Hardware accelerated data compression in wireless sensor networks

Hardware beschleunigte Datenkompression in drahtlosen Sensornetzen
Bachelor-Thesis von Jaco A. Hofmann
August 2012

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Embedded Systems and Applications

Hardware accelerated data compression in wireless sensor networks
Hardware beschleunigte Datenkompression in drahtlosen Sensornetzen

Vorgelegte Bachelor-Thesis von Jaco A. Hofmann

1. Gutachten: Andreas Koch
2. Gutachten: Andreas Engel

Tag der Einreichung:

Erklärung

Hiermit versichere ich gemäß der Allgemeinen Prüfungsbestimmungen der Technischen Universität Darmstadt (APB) §23 (7), die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 13. August 2012

Jaco A. Hofmann

Contents

Abstract

The “Deutsches Primatenzentrum” (DPZ) in Göttingen and the institute for electromechanical construction at TU Darmstadt cooperate in researching the neural activity of apes. One of their experiments requires the apes to move freely in an area of about 300 m^2 while their brain activity is being measured. Thus wireless transmission techniques have to be used to collect measured sensor data for further evaluation.

This work researches compression algorithms suitable for reducing the transmission bandwidth required for these experiments. The aim is to reduce the energy consumption of the measurement platform by enabling the usage of lower power transmission techniques. These lightweight transmission techniques are currently not usable due to their lower bandwidth.

After evaluating various compression algorithms, Adaptive Differential Pulse Code Modulation (ADPCM) proves the most valuable, facilitating a data reduction of up to 36% of the original size. A version of ADPCM with a forward adaptive predictor is implemented on both an FPGA and an MCU. In comparison, the FPGA performs significantly better thus supporting a potentially wider range of application scenarios.

Zusammenfassung

Das “Deutsche Primatenzentrum” (DPZ) in Göttigen und das Institut für Elektromechanische Konstruktion (EMK) der TU Darmstadt untersuchen im Rahmen einer Kooperation die neuronalen Aktivitäten von Affen. Eines dieser Experimente erfordert, dass sich die Affen frei, auf einer Fläche von etwa 300 m^2, bewegen können. Daher wird, zur weiteren Auswertung, auf drahtlose Übertragungstechniken zur Übertragung der gemessenen Sensorwerte zurückgegriffen.

Diese Arbeit untersucht Kompressionsalgorithmen auf die Eignung zur Kompression der Daten zur Reduzierung der benötigten Bandbreite. Das Ziel dieser Kompression ist es, den Energieverbauch der Messplatform zu reduzieren indem, durch die Kompression, Übertragungstechniken mit niedrigerer Bandbreite aber auch niedrigerem Energieverbrauch eingesetzt werden können.

Der Vergleich verschiedener Kompressionalgorithmen stellt Adaptive Differential Pulse Code Modulation (ADPCM) als am besten geeignetes Verfahren heraus. Mit optimierten Einstellungen wird dabei eine Reduzierung auf etwa 36% der Ausgangsgröße erreicht. Zudem wird eine Version von ADPCM mit vorwärts adaptivem Prädikator implementiert. Dabei wird eine Version für ein FPGA erstellt und eine Version für einen Mikrokontroller. Der Vergleich der beiden Implementierungen zeigt, dass das FPGA signifikant besser geeignet ist und dadurch einen größeren Anwendungsbereich unterstützen kann.

1 Introduction

Wireless sensor networks (WSN) consist of wirelessly connected nodes. Typically, these nodes are designed to work with as little power as possible to allow for maintenance-free run-times of up to a year. WSN applications are widely spread.

Figure 1.1: Common wireless sensor network platform Tmote Sky. This platform provides a TI MSP430F1611 microcontroller, 48 kbyte Flash and a 250 kbps wireless transceiver. Furthermore, it features various possibilities to access sensors. With as little as 15.3 μW power consumption in sleep mode this device can operate without human interaction for long times. Source: `http://www.snm.ethz.ch/snmwiki/pub/uploads/Projects/tmote_sky_small.jpg`

They range from unintrusive animal observation to forest fire detection and E-Health patient observation. Thus, wireless sensor networks are a useful tool in many research areas. While the first WSN were mostly used to monitor larger areas, recent WSN are also used in smaller areas like buildings or research labs. As most of the sensor nodes operate on batteries, energy consumption is a major restriction for the sensor networks operation. When observing the energy consumption of wireless sensor networks, a significant amount of energy is spent for radio transmission. Calculations on the other hand are relatively cheap. Many different techniques evolved around the problem of high power consumption of wireless transmissions. One of these are routing algorithms that try to transmit data in the most efficient way through the network. Another approach is the local processing of the measured data in the sensor node itself. For this approach the sensor data has to be analysed and the crucial events have to be classified by algorithms like hidden markov models, neural networks or simple thresholds. If the data analysis depends on combining information from multiple sensors, this approach is not viable.

