Data Reduction Techniques

— Seminar "New trends in HPC"—
— Project "Evaluation of parallel computers" —

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

Vorgelegt von:	Kira Isabel Duwe
E-Mail-Adresse:	0duwe@informatik.uni-hamburg.de
Matrikelnummer:	6225091
Studiengang:	Master Informatik
Betreuer:	Michael Kuhn

Hamburg, 05. September 2016

Abstract

Due to the ever-present gap between computational speed and storage speed every computer scientist is confronted with a set of problems regularly.
The storage devices are not capable of keeping up with the speed the data is computed with and needs to be processed with to fully exhaust the possibilities brought up by the computational hardware.
In High Performance Computing (HPC) storage systems are gigantic and store around two-digit numbers of PB while they need to cope with throughput around TB/s.
So, for this field of research it is even more important to find solutions to this growing problem.
The difficulties handling the Input and Output (I/O) could be reduced by compressing the data.
Unfortunately, there is little or no support by the file system except some local concepts of Zettabyte File System (ZFS) and btrfs.
This report covers on the one hand the theoretical background on data reduction techniques with a focus on compression algorithms. For the basic understanding compressibility, the Huffman-Encoding as well as the Borrows-Wheeler-Transformation are discussed.
To illustrate the difference between various compression algorithms the Lempel-Ziv (LZ)-family and the deflate format act as a starting point. A selection of further algorithms from simple ones like Run Length Encoding (RLE) to bzip2 and Zstandard (ZSTD) is presented afterwards. On the other hand, the report also documents the work to the practical side of compression algorithms and measurements on large amounts of scientific data.
A tool called fsbench providing the basic compressing infrastructure while offering a number of around fifty different algorithms is analysed and adapted to the needs of HPC. Additional functionality is supplied by a Python script named benchy to manage the results in SQLite3 data base.
Adaptations to the code and reasonable evaluations metrics as well as measurements on the Mistral test cluster are discussed.

Contents

1 Foreword

This report is divided into two large parts.
The first one covers the theoretical background of data reduction techniques, which I discussed in my talk in the seminar “New trends in high performance computing”.
It begins with different approaches in general to data reduction as scientists have a slightly different view as computer scientists on how to handle the problems.
Those approaches are then discussed in more detail, starting with mathematical operations and transforms such as the Fourier transform.
Afterwards, essential knowledge for understanding compression algorithms is presented as well as a brief overview over other techniques such as deduplication and recomputing.
It is followed by a list of compression algorithms, I found interesting for quite a few reasons.
This will be the biggest part as my masters project was to extend a tool to evaluate compression algorithms.
Therefore, a detailed insight is given for a number of selected compression algorithms for example the LZ-family, especially LZ4, and bzip2.
At the end, there is a conclusion summarising the most important aspects.

The second part documents my work in within the project “Evaluation of parallel computers”. The main goal was to improve and extend the general infrastructure to compare different compression algorithms on large amounts of scientific data stored in several different data formats.
Due to some basic erroneous behaviour the focus shifted over time to fixing the infrastructure and providing general solutions on how to approach compression algorithms’ evaluation. I analysed the already existing tool “fsbench” and adapted the input from a file based approach to a stream like behaviour.
Those changes and the ones to the corresponding Python script “benchy” are documented and explained later on as well as all the problems I encountered on my way.
At last the measurements I run on the Mistral test cluster are discussed.

2 Introduction - Seminar

This chapters elaborates some of the reasons why scientists, especially computer scientists are concerned with data reduction techniques. As an example, details on the planned square kilometre array are given to illustrate the extreme requirements on storage systems.

Due to the ever-present gap between computational speed and storage speed every computer scientist is confronted with a set of problems regularly.
The storage devices are not capable of keeping up with the speed the data is computed with and needs to be processed with to fully exhaust the possibilities brought up by the computational hardware.
In high performance computing (HPC) storage systems are gigantic and store around two-digit numbers of PB while they need to cope with throughput around TB/s.

So for this field of research it is even more important to find solutions to this growing problem.
The difficulties handling the I/O could be reduced by compressing the data.
Unfortunately, there is little or no support by the file system except some local concepts of ZFS and btrfs.

Such an amount of data comes with further problems.
Often ignored is the maintenance aspect. The storage usually needs constant power supply to store the data. Besides the electricity bill there are the costs to buy the storage. So reducing the amount of information stored will reduce the bill or enable storing additional data as well.

However, handling growing sets of data is not only a problem with regards to capacity or budgets' limits. Another important aspect is transmitting data consumes time and energy.
Exceeding the limits of a system can have a lot of severe consequences.
An example to illustrate further problems is the Kepler satellite.
First of all, it is an isolated system not capable of adapting on the fly with just buying new hardware. Therefore, it is crucial to find a fitting solution over a long period of time.

The scientific data is sent down to earth every 30 days.
Since 95-megapixel images are recored by the Kepler satellite once every six seconds the resulting data set size exceeds the storing and transfer capabilities of the system by far. [Bor09, p.24-25]. The down link to earth is provided via a K_a-band with a maximum transfer rate of around 550 KB per second [Bor09, p.7].
So, data selection as well as data reduction is needed to be able to send the data most

important to the mission down to earth.

And reducing data becomes even more important as the experiments conducted for example in physics are producing more and more data.
Without this massive amount of data scientists won't be able to gain any new insights to their research field. The 'easy questions' are already answered.

For example the square kilometre array (Square Kilometre Array (SKA)), a gigantic radio telescope, is needed to further investigate galaxy formation and their evolution as well as to test Einstein's general theory of relativity.
It will be able to survey the sky more than ten thousand times faster than before while being 50 times more sensitive. [TGB+13]. The resulting data rates will be about many petabits/s of data which represents over 100 times the internet traffic data rates of today. This is due to the enormous measurements of the collecting area which sums up to over a square kilometre and inspired the name.

All those diverse problems will very likely not just disappear but continue to grow rapidly.
Therefore, it is inevitable to find solutions on how to handle the exponential growth of produced data and the enhanced requirements on storage systems.

3 Different data reduction techniques

This chapter covers different ways of data reduction, particularly mathematical operations like the Fourier series as well as deduplication and recomputation.

As already mentioned, scientists are closely involved in this topic as well as computer scientists who normally solve any occurring problems around data storage.
However, they have slightly different methods how to reduce the data.

Physicists for example will rather discuss what data is most import and which precision is really needed than how to compress it on already limited hardware.
They are more interested in filtering the data to keep the valuable one.
Computer scientists on the other hand are often just concerned with how to compress the data as small and as fast as possible.

This results in a variety of approaches to minimise the data size.
In the following, a selection of attempts is presented and briefly discussed.
In general, there are two basic methods called lossless and lossy reduction.
Lossy reduction is the term for all those variants which, in one way or another, lose data and therefore manage to keep the size smaller by just discarding some part of the data. For example cutting down the decimals of a number decreases the size while losing precision.
In contrast, lossless reduction describes all those variants which are able to maintain all the information by reducing the redundancy, e.g. deduplication or the algorithms discussed in chapter 5.

3.1 Mathematical Operations

The field of mathematical operations with the aim to reduce data size is very large.
Therefore, this section is only able to merely touch the surface and point to further and additional literature.
There is a large variety of operations ranging from simple ones like rounding to a certain precision to more complex approaches like Fourier series or the Fourier transform.

In case of the Fourier series, one can describe the spectrum of a periodic signal using sine and cosine coefficients and if only a certain precision is needed discard the higher terms.
For describing aperiodic signals the Fourier transform is used.
Fourier series: $f(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty}(a_k \cdot cos(kt) + b_k \cdot sin(kt))$
Fourier transform : $f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \hat{f}(\omega) e^{i\omega t} \, \mathrm{d}\omega$

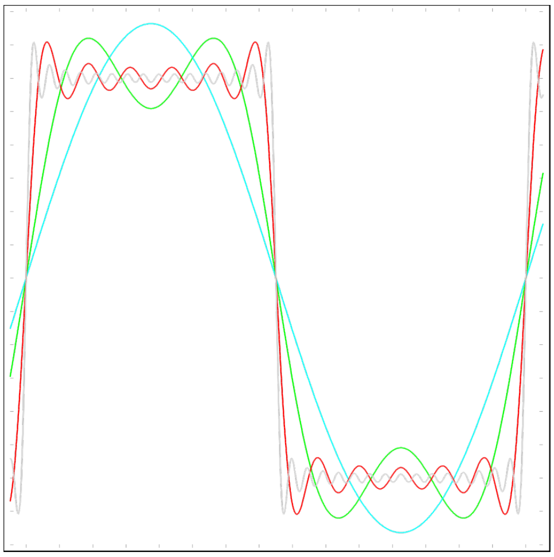
Figure 3.1: Rectangular pulse and approximating Fourier series [fou]

In Figure 3.1 the approximating Fourier series for a rectangular pulse is illustrated. The first order is a normal sine function. With growing order the number of coefficients (visible as increasing number of minima and maxima) for the resulting sum comes closer to the original pulse.

- **Cyan**: first approximation
- **Green**: second approximation
- **Red**: sixth approximation
- **Grey**: fifteenth approximation

If a sufficient precision is reached one can describe the signal with this approximation and cut down the size of storage drastically.
Another way of using this idea of splitting a signal into its frequencies is by applying the fast Fourier transform, which is an efficient algorithm to calculate the discrete Fourier transform.
This method is often used when processing images or audio signals.

Typically, images result in only a few frequencies with a high amplitude. Hence, one can dispose of the rest.
Another example is the MP3-Format which uses a modified version of the discrete cosine function to reduce the stored frequencies to the ones human can perceive (as described in ISO/IEC 11172-3:1993).
Other lossy approaches are statistical methods like linear regression and smoothing.

Over the last decades not only the number of observation increased but also the number of variables corresponding to an observation. This leads to high-dimensional data sets in which not all of the variables are equally important to understand the observed system. Thus, an obvious solution is to transform the problem to a lower dimension while still enabling sensible analysis. The difficulty lies clearly in determining the boundary what a still sufficient data set looks like. This problem and following ones are discussed by Fodor in "A survey of dimension reduction techniques" [Fod02] as well as in [Rus69].

A combination of different approaches is discussed to face the gigantic challenges arising with the SKA in "Integrating HPC into Radio-Astronomical Data Reduction" [WGHV11] and more detailed in "the murchison widefield array: square kilometre array precursor "[TGB+13].
Besides image processing with the fast Fourier Transform, also reducing the dimensions plays an important role to cut down the size of the data. As integrating the visibility over longer time intervals is not a useful change, snapshot images and therefore shifting the problem to the image domain are described as well.
Another way of handling large data sets is presented by Namey et al. in "Data Reduction Techniques for Large Qualitative Data Sets "[NGTJ08]. They combine filtering the data, clustering techniques as well as using similarity matrices to compare structured sets.

3.2 Deduplication

Another attempt to reduce the size of a file without losing any information is called deduplication.
The data of a file is split into non-overlapping blocks. Every block is stored only once for the whole file. For every following occurrence a reference to the specific block is set in an additional table. There exist different techniques for splitting the data into blocks.

- **Static chunking**: The data is divided into equally sized blocks, usually according to the underlying file system's block size or a multiple of it.
- **Content defined chunking**: To reduce the redundancy which is introduced by static chunking, as even slightly different data sets result in several data blocks, content defined chunking provides a different approach. To be able to discover identical areas not equal to a block size, a hash value is computed for all sliding windows over the data ([Mei13]).

To recognise duplicates the checksum of each block is computed, often based on Secure Hash Algorithm 256 (SHA256). For deduplication not to slow down the system, the reference table as well as the checksums needs to be located in the main memory to allow quick access for every look up. The achievable deduplication rate is determined by the block size. A larger block size decreases the rate whereas a smaller rate increases the memory requirements. For every TB of data using a block size of 8 KiB between 5-20 GB of additional memory are necessary to store the table ([Kuh16b],[Kuh16a]). To justify this effort a proper data reduction rate is essential. However, a large scale study on deduplication in online file systems implies a reduction rate of 20 % to 30 % [MKB+12].

A detailed description of different chunking procedures, duplicate detection approaches, deduplication techniques and their performance is offered by D. Meister in his dissertation [Mei13]. Mandagere et al. provide an evaluation of several different techniques as well as general advise for system administrators [MZSU08]. Further data reduction possibilities in data deduplication systems like file recipe compression are discussed in detail by Meister et al. [MBS13].

3.3 Recomputation

This section is completely based on [Kuh16a].
A different method to reduce the storage requirements of long time storage is recomputation. With this approach the results are recomputed again when requested. This allows to drastically decrease the used storage. However, it introduces several difficulties.
As bit by bit correctness is often requested, even slightly different results are unacceptable. To be able to recompute results maybe even years later on a different machine, it is crucial to find an appropriate set of information to store.

- **Storing binary files**: This approach is simplified by using containers and virtual machines. Recomputation on the same machine is possible by static linking and storing the used modules. However, the execution on a different architecture is difficult. Differences in the underlying instruction sets like x86-64 or the Performance Optimization With Enhanced Risc (POWER) architecture (Performance Optimization With Enhanced Risc) or the endianness complicate the recomputation.

- **Storing source code**: Variation in the used hardware can be dealt with more easily. Additional overhead is introduced to run the code with different compilers on different operating systems. New technologies like processors or network components might induce changes.

Whether this methods is sensible depends mostly on the access requests and the costs and capabilities of the system. As the development of computation units evolves differently than the development of storing medias archiving becomes more expensive in relation to recomputing and therefore is more likely to be used in the future.

