test data that consists of large amounts of files. Our test data consists of the current *Linux* kernel tree (as-is and in a *tar* archive), 500 MB of random data (generated from `/dev/random`) and *NetCDF* data.

`linux` 637 MB comprised of 47971 files (average file size ~14 kB).

`linux.tar` Uncompressed *tar* archive of the *Linux* kernel.

`random.dat` 500 MB of incompressible, random data.

`TP04` 10 GB of *NetCDF* data.

`mixed.dat` Combination of `linux.tar` and `random.dat`.

The reason for using the *Linux* source tree as-is and in a *tar* archive is that this nicely demonstrates how the same data (with the same size) can create completely different results. The overhead of processing that many files in ZFS is enormous and not only increases the runtime by a large margin, it also heavily influences the resulting compression ratio.

6.2 Preliminary Measurements

This section presents measurements that were made with the default version of ZFS. Section 6.2.1 analyses the nine different available compression levels of gzip. In Section 6.2.2 we have a look at the influence of the selected block size on compressibility with selected algorithms. Section 6.2.3 shows the impact of high system load on the compression speed and energy consumption.

6.2.1 Study of GZIP's Compression Levels

This section presents measurements of the nine different compression levels that gzip offers. Those nine levels are also available in ZFS, currently implemented in the form of nine different compression algorithms. In the future though, ZFS will offer a compression level property since other algorithms also offer levels. If no level is specified (`zfs set compression=gzip` as opposed to `zfs set compression=gzip-X`), ZFS will choose level six.

6.2.1.1 Compression of `linux.tar`

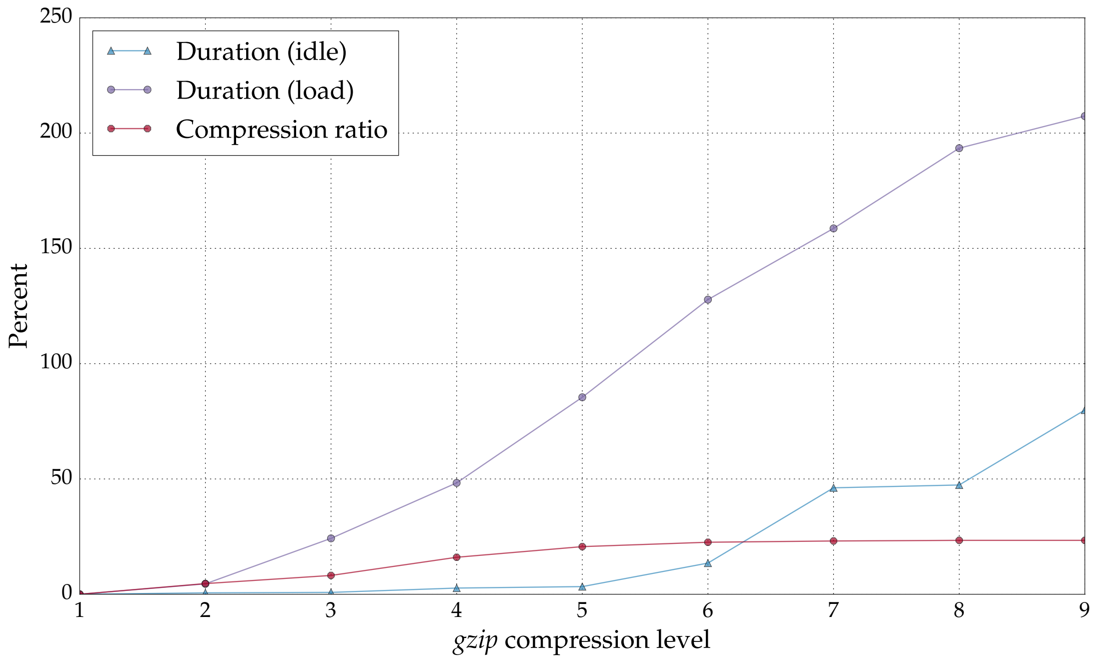


Figure 6.1: Comparison of the nine different gzip compression levels (% increase compared to level 1) when compressing `linux.tar`.

In Figure 6.1 we can see how the time during load (100% CPU utilization), during idle (nothing else is happening on the machine, apart from the compression) as well as the compression ratio increases together with the compression level. In this example, the very good to compress file `linux.tar` was tested. Despite its good compressibility, increasing the level from one to nine only increases the achieved compression ratio by around 25%. This is not a negligible amount, but at the same time we can see a big discrepancy when we look at the time needed to compress the file. Compared to the ratio increase of 25% the runtime increase of 80% on level nine is huge. When the system is under heavy load it gets even worse. Instead of 80%, level nine needs over 200% longer to compress the file.

Interestingly, the last level where the increase in compression ratio is higher than the runtime increase, is level six, the default level of ZFS. However this is not the case under load.

6.2.1.2 Compression of `random.dat`

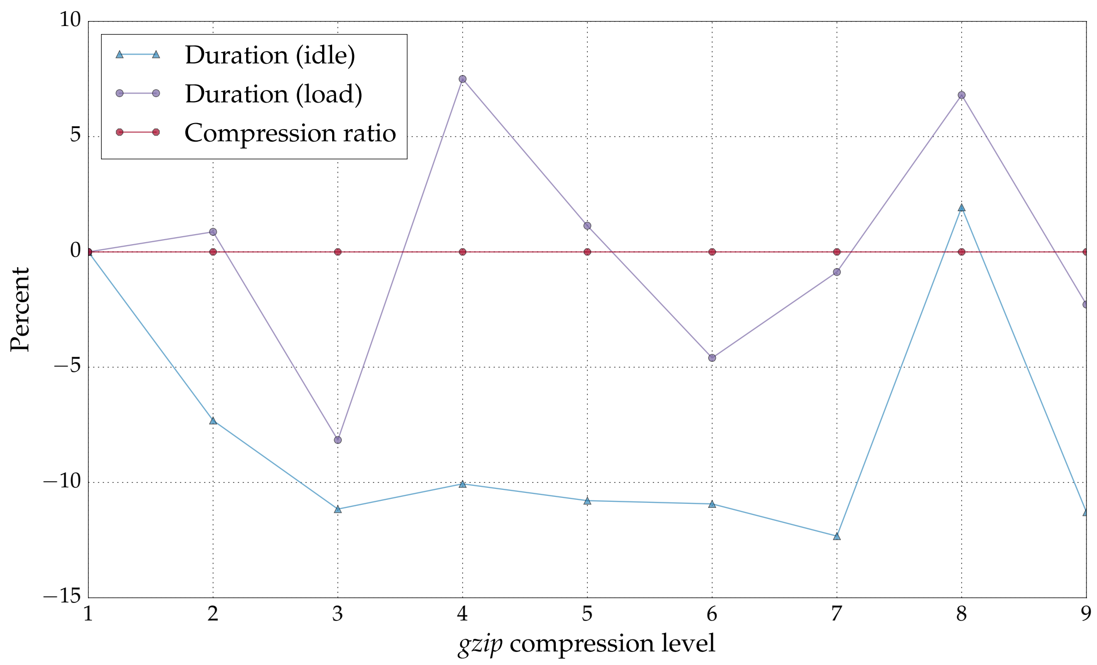


Figure 6.2: Comparison of the nine different gzip compression levels (% increase compared to level 1) when compressing `random.dat`.

In Figure 6.2 we can see how the gzip levels compare when trying to compress incompressible data. As we can see, the differences between the levels range between +10 and -15%. It can be concluded, that increasing the gzip level has no noteworthy impact when compressing incompressible data.

6.2.2 Influence of Block Size on Speed and Compression Rate

In this section we will examine the impact of block size in ZFS on compression. By default, the block size is set to 128 kB. ZFS allows block sizes between 512 B and 128 kB. The next valid block size is always twice as much as the previous one, e.g. 1 kB, 2 kB, 4 kB etc.

6.2.2.1 Compression of `linux.tar`

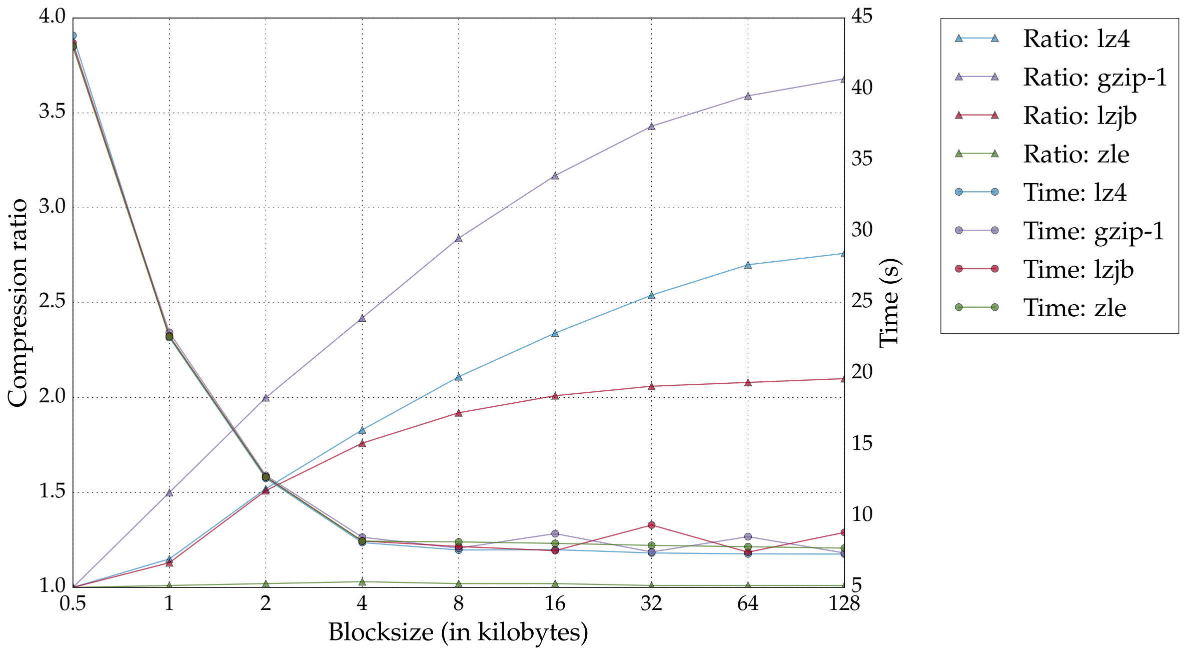


Figure 6.3: Impact of block sizes when compressing `linux.tar`.

In Figure 6.3 we can see the compression of `linux.tar` with all different block sizes using LZ4, ZLE, LZJB and gzip-1. The smaller block sizes (0.5 kB – 2 kB) thereby seem to create a huge overhead, as the runtime is significantly higher than for the remaining block sizes. For this test file, the block sizes between 4 kB and 128 kB have equal performance for all used algorithms with minor variations. As the file is very good to compress, increasing the block size also increases the compression ratio significantly (apart from ZLE). The reason is that the LZ77-based algorithms can build bigger dictionaries that way to look for redundant data.

6.2.2.2 Compression of `linux`

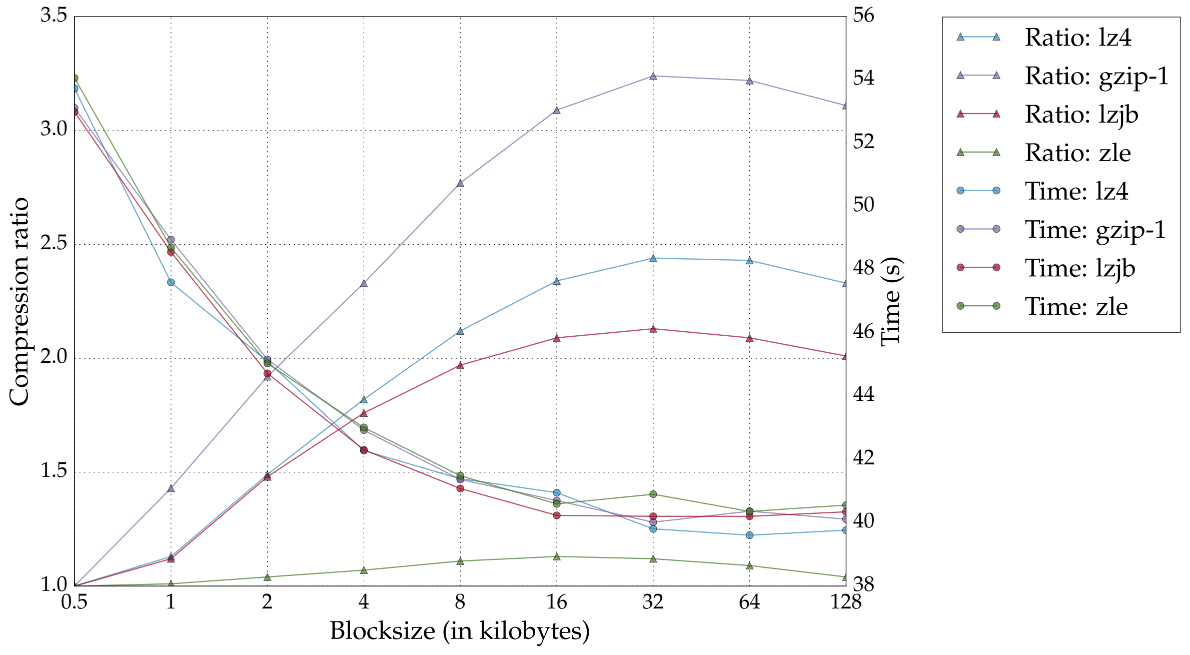


Figure 6.4: Impact of block sizes when compressing `linux`.

In Figure 6.4 we can see the compression of (essentially) the same data, just not in a *tar* archive. In that case (with more than 50000 files of an average 14 kB size), we can observe a sweet spot at around 32 kB. We can also see that the complexity of compressing that many files outweighs the overhead of smaller block sizes to some degree. In the previous example the difference between a block size of 512 B and 128 kB was ~43 seconds to ~7 seconds. Now the difference is just ~53 seconds to ~40 seconds.

If we factor in both the achieved compression ratio, and the needed runtime, it can be concluded that 128 kB is the best block size (for the used data), despite (for some reason) the block sizes 64 and 128 being slightly worse in terms of compression ratio in the second example.

6.2.3 Impact of High Load on Write and Compression Speed

This section presents measurements of all available compression algorithms in ZFS (including turned off compression). The focus lies on the consumed energy. All tests were conducted during load (100% CPU utilization) and idle.