4 Information theory and probability codes

If not noted otherwise this chapter is based on "Introduction to Data Compression" by G. Blelloch [Ble01].
This chapter covers the basic knowledge needed to understand compression algorithms. First, an insight is given into information theory and terms like entropy or prefix codes. Afterwards, some essential transforms like the move-to-front transform or the Burrows-Wheeler transform are explained. Lastly, the correlation between entropy of a data set and its compression ration is discussed as well as advanced method to estimate the compressibility of data.

4.1 Entropy

Originally, the term entropy was introduced in statistical physics. In the following, it is used as a measurement for the information density of a message. Shannon defined entropy as,

$$H = \sum_{s \in S} p(s) log_2 \frac{1}{p(s)},$$

where p(s) is the probability of a message s.
Considering an individual message $s \in S$, the self information of a message is defined as,

$$i(s) = log_2 \frac{1}{p(s)}.$$

The self-information is the logarithmic measure how much information a symbol or a message carries.
In other words, the entropy of a symbol is defined by the expected value of the self-information. This means, an often occurring message contains less information than a rare message. For example, a message saying it is going to rain in Hamburg is less informative than one stating it is going to get hot and sunny.
To take dependencies between messages into account, conditional entropy and Markov chains are used. They are discussed in detail in [Ble01, p.7-9].

4.2 Kolmogorov complexity

There are several approaches to describe the complexity of a message. One of them is the Kolmogorov complexity named after the Russian mathematician Andrey Nikolaevich

Kolmogorov. It is the shortest description of statements that will produce the character sequence looked at. For example:

message: ABABABABABABABABABAB

shortest description: AB*10

$\Rightarrow$ Kolmogorov complexity : 5

The message is a string consisting of ten times “AB” which is the shortest description to produce “ABABABABABABABABABAB”. So, the Kolmogorov complexity is a measure for the structuredness of piece of data and also of its compressibility.
The definition of the Kolmogorov complexity using prefix machines as well as its behaviour regarding several distribution functions and why the expected Kolmogorov complexity equals the Shannon Entropy is explained in detail by Grunwald in [GV04].

4.3 Probability codes

In the following, the term “message” is used to describe a part (character or word) of a larger message which will be referred to as “message sequence”.
Each message can be of a different probability distribution than the others. Blelloch uses the example of sending an image to illustrate this [Ble01, p.9].
The first message might contain information about the colour and the next message specifies a frequency component of the previous colour.
In coding theory algorithms are distinguished in two classes: those which use a unique code for each message and those which combine the codes for different messages.
Huffman codes belong to the first class whereas arithmetic codes are in the second class. They are able to achieve better compression but due to the combination of codes the sending of messages has to be delayed. In this report only the first class is covered.
A detailed insight into arithmetic codes is given by Said in “Introduction to arithmetic coding-theory and practice ”[Sai04].

4.3.1 Prefix codes

When using compression, often there are code words of a variable length.
To avoid ambiguity where one code words stops and another starts, there exist several approaches. Using a special stop symbol or sending the length of a code word, however require additional data to be send.
A more efficient solution are prefix codes. Those are uniquely decodable codes.
A prefix code is a code where no code word is prefix of any other code word. For example:

$$a = 1,\ b = 01,\ c = 000,\ d = 001$$

$$010001001101 = bcadab$$

When decoding such a message there is only one fitting match.
Once this is found, there does not exist any longer code that will also match due to the prefix property. Another important feature of prefix codes is, that one can decode a message without knowing the next message. A prefix code is optimal, if “there is no other prefix code for the given probability distribution that has a lower average length”[Ble01, p.10]. It is important to note the following property:

$$\text{If C is an optimal prefix code for the probabilities } \{p_1, p_2, ..., p_n\ \}$$

$$\text{then } p_i > p_j \Rightarrow l(c_i) \geq l(c_j).[\text{Ble01}, p.12]$$

If the probability for i is greater than for j this implies that the code for j must be at least as long as the one for i. Or in other words the encoding of common symbols takes less bytes than that of less probable ones.

4.3.2 Huffman codes

The Huffman code, named after David Huffman, is an optimal prefix code. In Listing 4.1 the pseudo code for creating the prefix tree for a given set of messages and their probabilities is noted.

Listing 4.1: Huffman prefix-tree algorithm

```
1. BUILD a tree for every message with a single vertex with
   ↪ w_i = p_i
2. REPEAT until only one tree left
    1.SELECT t_1,t_2 with minimal root weights (w_1,w_2)
    2. COMBINE them into one tree:
        1. ADD new root with w_r = w_1 + w_2.
        2. MAKE t_1,t_2 children of v_r.
```

For the four equally probable message a, b, c, d the resulting Huffman prefix tree is depicted in Figure 4.1. It is the shortest code possible code that can be created by a character based coding.

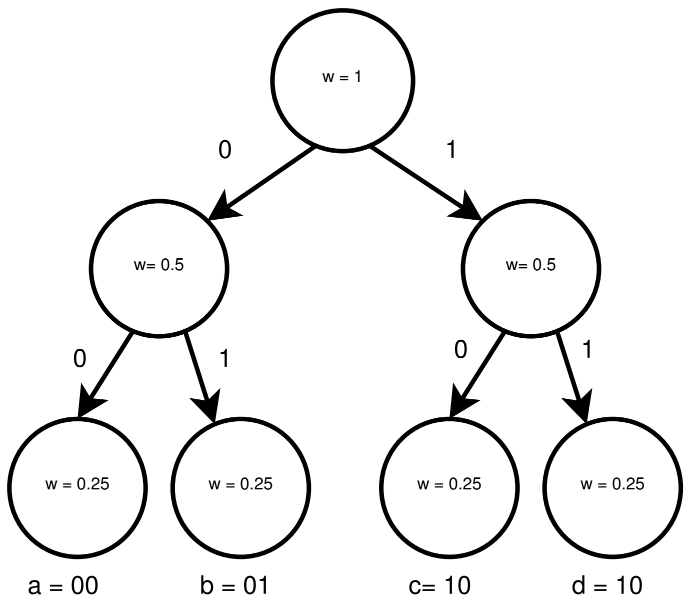

Figure 4.1: Huffman prefix tree

In Table 4.1 two possible Huffman codes are shown. This situation originates in two equal probabilities for the occurring symbols. The resulting Huffman codes are both optimal, sharing an average of 2.2 bits per symbol. The lengths however differ heavily. To reduce the variance in the code length the pseudo code for creating the Huffman prefix tree can be altered. In case of the same probabilities for different vertices the vertex is chosen for merging that was produced the earliest in the algorithm.

Symbol	Probability	Code 1	Code 2
a	0.2	01	10
b	0.4	1	00
c	0.2	000	11
d	0.1	0010	010
e	0.1	0011	011

Table 4.1: Two possible Huffman codes
Taken from [Ble01, p.14]

4.3.3 Run-length encoding

One of the simplest algorithms to take the context into account is the run-length encoding (RLE). The basic idea is to compress each sequence of equal characters to the character itself and its occurrence. For example the string "aaabbfdddddeecccc" would be compressed to "a3b2f1d5e2c4".
As these numbers change the original character occurrence significantly an approach to preserve the compressibility of a character sequence the symbol counts are written separately. The previous example string would become "abfdec" and "321524".

Another version is the zero length encoding (Zero Length Encoding (ZLE)) which only eliminates the zero sequences in a binary string. Normally, this leads only to low compression rates.

4.3.4 Move-to-front transform

The following part is mostly based on the report by Burrows and Wheeler ([BW94, p.5-8]) as well as on [mov15].
Another simple approach that considers the context of symbol for encoding is the move-to-front transform (Move-To-Front transform (MTF)). It only offers a permutation of the input not a compression! The input are a finite alphabet and a character sequence of this alphabet. The output of MTF is a sequence of nonnegative integers where every integer is smaller than the length of the alphabet.

Listing 4.2: MTF algorithm

```
1. WRITE the complete alphabet into a string a
2. FOR ALL symbol s of the input:
    1. RETURN position of s in a
    2. REMOVE s from a and append in front
```

After the execution of the MTF algorithm, as noted in Listing 4.2, often occurring characters of the input are located further to the front of the alphabet.
Therefore, the output is more likely to contain small numbers which is useful to compress the sequence of integers afterwards [mov15].
In Table 4.2 the steps of encoding the input string "Mississippi" with the MTF algorithm are explained. The used alphabet is shrunk to "ABCIMPSabcimps" to shorten the example.
Input is the current input character of the string to be encoded. *Output* is the position of this character in the current *alphabet* (starting with 0) and *alphabet'* is the one resulting from pushing the character to the front of the alphabet.
The resulting code is (4,10,13,0,1,1,0,1,13,0,1) in contrast to (4,10,13,13,10,13,13,10,12,12,10) when the alphabet is not sorted.
This highlights the basic assumption of equal characters to appear close to each other and thus leading to low values. Due to the smaller numbers a following compression is more efficient. The MTF is often used after a Borrows-Wheeler transformation (section 4.4) which already raises the probability for equal characters to appear close to one another.

Input	Alphabet	Output	Alphabet’
M	ABCIMPSabcimps	4	MABCIPSabcimps
i	MABCIPSabcimps	10	iMABCIPSabcmps
s	iMABCIPSabcmps	13	siMABCIPSabcmp
s	siMABCIPSabcmp	0	siMABCIPSabcmp
i	siMABCIPSabcmp	1	isMABCIPSabcmp
s	isMABCIPSabcmp	1	siMABCIPSabcmp
s	siMABCIPSabcmp	0	siMABCIPSabcmp
i	siMABCIPSabcmp	1	isMABCIPSabcmp
p	isMABCIPSabcmp	13	pisMABCIPSabcm
p	pisMABCIPSabcm	0	pisMABCIPSabcm
i	pisMABCIPSabcm	1	ipsMABCIPSabcm

Table 4.2: Move-to-front encoding
Taken from [mov15]

In Table 4.3 the reverse procedure (decoding) is illustrated. Now the integer sequence (4,10,13,0,1,1,0,1,13,0,1) is to be decoded while using the alphabet “ABCIMPSabcimps”. *Position* is the current integer of the sequence and *output* the decoded symbol.
The columns *alphabet* and *alphabet’* are the same as above.

Position	Alphabet	Output	Alphabet’
4	ABCIMPSabcimps	M	MABCIPSabcimps
10	MABCIPSabcimps	i	iMABCIPSabcmps
13	iMABCIPSabcmps	s	siMABCIPSabcmp
0	siMABCIPSabcmp	s	siMABCIPSabcmp
1	siMABCIPSabcmp	i	isMABCIPSabcmp
1	isMABCIPSabcmp	s	siMABCIPSabcmp
0	siMABCIPSabcmp	s	siMABCIPSabcmp
1	siMABCIPSabcmp	i	isMABCIPSabcmp
13	isMABCIPSabcmp	p	pisMABCIPSabcm
0	pisMABCIPSabcm	p	pisMABCIPSabcm
1	pisMABCIPSabcm	i	ipsMABCIPSabcm

Table 4.3: Move-to-front decoding
Taken from [mov15]

4.4 Burrows-Wheeler transformation

This section is based on the report by Burrows and Wheeler [BW94] as well as [Gil04].
In contrast to the previous algorithms the Burrows-Wheeler transformation (Burrows-Wheeler Transform (BWT)) is not a compression algorithm!
It provides a permutation of the input data where the same characters are more likely to

follow each other. So the output of the BWT has the same length as its input. A brief pseudo code for the general functionality is provided in Listing 4.3.

Listing 4.3: Burrows-Wheeler transformation

```
1. Build all rotations of given input string
2. Sort this with a stable sorting algorithm
3. Result = last column of sorting table
```

To further illustrate this, an example is given in Table 4.4.
The input string “ANANAS” is to be transformed by BWT. $ denotes the finishing symbol. Without such a symbol the use of an index is necessary to mark the end. The left column depicts all the rotations of the string. Afterwards, the rotations are sorted with an algorithm that keeps the internal order of the letters to provide a stable sorting. In this case a lexicographic sorting was used. The right column provides the result which consists of the last letter of each sorted row.

Step 1: Rotations	Step 2: Sorting	Step 3: Result
ANANAS$	**$**ANANAS	S
$ANANAS	**A**NANAS$	$
S$ANANA	**A**NAS$AN	N
AS$ANAN	**A**S$ANAN	N
NAS$ANA	**N**ANAS$A	A
ANAS$AN	**N**AS$ANA	A
NANAS$A	**S**$ANANA	A

Table 4.4: Burrows-Wheeler transformation
Taken from [bur15]

4.5 Compressibility

This section is based on “To zip or not to zip” by Harnik et al. [HKM+13].
The performance of compression algorithms relies mostly on the compressibility of the data itself and on the ability to detect incompressible data quickly.
The entropy of a randomly generated string is a lot higher than those of a typical English text as it follows certain rules how the letters are combined.
Two approaches to detection of incompressible data are discussed in the following.

4.5.1 Prefix estimation

A technique to estimate the compressibility of a file is to compress only one block, normally the first one, and use the result to decide whether to compress the whole file or not at all. The performance of this method relies on the assumption the block resembles the whole file. Compressing the prefix does not result in any overhead only if the decision

is made to compress everything. In case of an incompressible first block the effort of trying to compress it goes to waste as the rest of the file will not be compressed. Prefix estimation is a simple solution that can minimise the time spent on incompressible data considerably as it saves the time of attempting to compress the remaining blocks.
However, this procedure fails when compressing a file including a header whose structure and therefore its compressibility differs a lot from the rest of the blocks.