6.2.3.1 Compression of `linux.tar`

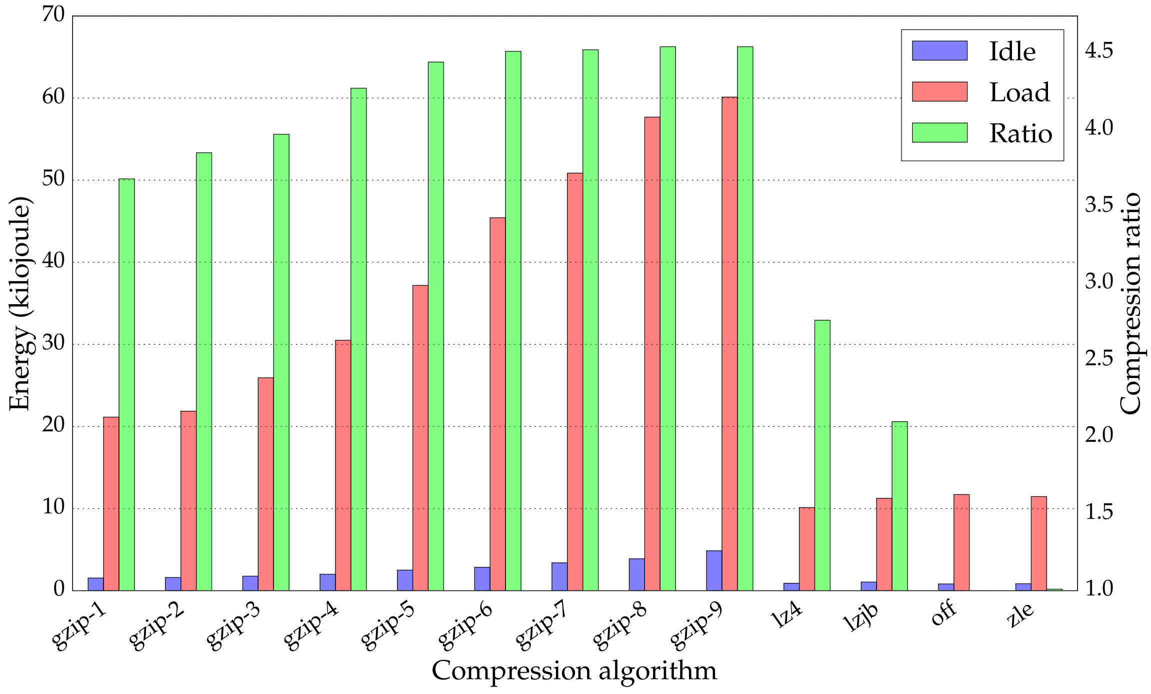


Figure 6.5: Energy spent compressing `linux.tar` under load and idle.

In Figure 6.5 we can see the energy consumption under full system load and idle, as well as the achieved compression ratio, for all available compression algorithms when compressing `linux.tar`. It can be seen that gzip achieves a significantly higher compression ratio, across all levels, than the other algorithms. After level six though, the ratio no longer increases considerably.

Looking at the energy consumption of gzip during system load we can see a steady increase of five to ten kilojoule between levels two and eight. At the beginning (level one to two) and in the end (level eight to nine) the increase is not that high. The consumed energy increases regardless of the achieved compression ratio, more spent energy does not translate to a higher ratio.
The energy consumption during idle does not look as drastic. Especially when comparing the first levels of gzip to LZ4 and LZJB, spending the extra effort looks very promising. Although the energy consumption increases we get a large increase in

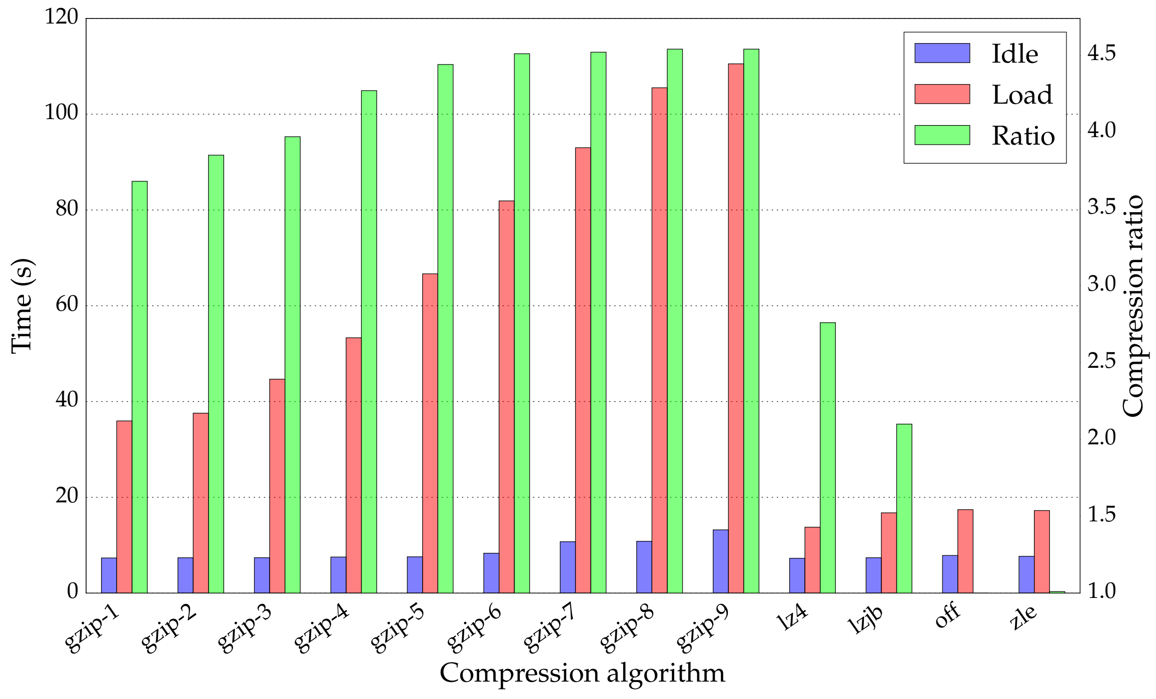


Figure 6.6: Time spent compressing `linux.tar` under load and idle.

compression ratio in return. In Figure 6.6 we see the data from the same benchmark, only this time how much time the compression took instead of the spent energy. In most aspects the time and spent energy correlate one to one. However, when looking at gzip-1, LZ4 and LZJB, we can see that they all finished faster than the run with compression turned off. Apart from gzip-1, the algorithms even finished faster when the system was under load.

We can conclude that for this test data, gzip-1 offers great compression ratios. When the system is idle it is even faster to compress-and-write than just to write the data uncompressed. During load however, gzip increases the runtime significantly whereas LZ4 and LZJB still finish faster than the uncompressed run.