4.5.2 Heuristic based estimation

This approach is based on a combination of heuristics to identify the compressibility of a block. These heuristics need to be very efficient to not induce more overhead than actually compressing the block.
In order to fulfill this requirement random samples are analysed.
The entropy (see section 4.1) is an indication for the compressibility of a data block. "Byte entropy is an accurate estimation of the benefits of an optimized Huffman encoding." [HKM+13, p. 236]. The correlation between the entropy and the compression ratio is illustrated in Figure 4.2 for 8KB data blocks, where the compression ratio is defined as the compressed file size in comparison the the original file size.
For the used data set an entropy lower than 5.5 results in compression ratio less than 0.8 and can be regarded as compressible data.
However, there is also data with a higher entropy that still can be compressed well. The reason is, that the entropy does not capture repetitions.
To avoid this kind of wrong classification an advanced method is necessary to compute the compressibility.

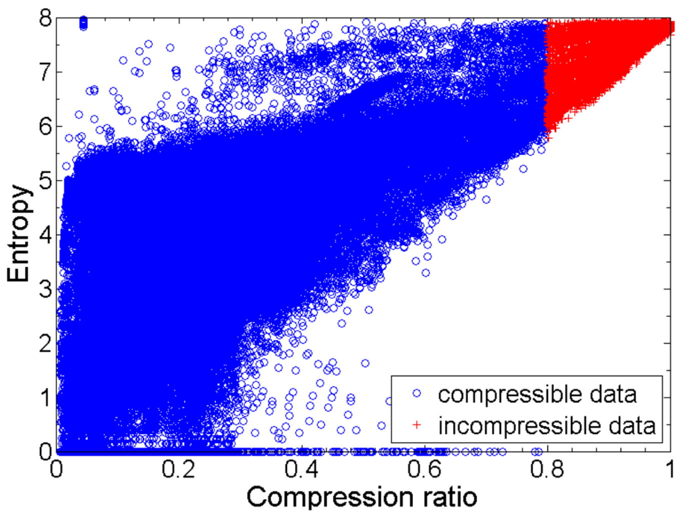

Figure 4.2: Entropy - compressibility relation [HKM+13, p. 8]
(Note: compression ratio = compressed size/uncompressed size)

In the following, a suggestion by the IBM research team in Haifa is elucidated.
A static threshold for the compression ratio is set to keep the balance between the available resources and compression savings. Above a compression ratio of 0.8 the data should not be compressed in the considered system and application. This value can be easily adopted to the performance of a different system.
The algorithm to estimate the compressibility is based on the following three heuristics:

- **Data core set size**: This is the set of unique symbols that covers a major part of the data. A smaller set is more beneficial for a Huffman encoding and probably more repetitive than a larger one. The exact value to represent the major part is to be defined with regards to the specific surrounding requirements.
- **Byte entropy**: As mentioned before the entropy is useful to cover a certain type of compressibility. A lower entropy typically results in a better compression ratio.
- **Pairs distance from random distribution**: This approach captures the probability of pairs of symbols to appear after each other in the core set. It aims to differentiate between data that contains repetitions and randomly ordered data. The calculation is defined as the Euclidean distance between "the vector of the observed probability of a pairs of symbols appearing in the data, and the vector of the expected pair probabilities based on the (single) symbol histogram" [HKM+13, p. 236]:

$$L_2 = \sum_{\forall a \neq b \in coreset} \left(\frac{freq(a) * freq(b)}{(\text{size of sample})^2} - \frac{freq(a,b)}{\text{number of pairs}} \right)^2$$

Listing 4.4: Algorithm to estimate compressibility

```
Compress if:
1. data size < 1KB
2. total number of symbols < 50
3. coreset size < 50
4. entropy < 5.5
5a. 5.5 < entropy < 6.5 AND L_2 = high (e.g. 0.02)
Do not compress if:
3. coreset size > 200
5a. 5.5 < entropy < 6.5 AND L_2 = low (e.g. 0.001)
5b. entropy > 6.5 AND L_2 = low (e.g. 0.001)
```

The algorithm to estimate the compressibility is noted in Listing 4.4 and illustrated in Figure 4.3. It aims to recommend an advise as fast as possible whether to compress or to store. Therefore, the single steps are ordered by their calculation time. The general flow,from top to bottom, passing from one heuristic to the next complex one is shown by the downward arrows. The vertical bars represent the recommendation thresholds.
The first step **(1)** is to always compress small amounts of data as computing the heuristics takes longer than actually compressing it.
(2) If there is only a small number of symbols (e.g. 50) in the data then the data should be compressed. **(3)** Data with a small core set (e.g. < 50) should be compressed. If the core set size exceeds the threshold (e.g. 200) than it should not be compressed. **(4)** Data with a small entropy (e.g. < 5.5) should be compressed.
Step **(5)** covers the decision for data with a medium entropy **(5a)** and a high entropy **(5b)**. Data with a medium and high entropy and nearly random ordering should be stored without compressing. If the heuristics can not provide a clear decision (value between thresholds) only Huffman encoding should be used to compress the data.
Finally, data with a high entropy and a high distance value should be compressed.

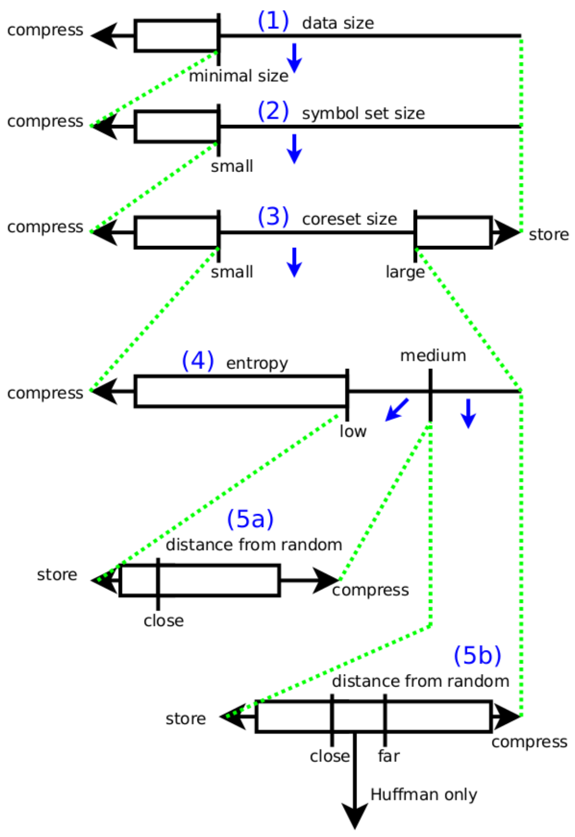

Figure 4.3: Based on IBM heuristics to detect incompressible data
[HKM+13, p. 9]

The extreme ends of the compressibility scale are covered quickly in this heuristic approach. Data with a very good compression and data that is near the incompressible end are classified fast.
But, for compression ratios between 0.4 and 0.9 all or most of the particular steps need to be calculated.
In the evaluation, this algorithm proved to be much faster than the simple prefix estimation for every compression ratio.
However, its strong side is clearly classifying incompressible data fast where the prefix estimation induced just overhead. With compressible data on the other hand prefix estimation has no overhead at all.
Thus, the general advise is to combine these two techniques and switch between them according to the properties of the current data sets.

5 Compression algorithms

In this chapter a set of popular compression algorithms is presented. First, a brief definition of the basic terms in context of evaluation of compression algorithms is given. Afterwards the algorithms based on the approach by Lempel and Ziv are discussed in detail. Then the Deflate format and the related algorithms are illustrated as well as the ideas of Gipfeli and Brotli. They are compared against the performance of bzip2 which is explained at last.

The basic idea of data compression is to reduce redundant information. To achieve this, the information is transformed to a different representation. This process is called compression, vice versa decompression. It can either be a lossless or lossy transformation. In this report, only lossless compression algorithms are discussed.
The terms compression speed, compression ratio and space savings, if not marked otherwise, are defined as follows:

- Compression speed $= \frac{\text{uncompressed size}}{\Delta\text{time}}$ measured in $[\frac{MB}{s}]$
- Compression ratio $= \frac{\text{uncompressed size}}{\text{compressed size}}$,
  e.g. 12 MB file compresses to 3 MB file has compression ratio of $12/3 = 4$
- Space savings $= 1 - \frac{\text{compressed size}}{\text{uncompressed size}}$, e.g. $1 - \frac{3}{12} = 0.75$

Normally, either a high compression speed with a acceptable compression ratio or a high compression ratio with acceptable compression speed is aimed at. Which performance is acceptable depends largely on the specific use case and the system.
The first case is for example given when optimising the I/O in a file system. In order to not induce further delays in an already critical part of the system, the focus lies on a high compression speed. Whereas, in case of long time archiving of data the space saving due to a high compression ratio is essential.
Compression can be part of three layers in the I/O stack:

- **Application layer**: The application itself enables compression. This allows for a selection of the most fitting algorithm in a specific case. Additional information about the structure of the data and the application the user typically has, can be considered.
- **File system layer**: Compression inside a file system so far is only implemented in ZFS and SquashFS. In case of ZFS only a static approach is supported which

results in the use of one algorithm for the whole file system. SquashFS is a small project to create a “compressed read-only filesystem for Linux” [Lou11]. Even though it was already included in the Linux kernel the project dispersed.

- **Device layer**: Compression in hardware is performed by an algorithm implemented in the firmware of a specialised hardware device. Therefore, it does not introduce additional overhead for the processor.

Compression on the file system level as well as on the device layer happens transparent to the application. The system itself handles the decision and does not involve the user. Following this idea, Basir et al. presented a “transparent compression scheme for Linux file systems” using Extended File System (EXT)2/3 as a basis [BY12].

5.1 LZ-Family

This section is based on the paper “A Universal Algorithm for Sequential Data Compression” by Ziv and Lempel [ZL77] if not marked otherwise.
In the following, the basic idea of the LZ77-algorithm by Lempel and Ziv is explained. It was the first approach to consider repetitions of symbol sequences and not just the probability of a symbol (see section 4.3). A repeating part just refers to the first occurrence of this specific character sequence using the already processed prefix as a dictionary. In order to minimise the time spent on searching the size of the dictionary is limited in practice. This is solved by using a buffer and a sliding window.
They are explained in the following:

- **Buffer**: This is the dictionary used to identify repetitions. The length of the buffer limits the recognisable character sequence.
- **Sliding window**: It allows to see a specific amount of characters following the current one. This enables taking repetitions into account.
- **x**: This is the buffer index starting a matching character sequence.
- **y**: It denotes the length of the reoccurring sequence.
- **z**: Contains the next character in the sliding window.

In Table 5.1 the procedure is illustrated in detail.
The input sequence is “Banane”. As the sliding window has a size of four the first four characters of the input are captured in the first step. Since there is no entry in the buffer no match can be found. Therefore, the output contains only the next character in the sliding window.
In step two and three, the dictionary still holds no matching entries which leads the algorithm to just shift the sliding window over the input.

This changes in step four. The sequence “an” of the input is also contained in the buffer resulting in the output “(5,2,E)”. The matching character sequence in the dictionary starts at the index five and has the length of two. So, the next unprocessed character in the sliding window is “e”.
Finally, the whole input is read and processed resulting in an empty sliding window.
The use of the triples enables the possibility to decode a code without an explicit dictionary as it can be built dynamically with the help of the triples [AHCI+12].
An in-depth analysis of the efficiency boundaries of the LZ77 algorithm in contrast to a code with full a priori knowledge is given by Lempel and Ziv in [ZL77].

Step	**Buffer**						**Sliding window**				**Output**
	1	2	3	4	5	6	1	2	3	4	(x,y,z)
1							B	A	N	A	(0,0,B)
2						B	A	N	A	N	(0,0,A)
3					B	A	N	A	N	E	(0,0,N)
4				B	A	N	A	N	E		(5,2,E)
5	B	A	N	A	N	E					(-,-,-)

Table 5.1: LZ77

The original LZ77-algorithm and its factorisation are the basis for several other compression algorithms also referred to as LZ-family. A detailed description of several factorisation methods is provided by Al-Hafeedh et al. [AHCI+12]. They also illustrate the main data structures, e.g. suffix tree, longest common prefix array, used by the different algorithms.
For a short overview the most prominent derivatives are noted, ordered by year they were published:

- **Lempel-Ziv 1977 (LZ77)**: (1977) The first algorithm based on a dictionary which is stored implicitly in the second component of the triple.
- **Lempel-Ziv 1978 (LZ78)**: (1978) In contrast to the LZ77 LZ78 does not store the length of a match (y in Table 5.1) enabling further compression of the text but also raising the need to store the dictionary explicitly. The initial dictionary is not empty like in LZ77 but consists of an entry for each possible input of length one. A new entry is added to the dictionary only if it is a prefix of an already included string [Ble01, p.33]. In order to decompress a LZ78 code the initial dictionary is needed.
  The success of the LZ78 algorithm is due to its fast dictionary operations.
  The dictionary is stored as a k-ary tree with the empty string as the root. Due to the prefix property all internal nodes belong to entries in the dictionary. So, every path from the root to a node describes a match [Ble01, p.36].
  The size of the dictionary can get problematic. To eliminate this problem, the dictionary can be deleted when exceeding a certain limit (used in GIF) or when it is not effective (used in Unix Compress)[Ble01, p.34].

- **Lempel-Ziv Storer and Szymanski (LZSS)**: (1982) Storer and Szymanski demonstrate in their paper “Data Compression via Textual Substitution” ([SS82]) that the LZ77 algorithm is only “asymptotically optimal for ergodic sources”. For an ergodic system the average over time is the same as the average over the space of all system’s states. So the LZ77 algorithm is only optimal for strings tending to infinity not for finite strings.
  Therefore, Storer and Szymanski present improvements to approach the optimum even for finite strings. They also introduced the use of a flag to differentiate between a character and a reference to the dictionary.