6.2.3.2 Compression of `random.dat`

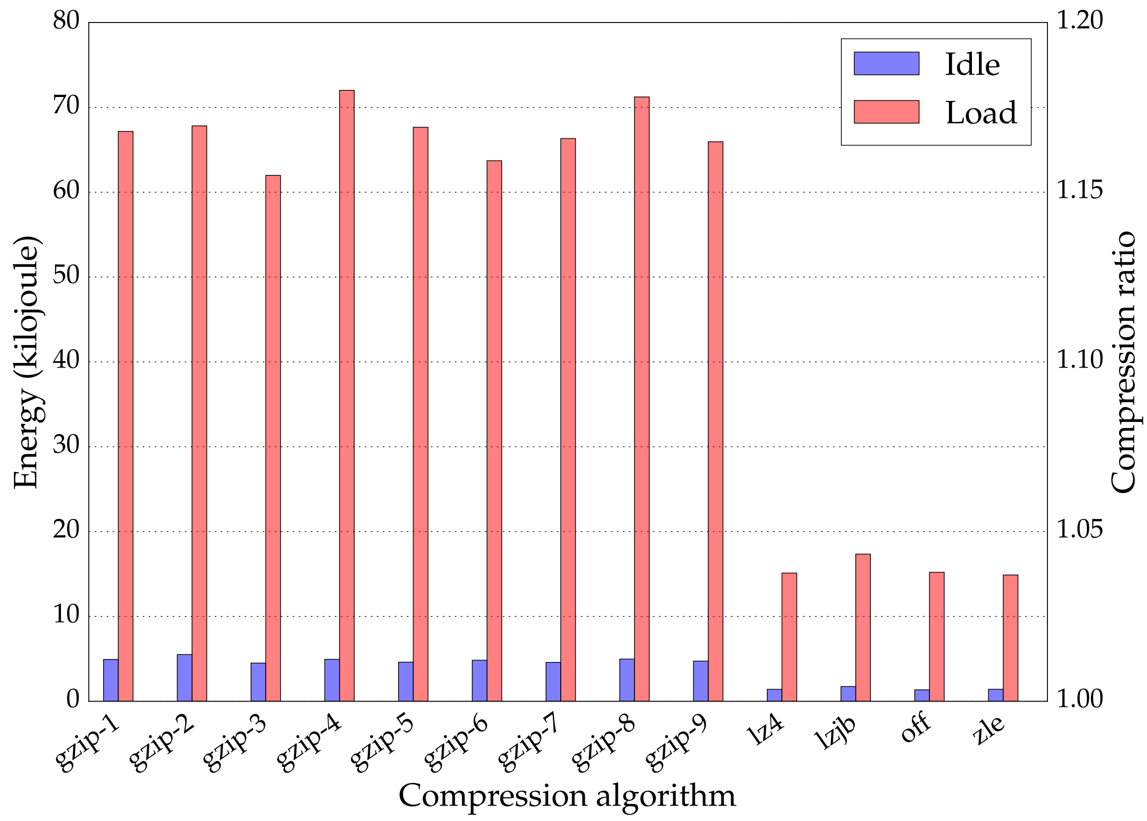


Figure 6.7: Energy consumption of trying to compress `random.dat`

How algorithms fare when working on incompressible data is very important. In Figure 6.7 we can see how much energy the different algorithms consumed trying to compress `random.dat`. As pointed out in Section 6.2.1 there is no big difference between the gzip levels regarding compression time. This translates one to one into the consumed energy. The other algorithms do significantly better, both under load and idle. Both LZ4 and ZLE do not create a measurable overhead, only LZJB is slightly slower than no compression under load. All gzip levels use more than twice the energy during idle and more than three times the energy during load compared to the rest. This is very significant and really boots out gzip as a viable “always on” compression algorithm for any kind of data.

6.2.3.3 Compression of `linux`

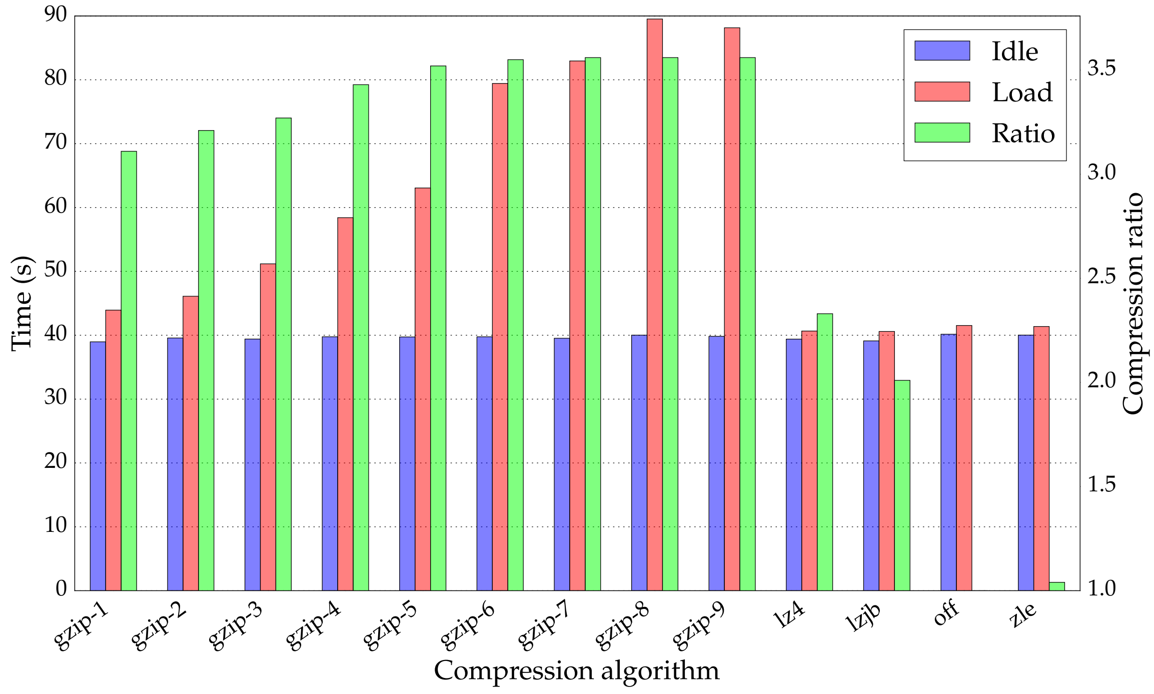


Figure 6.8: Time spent compressing `linux` during load and idle.

In Figure 6.8 we can see the *Linux* source tree being compressed by all available algorithms. As it is comprised of over 50 thousand files, ZFS has a lot more to do compared to writing (and compressing) the *tar* archive of it. The overhead of this is so huge, that we do not see any difference in compression time between the algorithms (during system idle). We basically get the compression ratio for "free" (time wise). It is astonishing that there is not even a difference between gzip-9 and the rest, the level which performed much slower in the other tests. During load we start to see a difference, gzip-1 is already slightly slower with the other levels getting increasingly slower. Still, gzip-9 took "only" twice the time of LZ4.

In Figure 6.9 we can see the energy consumption of this compression. This shows us that, again, the time spent compressing does not translate one to one to the energy cost. Although gzip does not take longer than the rest during idle, it uses more energy (while achieving a larger compression ratio). Both LZ4 and LZJB use approximately the same energy as the run with compression turned off, while still offering huge compression ratios of 2.4 and 2.0, respectively.

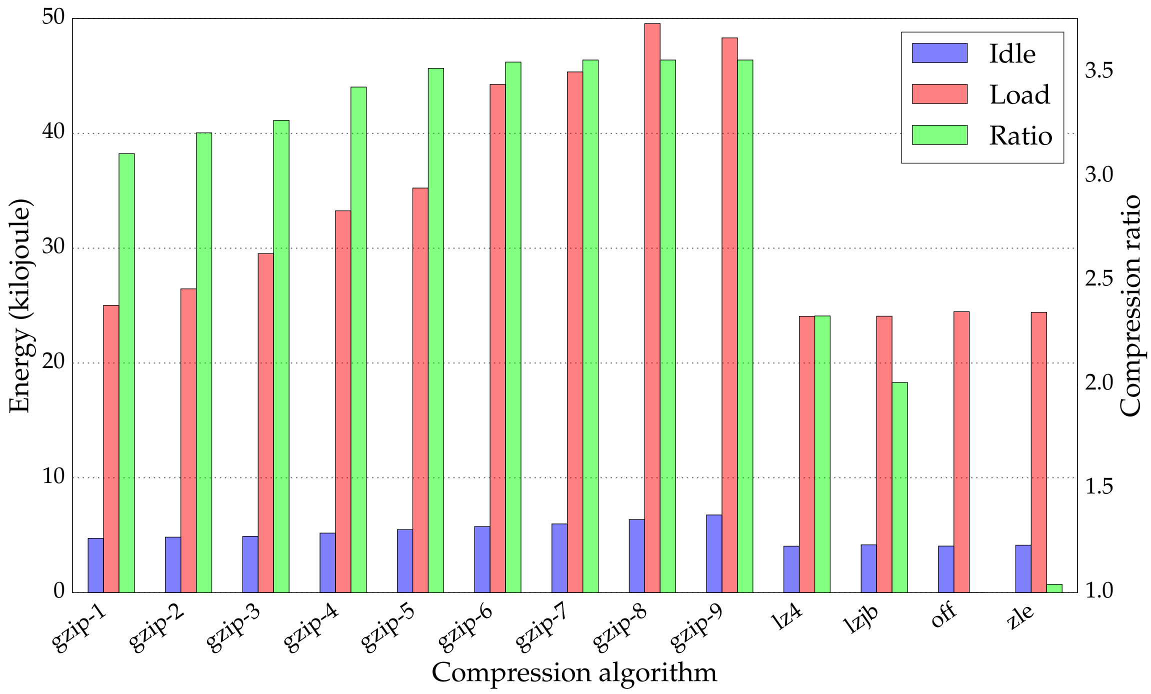


Figure 6.9: Energy consumption of compressing `linux` during load and idle.

6.3 Added Overhead

This section will analyse how much overhead the changes to ZFS created. Both the detection of the file type and the block testing result in more work being done before a block is finally written to disk. The question is how much overhead this creates. If the overhead is too large it may outweigh the advantages.

6.3.1 File Type Detection

To test the overhead of the file type detection we used the `linux` dataset. We compressed it once with gzip-1 and compression mode set to off. This test represents the normal behaviour of ZFS without any changes. To get a clean, comparable test with compression mode set to on, we renamed all files in the dataset to have the same extension. Then we set the compression algorithm for that extension to gzip-1 and compressed the dataset again. We compare the time needed to compress and the consumed energy. The compression ratio is obviously equal. As we can see in Table 6.1 there is no noteworthy overhead.

		Default		Adaptive	
Test	**Utilization**	**Time**	**Energy**	**Time**	**Energy**
Extension	idle	33.91 s	4530 J	34.20 s	4580 J
Extension	load	42.02 s	24 587 J	41.98 s	25 055 J
Block testing	idle	150.09 s	14 582 J	150.37 s	15 538 J
Block testing	load	174.29 s	94 790 J	192.41 s	104 107 J

Table 6.1: Overhead due to testing blocks or passing through the file extension.

6.3.2 Block Testing

To test the overhead of the block testing, and the inevitably introduced wait times, we used the *NetCDF* dataset `TP04` because of its bigger size. All files were renamed to an unknown extension to make sure that the file type detection does not prevent the block testing. As this method introduces (sometimes, not always) waits, it is expected to have some overhead. During idle, the runtime is essentially equal while the additional work introduces a slight increase in energy consumption of 6.5%. When the system is under heavy load (100% utilization on all cores) the runtime, and energy consumption, increases by considerable 10%. This is something that has to be kept in mind albeit not surprising since testing blocks with compression algorithms before deciding on one does not come for “free”.

6.4 Adaptive Block Compression

In this section we will investigate how the adaptive block compression worked on our test data. The primary metric will of course be the achieved compression ratio. In addition to that we will look at the consumed energy. The used energy gives us more insight than the used time. Often the time will be equal, yet more energy has been consumed since significantly more work has been done.

6.4.1 Compression of `linux.tar`

Compression of `linux.tar` results in a total of 4429 blocks being compressed by ZFS. Four of those blocks are not compressible. The archive mode compresses 3875 of those blocks using gzip-1 and 640 with LZ4. The result is a compression ratio of 3.52 with an energy consumption of 1.48 kJ which we can see in Figure 6.10. The complete gzip-1 compression resulted in a ratio of 3.68 with an energy consumption of 1.54 kJ. Under load the archive mode saves around 3 kJ since it compresses more blocks using LZ4.

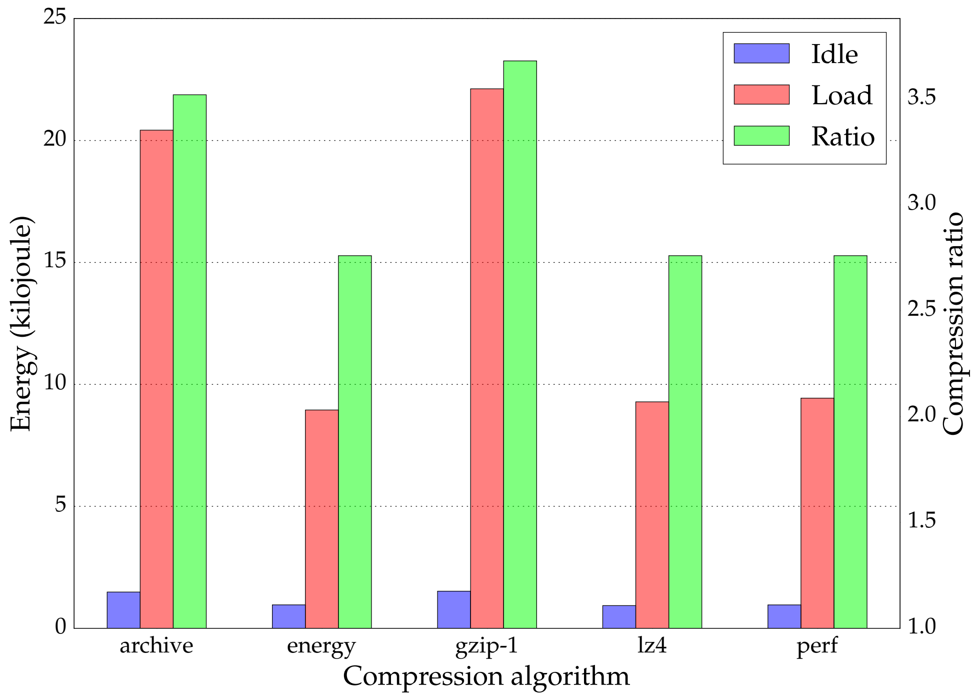


Figure 6.10: Comparison of adaptive block compression with gzip-1 and LZ4 when compressing `linux.tar` under load / idle.

The other modes, energy and performance, look like the LZ4 run. And indeed both compressed all blocks using LZ4.

6.4.2 Compression of `linux`

The complete `linux` test dataset consists of over 50 thousand files. In ZFS this results in 48358 blocks, more than 10 times the amount of the *tar* archive of the same data. Although the actual data of this test is very good to compress (demonstrated by the previous test), LZ4 still disabled the compression for 8048 blocks because it categorized them as incompressible. On the other hand, gzip-1, which has no early-abort feature like LZ4, turned off the compression for only 5787 blocks. Compression will be turned off in ZFS (regardless of the chosen algorithm) if less then 12.5 % were compressed. Since the adaptive block compression makes use of LZ4's early abort this results in a turned off compression for more blocks. As such, the archive mode has turned off compression for 13022 blocks, energy 14088 and performance even 14105. This results in compression ratios for the adaptive modes being worse than expected as we can see in Figure 6.11.

From this test we can conclude that for a dataset with many files being much smaller than the ZFS block size of 128 kB, using the LZ4 early abort squanders potential.

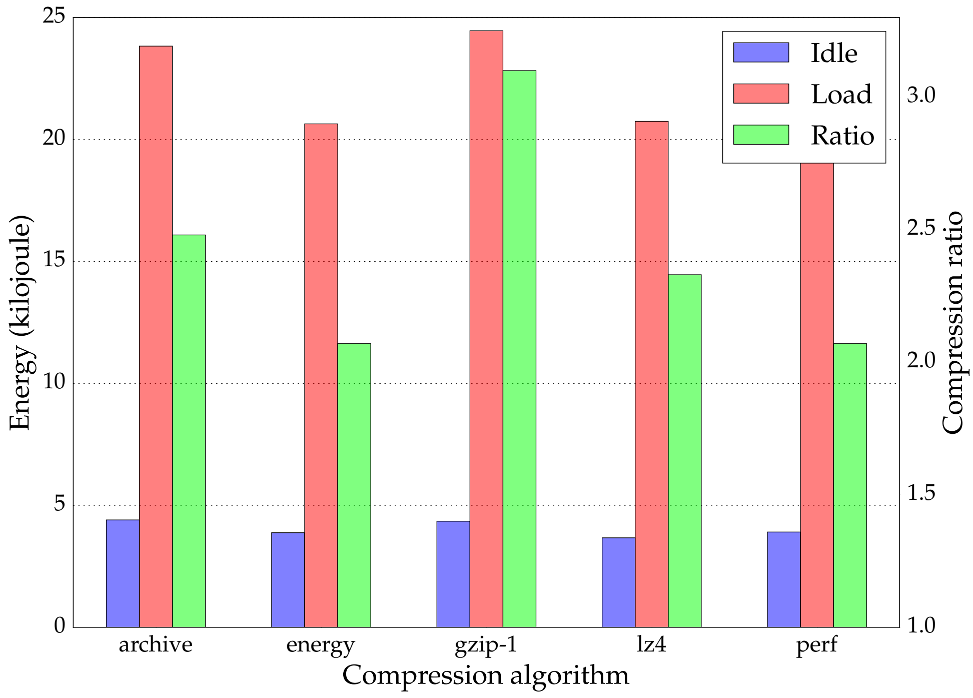


Figure 6.11: Comparison of adaptive block compression with gzip-1 and LZ4 when compressing `linux` under load / idle.