- **Lempel-Ziv Welch (LZW)**: (1985) This is a widely used adaption of the LZ78. The entries in the dictionary are usually referred to by a 12 bit long index. Entries ranging from 0 to 255 are initialised at the start. Following entries are added at run time starting with index 256. In Listing 5.1 the pseudo code is noted where pattern is the index of the corresponding pattern in the dictionary.

Listing 5.1: LZW Algorithm (taken from [lzw16])

```
INITIALISE dictionary d:
FOR ALL characters add entry := empty + character
pattern := empty
WHILE next character available:
    character := READ next character
    IF (pattern + character) contained in d
        pattern := pattern + character
    ELSE
        ADD pattern + character to d
        output := pattern
        pattern := character
IF (pattern != empty)
    RETURN pattern
```

In Table 5.2 an example is given how the LZW algorithm works.
The input string is “LZWLZ78LZ77LZCLZMWLZAP”, 22 characters long, which compresses to “LZW<256>78<259>7<256>C<256>M<258>ZAP”.
The output is only 16 characters long as a dictionary index equals one character.
To distinguish between references and characters a flag is set.
Step four illustrates the first substitution of a pattern, here LZ, with the index of dictionary entry, here <256>.

Step	Input	Found match	Output	New entry
1	LZWLZ78LZ77LZCLZMWLZAP	L	L	LZ (turns to <256>)
2	ZWLZ78LZ77LZCLZMWLZAP	Z	Z	ZW (turns to <257>)
3	WLZ78LZ77LZCLZMWLZAP	W	W	WL (turns to <258>)
4	LZ78LZ77LZCLZMWLZAP	LZ (= <256>)	<256>	LZ7 (turns to <259>)
5	78LZ77LZCLZMWLZAP	7	7	78 (turns to <260>)
6	8LZ77LZCLZMWLZAP	8	8	8L (turns to <261>)
7	LZ77LZCLZMWLZAP	LZ7 (= <259>)	<259>	LZ77 (turns to <262>)
8	7LZCLZMWLZAP	7	7	7L (turns to <263>)
9	LZCLZMWLZAP	LZ (= <256>)	<256>	LZC (turns to <264>)
10	CLZMWLZAP	C	C	CL (turns to <265>)
11	LZMWLZAP	LZ (= <256>)	<256>	LZM (turns to <266>)
12	MWLZAP	M	M	MW (turns to <267>)
13	WLZAP	WL (= <258>)	<258>	WLZ (turns to <268>)
14	ZAP	Z	Z	ZA (turns to <269>)
15	AP	A	A	AP (turns to <270>)
16	P	P	P	-

Table 5.2: An example for LZW
taken from [lzw16]

The detailed description of the algorithm and the improvements in contrast to LZ78 are elucidated by Welch in [Wel85].

- **Lempel-Ziv Ross Williams (LZRW)**: (1991) LZRW1 is designed for speed. To compress each byte 13 machine instructions are necessary and four for decompressing. A flag marks the distinction between a literal and a reference. A hash table is used to store the pointer. An analysis of the performance is provided in "An Extremely Fast ZIV-Lempel Data Compression Algorithm" by Williams ([Wil91]).

- **Lempel-Ziv Stac (LZS)**: (1993) also called Stac compression is the American National Standards Institute (ANSI) Standard X3.241-1994. It consists of a combination of LZ77 and a fixed Huffman coding. The LZS algorithm was also used in several Internet protocols:
  - Request For Comments (RFC) 1967 – PPP LZS-DCP Compression Protocol (LZS-DCP) [SF96]
  - RFC 1974 – PPP Stac LZS Compression Protocol [FS96]
  - RFC 2395 – IP Payload Compression Using LZS [FM98]
  - RFC 3943 – Transport Layer Security (TLS) Protocol Compression Using Lempel-Ziv-Stac (LZS) [Fri04]

- **Lempel-Ziv Oberhumer (LZO)**: (1996) LZO is based on LZ77. Its provides a good compression ratio with a fast compression speed. The strength of LZO is its fast decompression [Kin05] which works in-place.

- **Lempel-Ziv Markov Algorithm (LZMA)**: (1998) LZMA allows a large dictionary size and uses a different method to build the dictionary than LZ77 [TAN15]. On the one hand it uses a Markov-chain based range encoder which is a technique of arithmetic coding (see section 4.3) and on the other hand it includes several dictionary search data structures as hash chains and binary trees. [KC15] [sev]
- **Lempel-Ziv Jeff Bonwick (LZJB)**: (2010) LZJB is based on Lempel-Ziv Ross Welch 1 (LZRW1) and provides some improvements. It was written to enable fast compression of the ZFS crash dumps [lzj16]. Its compression rate is acceptable while the compression speed is fast [Ahr14],
- **Lempel-Ziv 4 (LZ4)**: (2011) LZ4 was developed by Yann Collet and is by now the successor of LZJB in ZFS.
  It offers the two data formats frames and blocks also referred to as file format. In the later, each compressed block consists of the literal length, the literals themselves, the offset representing the index of the match and the match length starting with a minimum length of four. LZ4 offers two different approaches to compression:
  - **LZ4 fast**: This algorithm is developed in order to achieve high compression speeds with acceptable compression rates. It offers different levels where the slowest level is 1. This is the regular lZ4 with high throughput which uses a single-cell wide hash table. The size of the hash table can be adapted. Obviously, a smaller table leads to more collisions and therefore reduces the compression rate [Col11] but also takes less time to compress. So, the higher the value of the acceleration level, the higher the compression speed but the lower the compression rate.
  - **LZ4 HC (high compression)**: With this algorithm the compression rate is about 20 % higher, whereas the speed is around ten times slower [Fuc16]. Complex structures like binary search trees (Binary Search Tree (BST)) or morphing match chains (Morphing Match Chain (MMC)) function as as search function finding more matches and therefore increasing the compression ratio. MMC does not only use one normal hash chain but also a second chain linking the matches. So, the number of comparisons shrinks as the first n Bytes are equivalent, where n is the minimum matching size. Those are incrementally adapted in order to minimise the memory usage. A detailed insight in the functionality and a comparison of MMC to BST are given by Collet in [Colb].

ZSTD

ZSTD, short for Zstandard, is a compression algorithm also developed by Yann Collet. Its goals are providing a good compression ratio and a good compression speed to fulfill the “standard compression needs” [Col15]. It is adaptable to the user’s needs similar to LZ4, trading compression time for compression ratio. The decompression speed is not affected by this adjustment and remains high.

ZSTD is a combination of an LZ based algorithm and a finite state entropy encoder (Finiste State Entropy Encoder (FSE)) as an improved replacement for Huffman encoding. It also offers a training mode that creates dictionaries based on a few samples for selected data types. This improves the compression ratio for small file dramatically [Col16].

5.2 Deflate

The Deflate data format is specified in the RFC 1951 [Deu96].
Its purpose was to provide a not patented, hardware independent format that is compatible with the format produced by gzip.
The Deflate algorithm uses a combination of an LZ77 based algorithm and a Huffman coding. To ensure no patent infringements the algorithm itself is not specified in detail, so it can be implemented in a way not covered by patents.
Each compressed data block holds a pair of Huffman code trees specifying the representation of the compression.
The format bases on two types: the literals for which no match was found and the pointer to the duplication, described by the length, limited to 258 bytes, and the backward distance, limited to 32 KB.
The representation of literals and the length is encoded in one Huffman tree, the distances in a separate Huffman tree.
In the Deflate format the Huffman coding has two additional rules: codes of the same length are ordered lexicographically and longer codes follow shorter ones. This simplifies the determination of the actual codes.
There are two versions of Huffman codes used: fixed ones and dynamic ones. They only differ in the way they define the literal/length and distance alphabets. For a detailed description of fixed and dynamic Huffman codes see [Deu96].
The evaluation of Deflate's performance,done by Alakuijala et al. in [AKSV15], is shown in Table 5.3. Due to the format description worded in general terms, there exist several implementations of the Deflate format. The most popular ones are listed in the following, ordered by their publication date:

- **ZIP**: This file format was invented by Phil Katz in 1989. It allows several compression algorithms to generate the format of which Deflate is the most common. The original implementation was patented in 1990 (U.S. Patent 5051745) [Kat90].

- **GZIP**: As stated in the RFC 1051 the possibility to produce Deflate files is not patented. Jean-Loup Gailly and Mark Adler developed it as a free software with its first release in October 1992. As the first letter indicates gzip is part of the GNU Project. A detailed description of the internals especially the table lookup is offered by Gailly and Adler at [GA15]. ZLIB is the related library providing an abstraction of the Deflate algorithm.

- **Zopfli**: Zopfli is another implementation of the deflate algorithm. It was developed in 2012 by Jyrki Alakuijala who is part of the Swiss Google Team. This inspired

the naming which derives from Zöpfli, a Swiss pastry. Zopfli enables a higher compression ratio while still being compatible to the Deflate format. The compression speed is around 80 to 100 times slower than with using gzip but results in a smaller output size (3.7 -8.3%) [AV].
The denser compression is possible due to the use of a shortest-path-search through the graph of all possible Deflate representations of the uncompressed data [Bhu13].

5.3 Algorithms by Google

5.3.1 Gipfeli

Gipfeli is a compression algorithm developed by R. Lenhardt and J. Alakuijala. They explained the naming as follows: "Gipfeli is a Swiss name for croissant. We chose this name, because croissants can be compressed well and quickly" [LA12, p.1].
This idea probably inspired the naming of Zopfli and Brotli as well.
Gipfeli is based on LZ77 and focuses on achieving a high-speed compression.
Therefore, Huffman codes or arithmetic codes are not used.
As opposed to most algorithms aiming for high compression speed, Gipfeli uses entropy coding, an ad-hoc entropy coding for literals and a static entropy code for the backwards references. In order to keep the resource usage in range the input is sampled to gather the statistics and build a conversion table. Processing the whole input to compute the entropy code for the literals is too time consuming.
Gipfeli includes a number of improvements for higher compression speed:

- **Limited memory usage**: The implementation uses only about 64 KB of memory. Therefore, it fits into the Level-1 and Level-2 caches of a CPU. This allows for a significantly faster compression.
- **Adapted references**: Gipfeli offers support for backwards references to the previous block. The value of a hash table entry describes the distance of the hashed symbols to the start of the block. All values smaller than the current represent a reference to the current block, otherwise to the previous block. Due to the semantics of building the hash table, there is no need to adapt the values when handling backward references. If no match is found for a certain time, the size of the steps is increased. This enables quick handling of incompressible input.
- **Entropy code for content**: The main savings (over 75%) using the entropy code come from compressing commands. However, it is also sensible to compress the content and reduce the number of bits written.
- **Unaligned stores**: Often, only a few bits need to be copied. So, it is much faster to use an unaligned store than to rely on *memcpy*.

5.3.2 Brotli

Brotli, named after another Swiss pastry (Brötli), was designed by Z. Szabadka and J. Alakuijala. It was submitted as a draft to the Internet Engineering Task Force (IETF) in April 2014. In July 2016 is has been approved as RFC 7932 [Ala15]. Brotli defines a new data format.
The general idea is to combine an LZ77 approach with a sophisticated Huffman encoding of a maximum length that needs to fulfill the same requirements as in case of the Deflate format (see section 5.2).
The resulting prefix codes are either simple or complex, which is indicated by the first two bits of the prefix code.
A value of one classifies the code as simple. In that case the code has only two more bits following, representing the number of symbols minus one.
The complex prefix codes base on a different alphabet size depending on their purpose:

- **Literal**: The alphabet used to compute the prefix code for a literal has a size of 256.

- **Distance**: Similar to other LZ algorithms the distance is used to reference duplicates and noted in the meta-block. The alphabet size for the prefix code is dynamic. Its computation is based on the sequences of past distances, the number of direct distance codes and the number of postfix bits:

  $16 + \texttt{NDIRECT} + (48 << \texttt{NPOSTFIX})$
  A detailed description is provided in the RFC [Ala15, p.17-19]

- **Insert-and-copy length**: The insertion length describes the number of following literals. The number of bytes to copy is defined by the copy length. The prefix code to encode this information is based on an alphabet size of 704.

- **Block count**: To compute the prefix code for the block count an alphabet size of 26 is used.

- **Block type**: NBLTYPES denotes the number of block types of literals (L), insert-and-copy-lengths (I) and distances (D).
  The alphabet size of each block type codes is `NBLTYPES(x) + 2`, where x is I,L, or D.
  The pair of block type t and block count n is called block-switch command and indicates a switch to a block of type t for n elements.

- **Context map**: The context map is a matrix containing the indexes of prefix codes. It indicates which prefix code to use for encoding the next literal or distance. There is one matrix of size 64 * NBLTYPESL for literals and one of size 4 * NBLTYPESD for distances.
  The alphabet size to compute the corresponding prefix codes is based on the number of run length codes (RLEMAX) and the maximum value of the context map plus one (NTREES) by `NTREES(x) + RLEMAX(x)`, where x is L or D.

Additionally, Brotli provides a pre-defined static dictionary that consists of around 13.000 strings, e.g. common words or substrings of English, Spanish, Chinese, Hindi, Arabic, HTML, Java script et cetera.
A procedure to transform the words expands the static dictionary. Therefore, every word in this dictionary has 121 forms, the original one and those produced by the 120 word transformations [AKSV15, p.2].
There exist different level of Brotli, from 1 indicating high compression speed to higher values for high compression ratio.
Alakuijala et al. compare the performance of Brotli level 1,9 and 11 against Deflate 1 and 9, LZMA and bzip2 in [AKSV15].
In Table 5.3 the results for three different tested corpora are noted.
In the uppermost block the Canterbury corpus was evaluated. The middle one shows a crawled web content corpus of 1285 files with a total size of 70 MB. 93 languages were used to reduce the bias induced by Brotli's static dictionary. In the bottom block the results for the enwik8 file are shown.
The highest value for each corpus and category is printed bold.
For every scenario Brotli compresses and decompresses faster than Deflate or bzip2. Brotli 11 even achieves the highest compression ratios.
Such evaluation motivate the decision to replace Deflate with Brotli. As of today, several web browser like Mozilla Firefox, Google Chrome or Opera support compression using Brotli [use16].