This can be circumvented by not relying on LZ4 to detect incompressible data, but instead gzip-1 and testing whether we compressed 12.5 %. As gzip is much slower when working on incompressible data this would however mean a significantly worse performance for adaptive block compression.

6.4.3 Compression of `mixed.dat`

The `mixed.dat` dataset is an artificial example constructed to demonstrate the usefulness of adaptive block compression. It contains 50 % of incompressible data (`random.dat`), a big obstacle for gzip-1. The rest of the dataset consists of `linux.tar`. We can see in Figure 6.12 that the archive mode achieves almost the same compression ratio as gzip-1 while conserving significant amounts of energy, both under load and idle. Under load the archive mode used 28 kJ to achieve a compression ratio of 1.6, while gzip-1 used 51 kJ. During system idle the difference is 2.2 kJ (archive) compared to 3.8 kJ (gzip-1). The modes energy and performance also detect the incompressible data and turn off compression for those blocks.

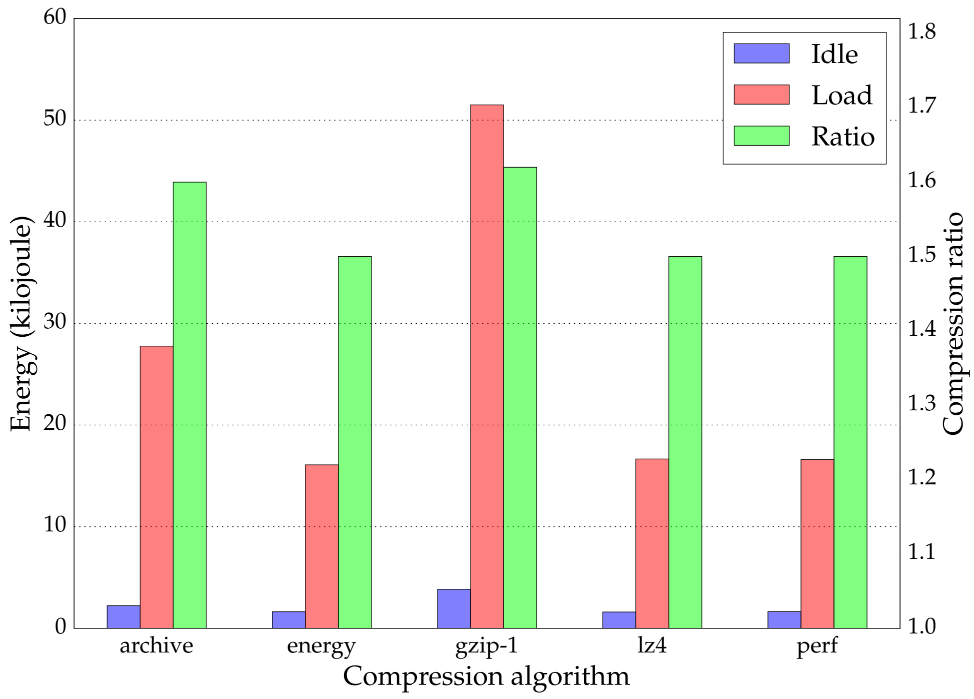


Figure 6.12: Comparison of adaptive block compression with gzip-1 and LZ4 when compressing `mixed.dat` under load / idle.

6.4.3.1 Block Compression Visualization

To see more than just numbers after the benchmark finished we visualized when blocks are compressed, how long the compression takes and along with that the current CPU utilization and energy consumption. The `mixed.dat` data is especially well suited for this as it allows us to see just how drastic the difference between good-to-compress and incompressible data is, and how much extra work that means for gzip.
In Figure 6.13 we can see `mixed.dat` being compressed by just gzip-1. A blue dot represents an actual data block being compressed, while the red dots are meta data. The position on the left y-axis shows us how long the compression of that block took. The right y-axis shows us the CPU utilization in percent and the current power consumption in watt. The first half of the graph visualizes the compression of the *tar* archive of the *Linux* kernel, the second half is the compression of `random.dat`.

As we can see, the compression in the first half takes between 1 and 2 ms for every data block. An incompressible data block in the second half needs 3 ms and on top of that working on incompressible data also results in a higher energy consumption as the CPU utilization goes up to 100 % during compression.

In Figure 6.14 we can see the same data being compressed using the archive mode. The majority of blocks in the first half is again between 1 and 2 ms, compressed by gzip-1. Some blocks are way above that between 2 and 6 ms — blocks that had to wait.

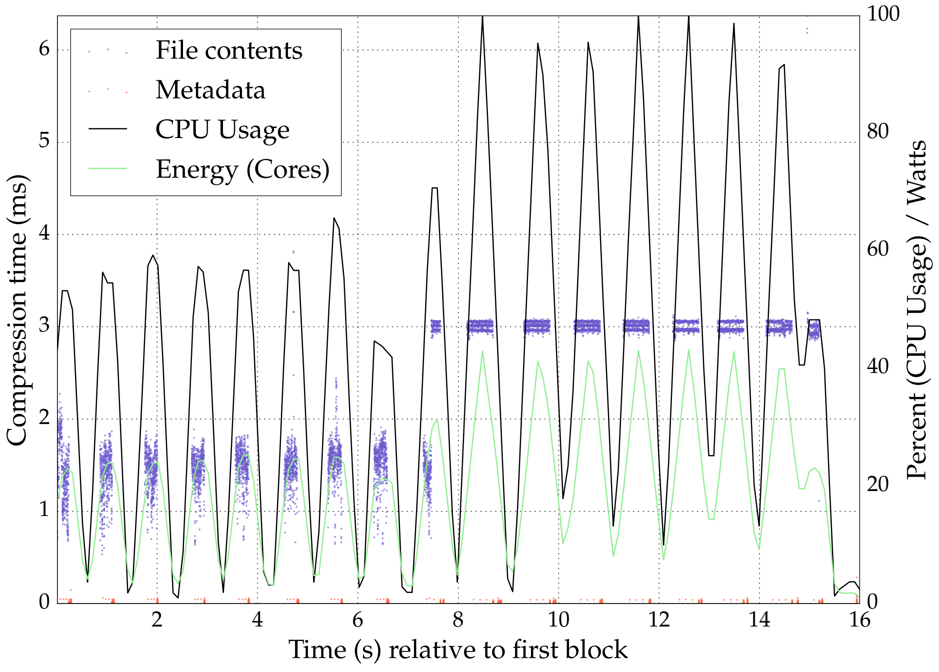


Figure 6.13: System utilization (CPU, power) and block compression times when compressing `mixed.dat` with gzip-1.

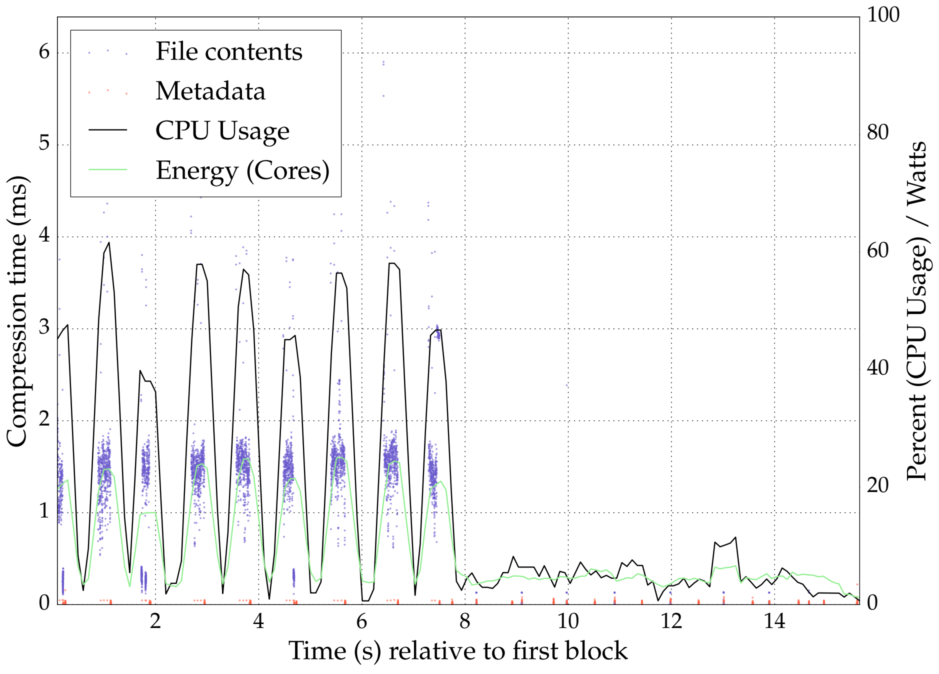


Figure 6.14: System utilization (CPU, power) and block compression times when compressing `mixed.dat` with adaptive mode archive.

At the 0, 2 and 5 second mark we can also see some blocks finishing between 0.2 and 0.5 ms. Those were compressed by LZ4 which is also reflected in the CPU utilization and power consumption at that time.

The second half shows us how huge the impact of compression can be compared to writing blocks uncompressed. The CPU utilization almost never goes above 10 % and the power consumption stays below 10 W.

This example demonstrated how archive mode can be used to still benefit of the better compression ratios of gzip while avoiding the enormous overhead of gzip when working on incompressible data.

6.4.4 Compression of `TP04`

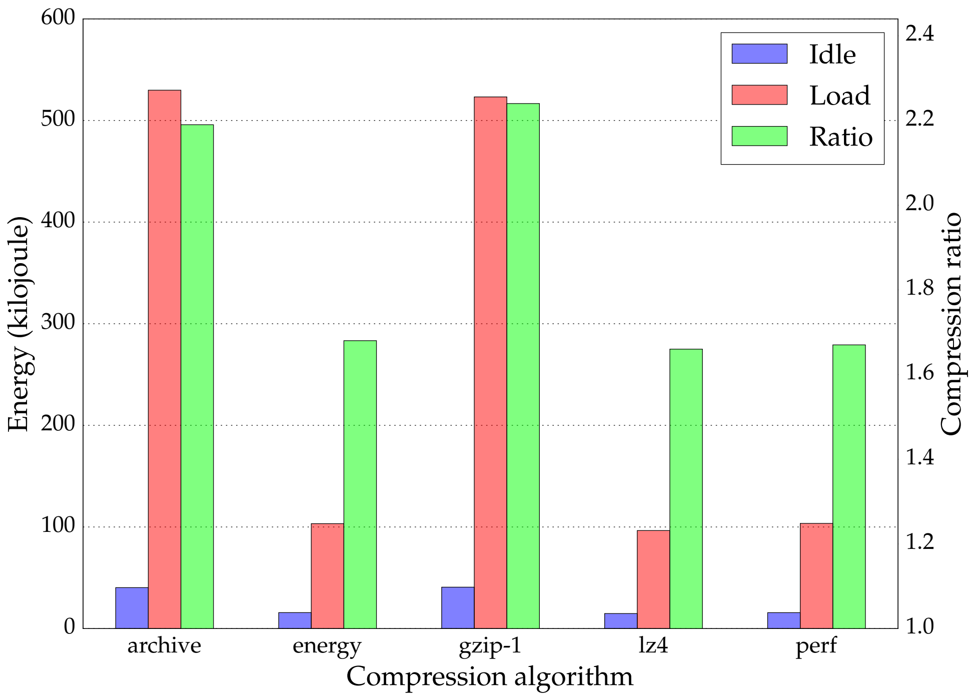


Figure 6.15: Comparison of adaptive block compression with gzip-1 and LZ4 when compressing `TP04` under load / idle.

The dataset `TP04` is the biggest of all with a size of 10 GB. In Figure 6.15 we can see the compression using all adaptive modes as well as gzip-1 and LZ4. In ZFS `TP04` is represented by total of 81697 blocks. The archive mode compresses 70191 with gzip-1, 1388 with LZJB, 8455 with LZ4 and turned off the compression for 1663 blocks. The resulting compression ratio is 2.2. During idle the compression consumed 40 kJ. During idle gzip-1 consumed 40.5 kJ and achieved a ratio of 2.24.

Summary

This chapter has shown measurements made both with the original and modified version of ZFS. The preliminary measurements showed that larger block sizes are preferable as it gives compression algorithms the possibility to build bigger dictionaries to identify redundant data. On top of that smaller block sizes also create considerable overhead. The different levels of gzip offer, to some degree, better compression ratios at the expense of significant runtime increase. When working on incompressible data gzip is significantly slower than all other algorithms implemented in ZFS.

The modifications of ZFS allowed us to make use of gzip in situations where the default version of ZFS would experience a severely worse performance. Despite adaptive block compression introducing a slight overhead its usefulness can be justified by the fact that its cost function reliably detects different kinds of data and reacts accordingly. This makes it possible to achieve significantly higher compression ratios while avoiding the huge performance penalties that algorithms like gzip bring along. File type specific compression introduced no overhead at all and as such is a very well suited addition to ZFS as it allows the user to realize a compression scheme that would not be possible otherwise.

Chapter 7

Conclusion and Future Work

This chapter concludes the thesis. The results of the designed and implemented new features will be summarized. Furthermore, ideas for future work that were out of scope for this thesis will be presented.

This thesis focused on the problems of compression in the file system layer. In other layers than the file system, compression algorithms such as gzip are frequently used to squeeze out every last bit of compression ratio. However, file systems usually prioritize performance over compression ratios and as such they use algorithms like LZ4 that yield good ratios while barely impacting, or even improving, the overall performance. Although algorithms like gzip are available, the lack of flexibility in the compression settings makes them an unattractive choice. Choosing an algorithm implies that everything gets compressed with that algorithm — including incompressible data.

The preliminary measurements showed that gzip can be a legimitate choice as a file system compression algorithm. When the system is not under full load and the data is compressible, gzip will generally finish as fast as LZ4 and LZJB while achieving better compression ratios. However, if confronted with incompressible data gzip will need up to 15 times longer than LZ4 for a single block. Under load gzip will generally be slower than LZ4 and LZJB since gzip needs more CPU resources and as such is affected more by the higher load.

In the course of this thesis two features were added to ZFS to add flexibility and to provide a way to make use of gzip while avoiding some of the performance penalties it brings along. In addition to that a new property was added which allows the user to specify a use case that further influences the selection of an algorithm.

The flexibility in terms of selecting a compression algorithm was increased with the implementation of file type specific compression. It allows the user to set the best compression algorithm for a specific file type if prior knowledge about it exists. Such knowledge could for example be that the file has already been compressed in user space and compressing it again would be worthless. In that case the compression can be disabled for that file type. Other files may show different degrees of compressibility with different algorithms being better than others. The more the user knows about the data used on the system, the better the overall compression can be. As determining the file type only happens once, when the file is created, the feature adds no noteworthy overhead. This procedure has one main weaknesses. It does not work on files that have no extension as that is the only way to determine the file type.

To negate weaknesses or account for strengths of some algorithms, adaptive block compression was implemented. It works with a selection of algorithms and tests the first out of a batch of blocks (which are part of a write operation) for compressibility. The compressions result in several parameters that are subsequently used to decide which of the tested algorithms is going to be used until the next test is scheduled. The parameters used are the time a compression needed, the difference in physical and logical size and constant values for power consumption that account for the higher demands of algorithms like gzip. The parameters result in a cost value for each compression where the algorithm with the lowest cost wins. This, on the one hand, works very well on large files that have differing compressibility and, on the other hand, also accounts for changing system load. If, in the beginning of the file, the system load is very low, algorithms like gzip may win. If towards the end of the file the load becomes higher gzip likely takes longer because of its higher demands and as a result will loose because of too high cost.