Algorithm	**Compression ratio**	**Compression speed [MB/s]**	**Decompression speed [MB/s]**
Brotli 1	3.381	**98.3**	334.0
Brotli 9	3.965	17.0	**354.5**
Brotli 11	**4.347**	0.5	289.5
Deflate 1	2.913	93.5	323.0
Deflate 9	3.371	15.5	347.3
bzip2 1	3.757	11.8	40.4
bzip2 9	3.869	12.0	40.2
Brotli 1	5.217	145.2	508.4
Brotli 9	6.253	30.1	**508.7**
Brotli 11	**6.938**	0.6	441.8
Deflate 1	4.666	**146.9**	434.8
Deflate 9	5.528	32.9	484.1
bzip2 1	5.710	11.0	52.3
bzip2 9	5.867	11.1	52.3
Brotli 1	2.711	**78.3**	228.6
Brotli 9	3.308	5.6	**279.4**
Brotli 11	**3.607**	0.4	257.4
Deflate 1	2.364	70.8	211.7
Deflate 9	2.742	18.1	217.4
bzip2 1	3.007	12.3	30.8
bzip2 9	3.447	12.4	30.3

Table 5.3: Performance of Brotli, Deflate and bzip2 on three different data sets
taken from [AKSV15]

5.4 bzip2 and pbzip2

bzip2 is a compression algorithm implemented by J. Seward based on the BWT transform and a Huffman coding, aiming for a high compression ratio.
Effros et al. present a theoretical evaluation for the BWT in “Universal Lossless Source Coding With the Burrows Wheeler Transform” [EVKV02].
They explain in detail the asymptotic analysis of the statistical properties of the BWT as well as the proofs of universality and the boundaries on convergence rates.
The optimal coding performance of BWT in contrast to LZ77 and LZ78 is established with the following results:
For a finite-memory source generating sequences of length n, the performance of BWT-based codes converges to $\mathcal{O}(\frac{\log n}{n})$ exceeding the performance of LZ77, converging to $\mathcal{O}(\frac{\log \log n}{\log n})$ and LZ78, converging to $\mathcal{O}(\frac{1}{\log n})$ [EVKV02, p.1062].
All BWT based codes include variations of the MTF coding [EVKV02, p.1068].
The individual steps for bzip2 are listed in the following:

- **RLE**: The first step of the bzip2 algorithm is to compress the initial data using a run-length encoding (see subsection 4.3.3), compressing sequences of equal characters to one appearance and a count.
- **BWT**: The Burrows-Wheeler transform (see section 4.4) is not a compression algorithm. It provides a permutation of the input data with a higher probability for same characters to appear following each other.
- **MTF**: The subsequent move-to front transform in most cases reduces the entropy of the data. It transforms the input data to a description, where the length of a symbol is based on the distance to its last appearance. Equal characters close to each other are transformed into low value integers. So, recently used symbols get a shorter description than symbols appearing less.
- **RLE of MTF result**: The second RLE is a lot more efficient due to the reordering of the characters by BWT and MTF such that the same characters are more likely to appear following another.
- **Huffman coding**: The performance of the Huffman coding is improved by the MTF due to the entropy reduction.

bzip2 achieves compression ratios comparable to method like Prediction by Partial Matching (PPM) with a better compression speed [Gil04].
It offers different compression levels reaching from 1 to 9. These level describe the block size used to process the data, raging from 100,000 Bytes (1) to 900,000 Bytes (9) [Sew10].
In order to improve the performance on parallel machines a multi-threaded approach was developed named pbzip2. The concurrent processing of blocks resulted in nearly linear speedup as evaluated by Gilchrist [Gil04]. This enables bzip2 to be still competitive in the future.

6 Conclusion - Seminar

In times of ever-growing computation speed and an dramatically increasing amount of data to be processed or stored, data reduction techniques are essential.
A balance between computational overhead and the possible storage savings needs to be found.
There are approaches to prioritise the data deemed most important like the Fourier series approximating a periodic signal.
Another solution is to use deduplication and store a block of data just once and reference to it when occurring later on. While it saves storage capacity, it introduces a significant increase of memory consumption.
Also the idea of recomputation results in several problems when trying to maintain the necessary environment.

Further methods to reduce the size of a data set are offered by a various number of compression algorithms.
Reaching from probability codes like entropy encoding or run-length encoding to dictionary based attempts the underlying concepts vary greatly.
A prominent example is the LZ algorithm using a sliding window to identify reoccurring sequences and referencing them via a dictionary.
This idea spawned several extending algorithms using numerous additional structures and functions like a Markov-chain based range encoder or morphing match chains in order to increase the compression speed and ratio.
This lead to a variety of solutions having big differences in their performance.
From fast and moderately dense compressing algorithms like LZ4 to slow and highly compressing approaches like bzip2 a lot of different formats and standards arose.

7 Introduction - Project

As discussed in the previous part of this report there are various methods to compress data. The goal of the project was to evaluate today's compression algorithms.
The following points were of special interest:

- Performance evaluation of compression algorithms considering specific file types
- General overview over algorithms' capabilities
- Providing suggestions for data centers and researchers

The task was consisting of several subtasks.
First, the already existing tool "fsbench" needed to be adapted the needs of HPC.
Afterwards, a measurement was to be conducted to enable a detailed evaluation.
The main goal was to improve and extend the general infrastructure to compare different compression algorithms on large amounts of scientific data stored in several different data formats.

Therefore, I analysed fsbench providing the basic compressing framework while offering a number of around fifty different algorithms. Also, the input method changed from a file based approach to a stream like behaviour.
Additional functionality is supplied by a Python script named "benchy" to manage the results in SQLite3 data base.
The changes in fsbench and "benchy" are documented and explained later on as well as all the problems I encountered on my way.
Following, adaptations to the code and reasonable evaluations metrics are discussed.
Over time the focus shifted from identifying the most promising algorithms for a file type to fixing the infrastructure and providing general solutions on how to approach compression algorithms' evaluation. At last, the measurements I run on the Mistral test cluster are discussed.

8 Design, Implementation, Problems

The necessary software to fulfill the project task can be split into three parts. The first parts deals with the compression tool itself. Following is the part handling the parallelisation of the tool via a python script. The last part is the database used to store the results.

- Fsbench is the programm used for the actual compression of the input file. It provides a framework to evaluate the performance of a variety of compression algorithms and codecs. A list of these is provided in the next section. The tool itself does not provide batch processing of different files.
- Benchy is the python script enabling parallelisation over the input files. The main structure was developed by Hatzel et al. in their project in 2015.
- An Sqlite3 database is used to store results. The underlying database scheme was adapted to fit the needs of HPC.

8.1 Current state of the programm

8.1.1 fsbench

Fsbench is a free open source tool written by Przemyslaw Skibinski in C++. It enables testing a set of compression algorithms on a specific input file.
The input file is defined internally as an `ifstream*`.
However, it does not offer the functionality typically associated with a stream.
Fsbench is not capable of reading continuously into the buffer. It is only able to read files that have a size which is smaller than a third of the available Random Access Memory (RAM) size. This is due to the implementation using three buffers: an input buffer keeping the original file, one buffer for compressing and another for decompressing. As the input is read into the input buffer in one piece the size of this buffer limits the processable file size.

So, my first modification was to change the I/O behaviour.
In Listing 8.1 the main structure is shown that enables a stream like input processing. First, the size of the whole input file is determined and split into blocks. The block size as well as the size of the available RAM are passed as a commandline argument to the programm .

Depending on whether the file size or the block size is bigger they are initialised accordingly.
An additional for-loop iterates over all blocks of input data and processes them sequentially. To reduce the I/O throughput on the evaluation system a data block is read only once and then compressed by all the selected compression algorithms.
The measurements are executed right before and after the encoding and decoding step.
When the last block is reached it is essential to check whether the block is completely filled.
If the size of the remaining input data is smaller than a block size, the reading function will fill in zeros until the block size is reached.
This will falsely increase the compression speed for the last block.

Listing 8.1: FSBench

```
if(file_size <= block size)
    block_size = this->input_size;
else
    this->input_size = block_size;
            ...
number_blocks = file_size / block_size;
if (rest)
    number_blocks ++;
            ...
FOR-LOOP over all blocks
    if (last block)
        block_size = file_size-(number_blocks * block_size);
    read input file(input buffer, block_size);
    FOR-LOOP over all algorithms
        FOR-LOOP iterations
            get time;
            encode;
            get time;
            ...
            get time;
            decode;
            get time;
```

The compression algorithms available in FSBench are listed below without the bugged ones which are mentioned in section 8.3. Algorithms analysed in the evaluation are print in bold.

- LZFX **LZ4** lzg LZSS-IM LZV1 **LZO** 7z-deflate64 Nobuo-Ito-LZSS nop zopfli/zlib fastlz Halibut-deflate **lzjb** **LZ4hc** zling LZF bcl-lzfast **blosc** BriefLZ QuickLZ bcl-rle QuickLZ-zip/zlib nrv2b **ZSTD** nrv2d z3lib miniz **lrrle** **7z-deflate** crush

zlib Yappy lodepng **bzip2** Snappy Shrinker LZWC lzmat pg_lz nrv2e bcl-huffman zlib/tinf

8.1.2 Python

The python script named benchy by Hatzel et al. provides an approach to process several input files in parallel.
Due to the changes to FSBench benchy needed to be adapted as well. The new I/O behaviour led to output statistics for each compressed block.
In order to populate the database with only one entry for each file a modification was necessary. As the statistics generation influences the program structure significantly I decided solve this issue in the python script.
Now, the results for one file are gathered and processed according to their type, e.g. sizes are summed up, speeds and ratios are averaged and then saved to the database. In order to analyse the compression behaviour for different file types a file type recognition was implemented as shown in Listing 8.2.
The first solution was to use the python package "magic" leading to a short and simple result. As Mistral does not support this package I used the `file`-command of Linux to solve this task.

Listing 8.2: File type

```
def get_file_type(filename):
val = subprocess.check_output(["file", "-b", filename],
   ↪ universal_newlines=True)
val = str(val.splitlines()[0])
return val
```

The next modification was to pin the threads to one core for their whole lifespan in order to avoid switching between different CPUs.
This enables the use of the related caches which allows for a higher performance.
The implementation is based on a taskset call setting the CPU affinity of a certain process using its process id as shown in Listing 8.3. Therefore, the Linux scheduler will not run the process on any other CPU. The parameter "-c" replaces the assignment using a bitmap with a list of processors [Lov04].

Listing 8.3: Python

```
worker()
try:
    r = ‘‘taskset -c cpu.count fsbench call’’
```

8.1.3 Database scheme

The database scheme was revised to keep it as simple and small as possible.
The task to offer a solution capable of dealing with HPC requirements introduced boundaries.
To enable evaluations of large amounts of data, from around several hundred GB to PB, each analysed file should result in just one entry.
The minimum attributes needed to allow a sensible analysis are listed below. The last three are added for a more comfortable comparison of the result. They could be computed when necessary to further decrease the database size:

- **Filename**: In order to interpret the data knowing which kind of input data lead to the results storing the filename is crucial. To be enable a clear identification the path is stored as well.
- **Codec**: The compression algorithm used
- **Version**: The version of the used compression algorithm to distinguish between different improvements.
- **File size**: The input file size is essential not only to determine the performance of the algorithm but to verify its correctness. If the decompressed file size differs from the input file size erroneous behaviour took place. The results related are discarded and not stored at all.
- **File type**: To analyse the relation between compressibility, compression algorithm and different file types.
- **Block size**: This is the command line argument passed to the program. The maximum value needs to be lower than a third of the system's RAM available To determine the influence of different block sizes it is stored.
- **Compression time**: The averaged minimum time needed to compress the input file with the specific compression algorithm. The choice of returning the best compression time was made in the internal design of Fsbench.
  Due to the new I/O behaviour the smallest compression time is returned for each processed block. The averaged result is then stored to the database.
- **File size after compression**: The compressed file size.
- **Decompression time**: The minimum time needed to decompress the compressed file
- **Ratio** = file size / compressed file size
- **Compression speed** = file size / compression time
- **Decompression speed** = file size / decompression time : The best ratio of the original input size and the decompression time, not the decompressed file and the decompression time!

8.2 Selection of the evaluated algorithms

Fsbench offers a long list of algorithms as shown above.
The evaluation of the whole would have taken too long, so I ran test on my computer to find the most promising algorithms.
Table 8.1 shows the results of the benchmarking on my computer (i5-4460 CPU @ 3.20GHz × 4, 16 GB RAM, SSD) on a small subset of data.
The best results in each column are in bold print. The algorithms selected for further evaluation are italicised.
Bzip2 provides the best compression ratio. A LRRLE (large range RLE) modification of the RLE algorithm has the highest compression speed. LZ4HC offers the fastest decompression.
Additionally, the Deflate implementation 7z-deflate, LZ4, LZO, zlib and ZSTD are selected as they provide an interesting set of features.
The LZ-family as well as the Deflate format are still the most prominent approaches to compressing data. So, keeping them as reference is a good way to evaluate new solutions. Blosc is an approach to compression on a different level. It aims to decrease the transfer time from the memory to the CPU. A strategy of blocking mechanism of the bus should speed up the transmission. An added shuffling should lead to higher compression ratios [PK15]. The developers claim to even surpass the memcpy() when simply copying memory due to the internal multi-threading.

Ratio	**Compression speed [MB/s]**	**Decompression speed [MB/s]**	**Codec**	**Version**
2.079	9.946	143.143	7z-deflate	9.20
2.079	9.032	143.894	7z-deflate64	9.20
1.584	82.112	88.425	bcl-huffman	1.2.0
1.274	5.022	591.625	bcl-lzfast	1.2.0
1.018	**542.811**	1 191.825	bcl-rle	1.2.0
1.529	96.634	305.142	BriefLZ	1.0.5
2.130	*13.832*	*28.583*	*bzip2*	1.0.6
1.619	24.254	226.857	crush	0.0.1
1.371	239.405	696.914	fastlz	0.1.0
1.859	3.663	86.655	Halibut-deflate	SVN r9550
1.866	.659	91.832	lodepng	20120729
1.409	*414.286*	*2 135.998*	*LZ4*	*r127*
1.597	*38.419*	***2 164.104***	*LZ4hc*	*r127*
1.417	248.447	668.585	LZF	3.6
1.428	292.134	674.065	LZFX	r16
1.377	43.282	557.531	lzg	1.0.6
1.499	363.072	676.839	lzjb	FreeBSD r263244
1.590	21.976	341.228	lzmat	1.1
1.577	*352.944*	*768.560*	*LZO*	2.08
1.501	32.395	676.839	LZSS-IM	2008-07-31
1.086	267.869	870.221	LZV1	0.5
1.637	85.528	76.676	LZWC	0.4
1.862	8.998	317.513	miniz	1.11
1.539	0.569	1 370.599	Nobuo-Ito-LZSS	1.0
1.771	6.828	334.973	nrv2b	1.03
1.767	7.063	326.333	nrv2d	1.03
1.794	7.009	326.333	nrv2e	1.03
1.734	3.245	668.585	pg_lz	9.3.4
1.493	480.912	479.509	QuickLZ	1.5.1b6
1.559	402.131	1 522.888	Shrinker	r6v
1.341	331.596	1 142.166	Snappy	1.1.0
1.453	76.534	2316.505	Yappy	v2
1.895	5.355	139.738	z3lib	1.3
1.869	*7.022*	*301.230*	*zlib*	1.2.8
1.788	42.864	145.036	zling	20140324
2.080	0.142	306.850	zopfli/zlib	1.0.0/1.2.8
1.613	*139.857*	*451.845*	*ZSTD*	2015-01-31
1.0	10 964.794	-	memcpy	0
1.0	10 964.794	-	memmove	0
1.0	*364.682*	-	*blosc*	1.2.3
1.000	*6 852.996*	*6 852.996*	*lrrle*	0

Table 8.1: Evaluation of available algorithms on test data set

8.3 Encountered problems

In the following, a brief overview is given over the main problems encountered in this project.
First, compiling FSBench lead to several partly unsolved issues:

- The building type in the CMakeLists.txt is set using `set(CMAKE_BUILD_TYPE Release)`.
  Manually adding the flag as a command line argument via `cmake -DCMAKE_-BUILD_TYPE=Release` resulted in different results for the algorithms performance. Depending on the system it reached from being two times faster to a performance improved by the factor of ten.

- In order to evaluate Brotli C++11-support needs to be enabled. However, compiling accordingly decreases the performance of every other algorithm.

- As the default compiler version of gcc (4.4.7) on Mistral is too old to support C++11-support, a newer one needed be loaded.
  The solution was to use another flag in the command line (`cmake -DCMAKE_C_-COMPILER7sw/rhel6-x64 /gcc/gcc-5.1.0/bin/gcc`)

The next problems arose when testing the algorithms. There are several ones already marked in the CMakeLists.txt by the developer to produce errors. In the following, the algorithms of theoretical interest for the evaluation are listed that do not work:

- **Brotli**: The implementation seems to be erroneous as the results are even lower than any other algorithm tested. That does not meet the results of other measurements, e.g. discussed in subsection 5.3.2.

- **Gipfeli**: This algorithm is marked with the words “rarely works” which unfortunately do represent the behaviour.

- **Zopfli**: The implementation of the compressing part is fine. However, it is not available as a decoder and therefore not suited to be evaluated.

- **Ar**: This algorithms leads to infinite loop causing the system to hang up.

- **Density**: The result of running this algorithms are sigkills aborting the execution.

- **LZHAM**: This algorithm is already known “to cause problems on some platforms”. On the system for the first tests as well as the WR-cluster and Mistral and its test cluster errors and encoding problems appeared.

Lastly, two ideas are discussed that have been discarded.
The first one was to implement an abort after a certain time, in order to reduce the time spent on incompressible data by algorithms not capable of detecting it reliably. I thought of the following approaches which are all inadequate for different reasons:

- **Benchy**: The first idea was to just implement the abort as a timeout in the python script. Simply adding another parameter to the check_output function would be sufficient to terminate the processed algorithm. However, as the goal was to decrease the I/O this is not expedient. The reason is that the algorithms still to be processed are terminated as well. Introducing a new loop over the list of algorithms would in the worst case lead to a new call of Fsbench for every algorithm. The adapted reading of the input would be obsolete.

- **Job script**: Handling an abort in the surrounding job script would just shift the problem to the next level. The problems remains the same as with benchy.

- **Fsbench**: Sending a timeout signal inside of Fsbench is easy. Terminating the execution of the thread processing the current algorithm, however, is not as simple.

- **Fsbench**: Another possibility was to implement a timer inside the iteration loop of Fsbench. Certainly, this is the worst idea solving the issue at a completely wrong level.

The second idea was to measure the CPU workload and add it to the evaluation factors. There are several possibilities to implement such a function, e.g. parsing the output of `ps -p pid -o %C`. Then the question rose which values could be returned by such an approach. Implementing the measurement inside of benchy or fsbench would lead to a workload of 0 or 100 % as nothing else is running on the system.
However, measuring the compression time of an algorithm using 100 % of the CPU allows for an indirect analysis.

9 Measurements and evaluation metrics

This chapter presents the results of the conducted measurements and their interpretation. Afterwards, general advise is given regarding the area of application of different algorithms.

The measurements were performed on a test system inside the network of Mistral which is the supercomputer used by the Deutsches Klimarechenzentrum (DKRZ)(German climate computing center).
It provides a test environment for smaller projects. For the benchmarks one node was used. This ruled out problems with parallel access to the database from several nodes. In addition, it minimises the I/O as each block is read just once and not requested by different nodes as well.
The hardware details are listed in Table 9.1.

Architecture	x86_64
Cores	24
Thread(s) per core	1
Core(s) per socket	12
Socket(s)	2
Stepping	2
CPU MHz	1200.000
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	30720K

Table 9.1: Used hardware

The resulting performance per compression algorithms for a block size of 419 430 400 byte is shown in Table 9.2.
Besides the average values for the compression speed and the compression ratio the minimum values as well as the maximum values are listed.
This was necessary as the variance is significant.
A compression speed alternating between several byte per second to several GB per second is noticeable. The same holds for compression ratios ranging from under one, indicating increased file sizes, to extrem rations of nearly 50 000.
This behaviour is not explained easily without further analysis.

Therefore, several detailed insights into the data sets features are given.

Algorithm	**Min cs [KB/S]**	**Max cs [KB/S]**	**Avg cs [KB/S]**	**Min r**	**Max r**	**Avg r**	**Min ds [KB/S]**	**Max ds [KB/S]**	**Avg ds [KB/S]**
bzip2	0.0001	24 499.9746	988.0024	0.0057	49 976.2747	41.0922	0.0006	114 229.5162	3 568.4782
lrrle	0.0202	5 115 687.2202	465 545.7698	0.0250	3 182.5062	2.0478	0.0211	4 447 180.2250	482 577.1030
LZ4	0.0110	11 888 061.4504	132 636.9275	9.7274 e-07	232.7667	2.2561	0.0070	24 571 100.0386	162 718.7803
LZ4hc	0.0120	89 956.8805	8 055.7947	1.2159 e-06	234.0557	2.7629	0.0072	2 575 962.3226	146 359.7323
lzjb	0.0012	821 256.2785	34 605.4947	1.1848 e-07	30.3450	0.9199	0.0018	3 859 257.0710	33 880.0338
LZO	0.0062	17 130 322.7725	111 472.4110	8.6009 e-07	208.0836	1.8593	0.0009	29 530 927.7216	53 572.9378
zlib	0.0001	30 699.6455	1 852.6854	4.9133 e-06	793.1625	4.1526	0.0010	345 889.4163	18 066.0071
ZSTD	0.0190	3 970 313.2261	95 807.1384	3.2890e-05	2 350.2368	9.2758	0.0219	4 509 263.3381	208 305.7386

Table 9.2: Minimum, maximum, average for compression speed, compression ratio and decompression speed

In order to further highlight the abnormal results the average compression speed and compression ratios are illustrated in Figure 9.1.
For bzip2 to reach an average compression ratio of 41 when typical ratios are between one and ten, is not only conspicuous but also suspicious.
Additionally, the average compression speed of bzip2 is notably lower than presented in other evaluations, e.g. by Alakuijala et al. (see Table 5.3).
Also, LZ4 has proven itself to perform around 800 MB per second on the Silesia corpus [Cola]. So, at least one should examine why this deviation exists.

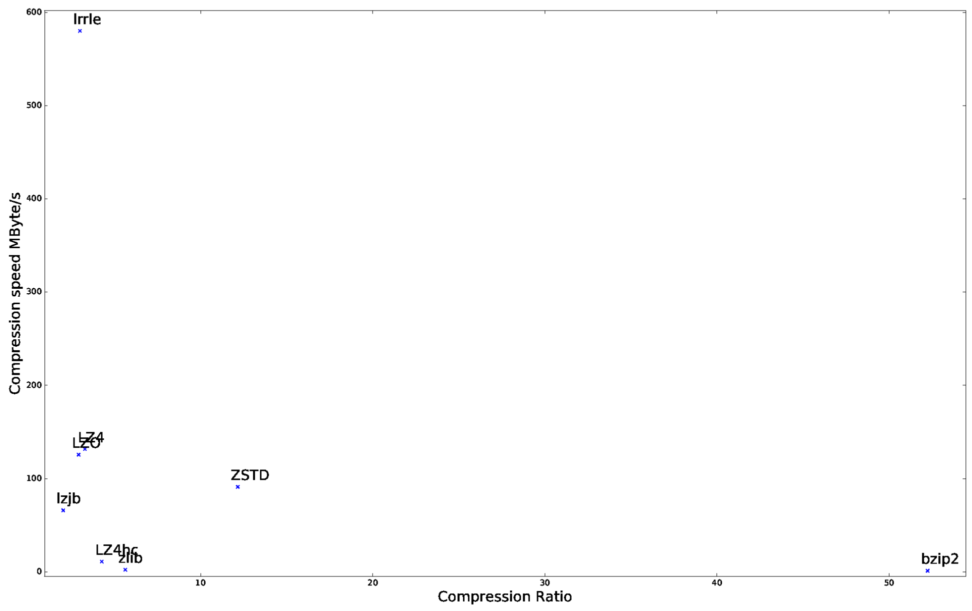

Figure 9.1: Avg. compression speed - avg. compression ratio

The first question arising was, why are there minimum compression ratios significantly smaller than one. This means the file size increased while compressing.
So, I analysed how many files of the whole data set did not shrink but grow. I was also interested in finding which algorithms lead to the most increased files.
Table 9.3 shows the number of files successfully measured for each algorithm compared to the amount with a compression ratio lower than one. The varying number of files processed is the result of problems some of the algorithms have with specific files.

Algorithm	**# Number of files**	**# files with c_ratio < 1**	**% increased**
bzip2	23 594	5 133	21.7
lrrle	23 787	11 194	47.1
LZ4	23 647	10 692	45.2
LZ4hc	23 787	9 708	40.8
LZJB	23 593	18 390	77.9
LZO	23 593	11 743	49.7
zlib	23 703	7 151	30.2
ZSTD	23 692	6 047	25.5

Table 9.3: Percentage of increase files per algorithm

The values illustrated in Table 9.3 are a lot higher than I would have guessed based on the literature.
In fact, the detection of incompressible data is said to be well developed by now. Possible explanations are discussed later on.
The higher percentage for all the advanced LZ-algorithms like LZ4,LZJB and LZO in contrast to zlib and ZSTD are surprising.
The most outstanding result are the astonishing 78 % of the files compressed by LZJB that did not become smaller.
As it is a basic function supported and developed further in the ZFS this can not be the supposed behaviour.
So, either the implementations are erroneous or the data set is very different from all those benchmarked so far.
If the later is the case then the data has to have some properties differing from the norm. Thus, I continued to analyse the data set.
The characteristics of data often correlate with the used data type. In order to find a clue the average compression ratio per file type was examined, as listed in Table 9.4. Note that the number of appearance is the number of processed files times the number of algorithms able to compress it without errors. Only four file types result in an average over 1: two are PostScript documents, one is NetCDF, the last American Standard Code for Information Interchange (ASCII) text. The entry named “data” gathers all those files that can not be classified as a specific data type. Possibly, they do not have magic number in their header indicating the file type.
Especially, interesting in the context of HPC are Network Common Data Form (NetCDF) and Hierarchical Data Format 5 (HDF).

They both belong to the group of formats called self-describing. These formats offer the possibility to add information about the data.
This allows the application to interpret the data without any further input. Also, interchanging data between different scientists becomes more easy as the necessary remarks can be made directly the data itself. They are widely used for example at the DKRZ.