To further increase the flexibility a new property called compression mode was implemented that works in conjunction with adaptive block compression and file type specific compression. It allows the user to specify a use case which may be archive, performance or energy. Archive mode strives for the best compression ratio, performance mode for the least decrease in file system throughput and energy represents a tradeoff. For file type specific compression this gives the user the ability to set different compression algorithms for the different modes for each file. In adaptive block compression this weighs different parameters more than others. For example, in archive mode the compression ratio is much more important than the needed compression time, whereas in performance it is the other way around.

Future Work

The following sections present ideas for future work. This includes evaluation of a new compression algorithm once it is implemented in ZFS as well as several improvements to the current implementation.

Other Compression Algorithms

The evaluation of the changes made in this thesis was limited to the choice of algorithms implemented in ZFS at that time. The differences between the present algorithms are significant with LZ4 being up to 20 times faster than gzip under certain circumstances. Recently, a new algorithm was developed, *Zstandard*[1]. It was developed by the developer of LZ4 and can be placed between LZ4 and gzip. It aims for higher compression ratios and still reaches half of LZ4's speed. It would be very interesting to see how the algorithm performs as part of the algorithms used in adaptive block compression.

Implementation

The current implementation is only prototypical as it requires the user to recompile ZFS in order to change the algorithms used for adaptive block compression or the lookup table for file type specific compression. Ideally both should be configurable with ZFS properties. Section 3.2.3 presented how this could be done for file type specific compression. The set of algorithms used for adaptive block compression could be represented as a simple, ordered list.

```
zfs set adaptive_algoritms lz4:lzjb:gzip-1
zfs set adaptive_algoritms gzip-1:lzjb:lz4
```

Ordered, because the testing ends if the first algorithm classifies the data as incompressible. Incompressible is all data that does not get compressed by at least 12.5 %. LZ4 classifies more data as incompressible than gzip since gzip reaches better compression ratios. This way we can control both the choice of algorithms and the overall performance with one property. If we want faster choices we use LZ4 as the first algorithm because of the fast early abort it conducts. If we want better compression ratios we use gzip as the first algorithm.

Summary

This chapter summarized the results of this thesis. The lack of flexibility when choosing how to compress data in ZFS has been circumvented with the implementation of two new features. Tasks for future work have been presented to improve the prototypical implementation and to evaluate and integrate new compression algorithms by the time they are implemented in ZFS.

[1]`http://fastcompression.blogspot.de/2015/01/`

Bibliography

[1] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The Zettabyte File System. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, 2003.

[2] L. P. Deutsch. Deflate compressed data format specification version 1.3, 1996.

[3] R. Elling. ZFS tutorial, June 2009. Slides of a presentation given at USENIX '09, June 14–19, University of California, San Diego, Chicago.

[4] GZIP is not enough! `https://www.youtube.com/watch?v=whGwm0Lky2s`, Oct. 2013. [Online; accessed 3-February-2015; Minute 14 – 16].

[5] D. A. Huffman et al. A method for the construction of minimum redundancy codes. *proc. IRE*, 40(9):1098–1101, 1952.

[6] Delays - Information on the various kernel delay / sleep mechanisms. `https://www.kernel.org/doc/Documentation/timers/timers-howto.txt`. [Online; accessed 3-February-2015].

[7] N. Kimura and S. Latifi. A survey on data compression in wireless sensor networks. In *Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on*, volume 2, pages 8–13. IEEE, 2005.

[8] C. Krintz and S. Sucu. Adaptive on-the-fly compression. *Parallel and Distributed Systems, IEEE Transactions on*, 17(1):15–24, 2006.

[9] LZ4 compression. `http://wiki.illumos.org/display/illumos/LZ4+Compression`, Feb. 2013. [Online; accessed 9-January-2015].

[10] LZ4 — Extremely Fast Compression Algorithm. `https://code.google.com/p/lz4/`. [Online; accessed 11-February-2015].

[11] LZ4-HC: High Compression LZ4. `http://fastcompression.blogspot.de/2011/09/lz4-hc-high-compression-lz4-version-is.html`, Sept. 2011. [Online; accessed 21-February-2015].

[12] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[13] N. Megiddo and D. S. Modha. One up on LRU. *login–The Magazine of the USENIX Association*, 28:7–11, 2003.

[14] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top 500 list. *Electronically published at* `http://www.top500.org/lists/2010/11`, 2014.

[15] S. Microsystems. ZFS on-disk specification draft. *Technical Report, Sun Microsystems*, 2006.

[16] OpenVPN Source Code. `https://github.com/OpenVPN/openvpn/blob/release/2.3/src/openvpn/lzo.c`. [Online; accessed 19-February-2015].

[17] OpenZFS History. `http://open-zfs.org/wiki/History`, May 2014. [Online; accessed 6-January-2015].

[18] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[19] C. N. Taylor and S. Dey. Adaptive image compression for wireless multimedia communication. In *Communications, 2001. ICC 2001. IEEE International Conference on*, volume 6, pages 1925–1929. IEEE, 2001.

[20] D. Vatolin, I. Seleznev, and M. Smirnov. Lossless video codecs comparison 2007. *Inf. téc., Graphics & Media Lab (Video Group) of Moscow State University*, 2007.

[21] B. Welton, D. Kimpe, J. Cope, C. M. Patrick, K. Iskra, and R. Ross. Improving i/o forwarding throughput with data compression. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 438–445. IEEE, 2011.

[22] Wikipedia. Festplattenlaufwerk – Geschwindigkeit. `http://de.wikipedia.org/wiki/Festplattenlaufwerk#Geschwindigkeit`. [Online; accessed 11-February-2015].

[23] Wikipedia. Mark Kryder – Kryder's Law. `http://en.wikipedia.org/wiki/Mark_Kryder#Kryder.27s_Law`. [Online; accessed 11-February-2015].

[24] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5):757–768, 1999.

[25] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

Appendices

List of Acronyms

ACE	Adaptive Compression Environment
API	Application Programming Interface
ARC	Adaptive Replacement Cache
Btrfs	B-tree File system
CPU	Central Processing Unit
DMU	Data Management Unit
DSL	Dataset and Snapshot Layer
DVA	Data Virtual Address
EU	European Union
ext	Extend File System
gzip	GNU zip
HDD	Hard disk drive
HFS+	Hierarchical File System Plus
HTML	Hypertext Markup Language
HTTP	Hypertext Transport Transfer Protocol
ID	Identifier
LFU	Least Frequently Used
LLOC	Logical Lines of Code
LRU	Least Recently Used
LVM	Logical Volume Manager
LZ77	Lempel-Ziv Compression
LZMA	Lempel-Ziv-Markov chain algorithm
LZO	Lempel-Ziv-Oberhumer
LZRW	Lempel-Ziv Ross Williams
NTFS	New Technology File System
NWS	Network Weather Service
PAPI	Performance Application Programming Interface
QL	Quantization Level
RAID	Redundant Array of Independent Disks
RAM	Random-access memory
RAPL	Running Average Power Limit
RLE	Run-Length Encoding
SCP	Secure Copy
SFTP	Secure File Transfer Protocol
SPA	Storage Pool Allocator

SPL	Solaris Porting Layer
SSH	Secure Shell
TCO	Total Cost of Ownership
UMTS	Universal Mobile Telecommunications System
URE	Unrecoverable Read Error
VBS	Virtual Block Size
VDEV	Virtual Device Driver
VFS	Virtual File System
VPN	Virtual Private Network
WWW	World Wide Web
ZAP	ZFS Attribute Processor
ZFS	Zettabyte File System
ZIL	ZFS Intent Log
ZIO	ZFS Input / Output Framework
ZLE	Zero-Length Encoding
ZPL	ZFS POSIX Layer

List of Figures

List of Listings

List of Tables

Eidesstattliche Versicherung

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

Ort, Datum	Unterschrift