File type	**# of appearance**	**Average ratio**
ASCII C program text	92	0.144
ASCII English text	254	0.157
ASCII text	2 774	**1.039**
ASCII text, with CRLF line terminators	230	0.389
ASCII text, with very long lines	115	0.320
Bourne-Again shell script text executable	217	0.096
Formula Translation (FORTRAN) program	276	0.251
Hierarchical Data Format (version 5) data	2 443	0.886
Korn shell script text executable	346	0.121
NetCDF Data Format data	164 196	**5.414**
Portable Data Format (PDF) document, version 1.2	116	0.741
PDF document, version 1.5	115	0.186
PostScript document text	18 124	**1.161**
PostScript document text conforming DSC level 3.0	7 452	0.908
PostScript -”- ,type EPS, Level 1	348	**5.632**
Sendmail frozen configuration - version E=1605	12	0.768
Unicode Transformation Format 8 bit (UTF-8) Unicode English text	116	0.509
compress’d data 16 bits	31 096	0.435
data (not recognised)	16 606	**17.655**
exported SGML document text	4 692	0.160
gzip compressed data (.nc, .n)	25 024	0.919

Table 9.4: File types and their compression ratio

In the following several reason for compression ratios < 1 are given:

- **Detection of incompressible data**: Being able to detect incompressible data does not only reduce the time needed to compress the file but also the file size. Every attempt to compress introduces overhead due to the necessary header.
- **Compressed Data**: The compression ratio for gzip compressed files is as expected. Due to the additional header overhead the file size increases slightly as represented in the 0.9.
- **Implementation of algorithms**: Another possibility to explain the enormous variance in the compression ratio and the number of increasing file sizes is an erroneous implementation.
- **Theory**: Even in theory not possible to always compress every input. If it would be possible to compress every input further, then lastly everything must be representible with nothing. However, even though theory has this boundary it does not explain this amount of increasing files.

- **File size**: If the input files are really small the header size is disproportionate and introduces massive overhead.

The only solution without verifying every implementation in detail is to consider the sizes of the files having a low compression ratio.

File type	Average file size [KB]
ASCII C program text	1.210
ASCII English text	4.807
ASCII text	**5 221.327**
ASCII text, with CRLF line terminators	152.461
ASCII text, with very long lines	28.680
Bourne-Again shell script text executable	0.497
FORTRAN program	12.872
Hierarchical Data Format (version 5) data	31 180.880
Korn shell script text executable	1.019
NetCDF Data Format data	**86 757.143**
PDF document, version 1.2	7 950.573
PDF document, version 1.5	74.575
PostScript document text	**1 851.516**
PostScript document text conforming DSC level 3.0	2 864.840
Sendmail frozen configuration - version E=1605	98 537.540
UTF-8 Unicode English text	71.043
compress'd data 16 bits	102.175
data (not recognised)	**1 164.387**
exported SGML document text	1 875 857.135
gzip compressed data	1.549

Table 9.5: Average file size per file type with compression ratio < 1

Table 9.5 shows the average file size in KB for every file type. The entries having a compression ratio over one are in print bold.
If the file size was the problem for the low compression ratio, the rest of the entries would have to have a small file size. Especially the average file size of 31 MB for an HDF 5 file and of 1 876 MB for an SGML file do not correspond to their small compression ratio. And it is unlikely to have a variance in the file size large enough to cover the actual correlation.
The most reasonable result is again that of the gzip compressed files. With an average size of 1.5 KB it is small enough to be influenced by the overhead induced by the header.

Table 9.2 not only revealed very small compression ratios but also gigantic ones of nearly 50 000.
This raises further questions. Where are high ratios to be found? Which file types lead to compression ratios over ten?

File type	**# r > 10**	**# r > 20**	**# r > 30**	**# r > 40**	**# r > 50**	**# r > 60**
ASCII text	72	35	24	14	10	10
ASCII text 2	2	-	-	-	-	-
NetCDF Data Format data	7 155	4 980	3172	2688	2675	1914
PostScript 1	56	-	-	-	-	-
PostScript 2	50	10	-	-	-	-
data	713	389	293	256	222	207

Table 9.6: Number of files reaching specific compression ratios per file type

ASCII 2 = ASCII , with very long lines
PostScript 1 = PostScript document text conforming DSC level 3.0
PostScript 2 = PostScript document text conforming DSC level 3.0, type EPS, Level 1

Table 9.6 illustrates the number of files sufficing gradation steps of compression ratios per file type. Note that again all occurrences are counted.
This table clarifies the variance. Obviously NetCDF files are the best compressible files in this evaluation. Even lrrle is able to compress three NetCDF files with a compression ratio exceeding 60.
Table 9.7 shows all algorithms except lzjb manage to reach this compression ratio.
I am not sure how lrrle manages to still compress files in a range from 300 to 1000 when LZ4,LZO and zlib can not keep up.
As discussed in the first part of this report RLE coding is one of the simplest approaches, often part of other algorithms (see subsection 4.3.3). It replaces a sequence of equal characters by a combination of symbol and the related count.

Algorithm	**# r > 10**	**# r > 20**	**# r > 30**	**# r > 40**	**# r > 50**	**# r > 60**	**# r > 100**	**# r > 300**	**# r > 1000**
bzip2	1773	1566	1440	1403	1398	1288	695	684	332
lrrle	44	32	27	27	27	17	14	14	5
LZ4	767	709	39	34	34	28	18	-	-
LZ4hc	1449	816	480	33	28	28	18	-	-
lzjb	91	37	9	-	-	-	-	-	-
LZO	782	75	39	34	28	28	15	-	-
zlib	1518	751	715	715	697	52	27	20	-
ZSTD	1624	1428	740	712	695	690	341	332	11

Table 9.7: Number of files reaching specific compression ratios per algorithm

Table 9.8 presents the results for the measurement divided in three parts for the file types NetCDF,HDF 5 and data for a compression ratio over 1.
Comparing NetCDF and HDF 5 it stands out that the minimum compression speed is significantly higher with HDF 5 files for all algorithms.
However, the maximum and average compression speeds for NetCDF are higher than the results for HDF 5.
On average NetCDF files achieve a compression ratio of 9.6282 for all compression algorithms, in contrast to 1.3365 for HDF 5 files and 39.7949 for “data” files.
As the category “data” gathers all not recognised file types it probably consists of variety of data types. This would explain why the results vary dramatically.
Only the minimum compression speeds are in the range of the other two file types.

The average compression speed of a NetCDF file is 120 435.88 KB/s, 88 690.86 KB/s for HDF 5 and 227 740.63 KB/s for data.
The three algorithms compressing the fastest while reaching an average compression ratio larger than one are LZ4, with an average of 153 108.01 KB/s, LZO, with an average of 136 790.59 KB/s and ZSTD, with an average of 127 906.57 KB/s. The compression ratio of lrrle for NetCDF and HDF 5 files excludes it from this listing.
Achieving the best compression ratios are bzip2, ZSTD and zlib. This is the expected result as these are the most complex and advanced algorithms explained in chapter 5.

File type	**Algorithm**	**Min speed [KB/S]**	**Max speed [KB/S]**	**Avg speed [KB/S]**	**Min ratio**	**Max ratio**	**Avg ratio**
NetCDF	bzip2	0.5084	9 596.8436	1 304.4447	1.0063	1 125.1810	46.3177
NetCDF	lrrle	187.1404	5115 687.2201	546 809.3168	1.0000	70.8411	1.4303
NetCDF	LZ4	13 011.9220	4 965 989.0160	125 688.3335	1.0008	101.8997	3.0378
NetCDF	LZ4hc	120.8082	89 956.8810	11 306.3463	1.0006	111.7287	4.0887
NetCDF	lzjb	21 111.1071	821 256.2785	55 502.0381	1.0060	23.7976	1.8452
NetCDF	LZO	8 157.3211	17 130 322.7726	122 790.5524	1.0004	80.9939	2.6385
NetCDF	zlib	49.0263	29 003.5476	2 259.9699	1.0052	181.1000	5.3135
NetCDF	ZSTD	803.1915	3 087 780.1562	97 826.0619	1.0091	360.8429	12.354
HDF 5	bzip2	18.38289	2 465.0011	694.9297	1.0006	7.8612	1.7111
HDF 5	lrrle	2 996.41237	1 054 795.9904	300 270.8906	1.0002	1.3948	1.0069
HDF 5	LZ4	36 719.65994	239 926.7591	101 940.3180	1.0093	1.4013	1.1590
HDF 5	LZ4hc	544.26218	9 270.5069	4 759.9255	1.0617	3.1225	1.4858
HDF 5	lzjb	29 089.59437	71 570.2631	59 100.4698	1.1361	1.1790	1.1683
HDF 5	LZO	28 762.35214	177 424.3462	95 057.1175	1.0128	1.5519	1.2742
HDF 5	zlib	80.26969	4 317.7810	1 814.5004	1.0002	4.5370	1.5428
HDF 5	ZSTD	3 125.29032	297 707.1038	145 888.6949	1.0007	4.5214	1.3439
data	bzip2	0.1032	24 499.9747	1 436.0667	1.0110	49 976.2746	226.4955
data	lrrle	1 585.3333	4 333 859.9793	1 106 985.7799	1.0000	3 182.5062	28.2917
data	LZ4	13 450.3809	3 856 405.0082	231 695.3775	1.0063	232.7666	6.7668
data	LZ4hc	130.9447	18 698.9368	8 509.7764	1.0003	234.0557	7.2023
data	lzjb	17 732.8867	712 821.0437	137 131.1071	1.0018	30.3450	3.1751
data	LZO	11 653.6552	1 887 259.8827	192 524.1121	1.0108	208.0836	6.5079
data	zlib	38.1979	30 699.6455	3 637.8858	1.0075	793.1625	14.0459
data	ZSTD	1 662.2913	3 970 313.2261	140 004.9383	1.0332	2 350.2368	25.8739

Table 9.8: Final results: compression speed with a compression ratio > 1 noted per file type and algorithm

For reasons of clarity the decompression results are shown in Table 9.9.
There are no obvious correlations between the decompression speed and the file type. Both NetCDF and HDF 5 lead to the same value range of a few to several MB/s for the average decompression speed. Notable is the maximum speed of LZ4 and LZO which exceeds 24 GB/s or even 29 GB/S for NetCDF files. However, the average speed suggests there are only a few file resulting in this performance.
The best average decompression speed is achieved by lrrle (712 233.54 KB/s), ZSTD (267 017.73 KB/s), LZ4HC (247225.84 KB/s) and LZ4 (246804.03 KB/s).

File type	**Algorithm**	**Min speed [KB/S]**	**Max speed [KB/S]**	**Avg speed [KB/S]**
NetCDF	bzip2	2.5230	46 108.7065	4 768.9714
NetCDF	lrrle	0.9878	4 447 180.2250	588 431.7359
NetCDF	LZ4	2.9634	24 571 100.0386	179 510.4468
NetCDF	LZ4hc	0.3704	2 575 962.3226	161 319.8480
NetCDF	lzjb	0.1113	3 859 257.0710	52 635.7912
NetCDF	LZO	1 249.3883	29 530 927.7216	82 316.8694
NetCDF	zlib	323.3456	197 489.9113	21 821.2852
NetCDF	ZSTD	1 126.8657	3 780 169.2320	199 329.0268
HDF 5	bzip2	93.3072	7 298.0868	2 196.5234
HDF 5	lrrle	3 092.0426	1 119 362.6837	320 294.4575
HDF 5	LZ4	40 578.7325	370 413.2421	188 906.7173
HDF 5	LZ4hc	38 969.3639	295 708.0504	163 954.5274
HDF 5	lzjb	42 648.8031	76 657.9616	58 190.7074
HDF 5	LZO	5 973.6156	70 546.0887	40 673.1534
HDF 5	zlib	487.8765	44 126.1235	15 293.9213
HDF 5	ZSTD	3 679.1392	951 834.1493	350 095.8926
data	bzip2	0.5191	114 229.5162	5 117.0319
data	lrrle	0.9771	4 435 393.0738	1 227 974.4214
data	LZ4	0.3235	2 100 796.4188	371 994.9144
data	LZ4hc	9 980.8481	2 148 036.6783	416 403.1472
data	lzjb	27 862.0433	378 264.4813	136 048.7592
data	LZO	1 367.8838	1 546 780.6175	173 586.3614
data	zlib	250.4454	345 889.4163	47 809.6156
data	ZSTD	1 861.04345	4 509 263.3381	251 628.2609

Table 9.9: Final results: decompression speed with a compression ratio > 1 noted per file type and algorithm

Depending on the scenario different algorithms meet the requirements.
The most typical use cases are listed with their main focus highlighted:

- **Compression speed,decompression speed**: An example is the file system on the computing nodes of a HPC system. In order to not slow the system's I/O down high compression speed and decompression speed are crucial. When compressing with a fitting algorithm the I/O might even increase as less data needs to be transfered. The decompression speed is important since the data is accessed regularly.

- **Compression ratio**: On archival systems the space savings introduced by a high compression ratio are the main focus.
  A high compression speed is rarely of interest as the computing costs are smaller than the maintenance costs for the archive. Depending on the access frequency a good decompression speed is useful, too.

LZ4 seems to be a good candidate to be part of the file system which is already implemented in the ZFS. Its compression and decompression speed with its compression ratio even allow for a increased I/O. ZSTD also offers an impressive compression ratio making it a sensible option for the first use case as well.
For archival purposes clearly bzip2 is still the most appropriate algorithm with nearly unrivaled compression ratios.

10 Conclusion - Project

The conducted measurement gave insight into the complexity of analysing compression algorithms.
There are several possible reasons for a certain behaviour. From erroneous implementations to special features of the analysed data sets many things need to be considered.
Domain knowledge of the specific file types is essential to evaluate which algorithm is appropriate for the given requirements.
Archiving data introduces different limitations than a file system on a computing node of a cluster.
LZ4 is an algorithm on the rising. Its high compression and decompression speeds and a compression ratio often comparable to a Deflate compression ratio make it valuable for a large range of applications.
Bzip2 is still the algorithm offering the highest compression ratio due to its complex internal structure.
Not every question raised in the beginning was answered until now.
There is still a lot to find out about the correlation of a specific file type and certain compression algorithms.

Bibliography

[AHCI+12] Anisa Al-Hafeedh, Maxime Crochemore, Lucian Ilie, Evguenia Kopylova, W.F. Smyth, German Tischler, and Munina Yusufu. A Comparison of Index-based lempel-Ziv LZ77 Factorization Algorithms. *ACM Comput. Surv.*, 45(1):5:1–5:17, December 2012.

[Ahr14] Matthew Ahrens. OpenZFS: a Community of Open Source ZFS Developers. *AsiaBSDCon 2014*, page 27, 2014.

[AKSV15] Jyrki Alakuijala, Evgenii Kliuchnikov, Zoltan Szabadka, and Lode Vandevenne. Comparison of Brotli, Deflate, Zopfli, LZMA, LZHAM and Bzip2 Compression Algorithms. Technical report, Google, Inc., September 2015.

[Ala15] Jyrki Alakuijala. Brotli Compressed Data Format. `https://datatracker.ietf.org/doc/rfc7932/`, 2015. (last accessed: 2015-11-01).

[AV] Jyrki Alakuijala and Lode Vandevenne. *Data compression using Zopfli.* Tech. rep. Google Inc., Feb. 2013.

[Bhu13] Kul Bhushan. Zopfli: Google's new data compression algorithm, March 2013.

[Ble01] Guy E Blelloch. Introduction to data compression. *Computer Science Department, Carnegie Mellon University*, 2001.

[Bor09] William J Borucki. Kepler: Nasa's first mission capable of finding earth-size planets. 2009.

[bur15] Burrows-Wheeler Transformation. `https://de.wikipedia.org/wiki/Burrows-Wheeler-Transformation`, 2015. (last accessed: 2016-08-16).

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[BY12] MA BASIR and MH YOUSAF. Transparent compression scheme for linux file system. *Nucleus*, 49(2):129–137, 2012.

[Cola] Yann Collet. LZ4 description. `http://fastcompression.blogspot.de/p/lz4.html`. (last accessed: 2016-08-02).

[Colb] Yann Collet. Morphing Match Chain. `http://fastcompression.blogspot.de/p/mmc-morphing-match-chain.html`. (last accessed: 2016-08-22).

[Col11] Yann Collet. LZ4 explained. http://fastcompression.blogspot.de/2011/05/lz4-explained.html, March 2011. (last accessed: 2015-11-02).

[Col15] Yann Collet. Zstandard - A stronger compression algorithm. http://fastcompression.blogspot.de/2015/01/zstd-stronger-compression-algorithm.html, January 2015. (last accessed: 2015-05-12).

[Col16] Yann Collet. Zstandard. http://cyan4973.github.io/zstd/, July 2016. (last accessed: 2016-08-10).

[Deu96] L Peter Deutsch. DEFLATE compressed data format specification version 1.3. https://tools.ietf.org/html/rfc1951, 1996. (last accessed: 2015-11-01).

[EVKV02] M. Effros, K. Visweswariah, S. R. Kulkarni, and S. Verdu. Universal lossless source coding with the burrows wheeler transform. *IEEE Transactions on Information Theory*, 48(5):1061–1081, May 2002.

[FM98] R. Friend and R. Monsour. IP Payload Compression Using LZS - RFC 2395. https://tools.ietf.org/html/rfc2395, December 1998. (last accessed: 2016-08-25).

[Fod02] Imola K Fodor. *A survey of dimension reduction techniques.* Technical Report UCRL-ID-148494, Lawrence Livermore National Laboratory, 2002.

[fou] Fourierreihe. https://de.wikipedia.org/wiki/Fourierreihe. (last accessed: 2015-11-07).

[Fri04] R. Friend. Transport Layer Security (TLS) Protocol Compression Using Lempel-Ziv-Stac (LZS) - RFC 3943. https://tools.ietf.org/html/rfc3943, November 2004. (last accessed: 2016-08-25).

[FS96] R. Friend and W. Simpson. PPP Stac LZS Compression Protocol - RFC 1974. https://tools.ietf.org/html/rfc1974, August 1996. (last accessed: 2016-08-25).

[Fuc16] Anna Fuchs. Client-Side Data Transformation in Lustre. Master's thesis, Universität Hamburg, 05 2016.

[GA15] Jean-loup Gailly and Mark Adler. gzip. http://www.gzip.org/algorithm.txt, 2015. (last accessed: 2015-11-01).

[Gil04] Jeff Gilchrist. Parallel data compression with bzip2. In *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems*, volume 16, pages 559–564, 2004.

[GV04] Peter Grunwald and Paul Vitányi. Shannon information and Kolmogorov complexity. *arXiv preprint cs/0410002*, 2004.

[HKM$^+$13] Danny Harnik, Ronen I Kat, Oded Margalit, Dmitry Sotnikov, and Avishay Traeger. To Zip or not to Zip: effective resource usage for real-time compression. In *FAST*, pages 229–242, 2013.

[Kat90] Phillip W. Katz. Us 5051745 a - zip. `https://www.google.com/patents/US5051745`, August 1990. (last accessed: 2016-08-26).

[KC15] Per Karlsen and Lasse Collin. LZMA. `http://sourceforge.net/projects/lzmautils/`, 2015. (last accessed: 2015-11-01).

[Kin05] Kingsley G. Morse. Compression Tools Compared. `http://www.linuxjournal.com/node/8051/print`, September 2005. (last accessed: 2016-08-25).

[Kuh16a] Michael Kuhn. Datenreduktion. `http://wr.informatik.uni-hamburg.de/_media/teaching/sommersemester_2016/hea-16-datenreduktion.pdf`, June 2016. (last accessed: 2016-08-16).

[Kuh16b] Michael Kuhn. Moderne Dateisysteme. `http://wr.informatik.uni-hamburg.de/_media/teaching/sommersemester_2016/hea-16-moderne-dateisysteme.pdf`, April 2016. (last accessed: 2016-08-16).

[LA12] R. Lenhardt and J. Alakuijala. Gipfeli - High Speed Compression Algorithm. In *Data Compression Conference (DCC), 2012*, pages 109–118, April 2012.

[Lou11] Phillip Lougher. SquashFS. `http://squashfs.sourceforge.net/`, February 2011. (last accessed: 2016-08-16).

[Lov04] Robert M. Love. taskset(1) - linux man page. `https://linux.die.net/man/1/taskset`, November 2004. (last accessed: 2016-11-09).

[lzj16] LZJB. `https://en.wikipedia.org/wiki/LZJB`, June 2016. (last accessed: 2016-08-25).

[lzw16] Lempel-Ziv-Welch-Algorithmus. `https://de.wikipedia.org/wiki/Lempel-Ziv-Welch-Algorithmus`, January 2016. (last accessed: 2016-08-25).

[MBS13] Dirk Meister, André Brinkmann, and Tim Süß. File recipe compression in data deduplication systems. In *FAST*, pages 175–182, 2013.

[Mei13] Dirk Meister. *Advanced data deduplication techniques and their application.* PhD thesis, Universitätsbibliothek Mainz, 2013.

[MKB$^+$12] Dirk Meister, Jürgen Kaiser, Andre Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. A study on data deduplication in HPC storage systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 7. IEEE Computer Society Press, 2012.

[mov15] Move-to-front. `https://de.wikipedia.org/wiki/Move_to_front`, 2015. (last accessed: 2016-08-16).

[MZSU08] Nagapramod Mandagere, Pin Zhou, Mark A Smith, and Sandeep Uttamchandani. Demystifying Data Deduplication. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, Companion '08, pages 12–17, New York, NY, USA, 2008. ACM.

[NGTJ08] Emily Namey, Greg Guest, Lucy Thairu, and Laura Johnson. Data reduction techniques for large qualitative data sets. *Handbook for team-based qualitative research*, pages 137–161, 2008.

[PK15] Prabhat and Quincey Koziol, editors. *High performance parallel I/O*. CRC Press, 2015.

[Rus69] Enrique H Ruspini. A new approach to clustering. *Information and control*, 15(1):22–32, 1969.

[Sai04] Amir Said. Introduction to arithmetic coding-theory and practice. *Hewlett Packard Laboratories Report*, 2004.

[sev] Seven zip, LZMA. . (last accessed: 2016-08-12).

[Sew10] Julian Seward. Bzip2, September 2010.

[SF96] K. Schneider and R. Friend. PPP LZS-DCP Compression Protocol (LZS-DCP) RFC 1967. `https://tools.ietf.org/html/rfc1967`, August 1996. (last accessed: 2016-08-25).

[SS82] James A. Storer and Thomas G. Szymanski. Data Compression via Textual Substitution. *J. ACM*, 29(4):928–951, October 1982.

[TAN15] Zaid Bin Tariq, Naveed Arshad, and Muhammad Nabeel. Enhanced LZMA and BZIP2 for improved energy data compression. In *Smart Cities and Green ICT Systems (SMARTGREENS), 2015 International Conference on*, pages 1–8, May 2015.

[TGB$^+$13] S. J. Tingay, R. Goeke, J. D. Bowman, D. Emrich, S. M. Ord, D. A. Mitchell, M. F. Morales, T. Booler, B. Crosse, R. B. Wayth, C. J. Lonsdale, S. Tremblay, D. Pallot, T. Colegate, A. Wicenec, N. Kudryavtseva, W. Arcus, D. Barnes, G. Bernardi, F. Briggs, S. Burns, J. D. Bunton, R. J. Cappallo, B. E. Corey, A. Deshpande, L. Desouza, B. M. Gaensler, L. J. Greenhill, P. J. Hall, B. J.

Hazelton, D. Herne, J. N. Hewitt, M. Johnston-Hollitt, D. L. Kaplan, J. C. Kasper, B. B. Kincaid, R. Koenig, E. Kratzenberg, M. J. Lynch, B. Mckinley, S. R. Mcwhirter, E. Morgan, D. Oberoi, J. Pathikulangara, T. Prabu, R. A. Remillard, A. E. E. Rogers, A. Roshi, J. E. Salah, R. J. Sault, N. Udaya-Shankar, F. Schlagenhaufer, K. S. Srivani, J. Stevens, R. Subrahmanyan, M. Waterson, R. L. Webster, A. R. Whitney, A. Williams, C. L. Williams, and J. S. B. Wyithe. The Murchison Widefield Array: The Square Kilometre Array Precursor at Low Radio Frequencies. *PASA - Publications of the Astronomical Society of Australia*, 30, 2013.

[use16] Can i use brotli? `http://caniuse.com/#search=brotli`, July 2016. (last accessed: 2016-08-27).

[Wel85] Terry Welch. High speed data compression and decompression apparatus and method. `https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=19851210&DB=&locale=de_EP&CC=US&NR=4558302A&KC=A&ND=2`, October 1985. (last accessed: 2016-08-25).

[WGHV11] Andreas Wicenec, Derek K Gerstmann, Christopher Harris, and Kevin Vinsen. Integrating HPC into Radio-Astronomical data reduction. In *General Assembly and Scientific Symposium, 2011 XXXth URSI*, pages 1–1. IEEE, 2011.

[Wil91] R. N. Williams. An extremely fast Ziv-Lempel data compression algorithm. In *Data Compression Conference, 1991. DCC '91.*, pages 362–371, April 1991.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

Acronyms

ADIOS	Adaptable I/O System. 31
ANSI	American National Standards Institute. 27
ASCII	American Standard Code for Information Interchange. 41
BST	Binary Search Tree. 28
BWT	Burrows-Wheeler Transform. 17
CPU	Central Processing Unit. 39
EXT	Extended File System. 24
FORTRAN	Formula Translation. 41
FSE	Finiste State Entropy Encoder. 29
HDF	Hierarchical Data Format 5. 43
HPC	High Performance Computing. 2, 6
I/O	Input and Output. 2
ISOBAR	In-Situ Orthogonal Byte Aggregated Reduction. 31
LZ	Lempel-Ziv. 2
LZ4	Lempel-Ziv 4. 28, 40
LZ77	Lempel-Ziv 1977. 24–28, 32
LZ78	Lempel-Ziv 1978. 25
LZJB	Lempel-Ziv Jeff Bonwick. 28, 40
LZMA	Lempel-Ziv Markov Algorithm. 28
LZO	Lempel-Ziv Oberhumer. 27, 40
LZRW	Lempel-Ziv Ross Williams. 27

LZRW1 Lempel-Ziv Ross Welch 1. 28

LZS Lempel-Ziv Stac. 27

LZSS Lempel-Ziv Storer and Szymanski. 26

LZW Lempel-Ziv Welch. 26

MMC Morphing Match Chain. 28

MTF Move-To-Front transform. 16

NetCDF Network Common Data Form. 43

PDF Portable Data Format. 41

POWER Performance Optimization With Enhanced Risc. 11

RFC Request For Comments. 27, 29

RLE Run Length Encoding. 2, 15

SHA256 Secure Hash Algorithm 256. 11

SKA Square Kilometre Array. 7, 10

UTF-8 Unicode Transformation Format 8 bit. 41

ZFS Zettabyte File System. 2, 6, 23, 28

ZLE Zero Length Encoding. 16

ZLIB . 40

ZSTD Zstandard. 2, 40

List of Figures

List of Tables

Listings