The approach that is researched in this work is more general. Compression algorithms are widely used in most networks nowadays. They aim at reducing the size of the data to be transmitted by changing its binary representation in a unique and reversible fashion. A simple example is the so called run-length encoding used for fax transmissions. Instead of storing each pixel of the original image, a series of equally coloured pixels is encoded as a tuple (length, colour). Many more sophisticated algorithms exist. A downside of many of these approaches are their large memory and processing power requirements. For WSN new approaches have to be investigated or old approaches tested for viability.

The work is structured as follows: Chapter 2 introduces the application scenario and the features of the data to be compressed. Chapter 3 provides a basic understanding of various compression algorithms. Chapter 4 compares the application of these algorithms to the data presented in Chapter 2. The hardware- and software-based implementation of the most valuable compression algorithm is described in Chapter 5. Finally, the performance of these implementations is analyzed and compared in Chapter 6. The last chapter concludes this work and provides a short outlook onto future tasks.

2 The Application

This chapter describes the application a suitable data compression algorithm is sought for. The chapter begins with a description of the application scenario including relevant constraints in Section 2.1. In Section 2.2 the data to be compressed is described and relevant features are shown.

2.1 Application Scenario

This work is motivated by a cooperation between the “Deutsches Primatenzentrum” (DPZ) in Göttingen and the institute for electromechanical construction at TU Darmstadt. The aim of this cooperation is to develop a mobile system for measuring the neural activities of apes. While the apes solve different tasks, the neural activity is measured and transmitted to a centralized processing unit for further analysis. One session lasts for over one hour. Due to high data rates, in node logging and analysis are not viable. Furthermore the ape should be able to move freely inside the testing area of about 300 m^2. Thus cables are impractical and wireless transmissions are needed.

The measurement of brain activities is carried out by four micro-electrodes inside the brain. Each of these electrodes has to be adjusted to a certain depth within the brain of the ape during a session to monitor different types of brain activities [12]. The adjustion is performed by a remotely controlled system. For adequate electrode positioning by a human operator, the data transmission delay needs to be less than about 100 ms. At the same time, the data transmission has to be fast enough to transmit four channels of 16 bit samples at 24 414 Hz (1.5 Mbit/s).

The area for all components is limited to 3 cm × 4 cm × 2.5 cm and the weight is limited to 40 g.

Currently a WLAN transceiver are used for data transmissions. Its comparably high energy consumption enlarges the batteries required to meet the run-time constraints. This results in a higher area usage. Lightweight transmission techniques like Bluetooth or Zigbee are not able to provide the required bandwidth. Thus, a compression method needs to be found that allows for a transmission bandwidth reduction and thus lowers energy consumption and weight of the components.

The following section will describe the exact features of the transmitted data.

2.2 Application data

The data generated by the sensors consists of 16 bit signed numbers.

For testing and evaluation, data from actual experiments is used. This data consists of about 130 million samples per channel for four channels in total. This corresponds to about 87 hours of logging. The statistical properties of the channels vary as can be seen in table Table 2.1. While the minima and maxima remain relative consistent over all channels, both the variance as well as the mean varies greatly. All means lie within

Channel	Minimum	Maximum	Variance	Mean
1	-2799	3623	8456.494224	-277.731958
2	-2505	3882	34709.583269	17.994374
3	-1925	2026	2935.727405	88.559409
4	-2464	3426	2356.920941	-149.533500

Table 2.1: Statistical properties of the data provided by the ape experiment.

0.0085% of the maximum value for signed 16 bit block encoded numbers of $2^{15} - 1$. Most of the samples lie within the range of $[-1000, 1000]$ as can be seen in Figure 2.1 for all channels.

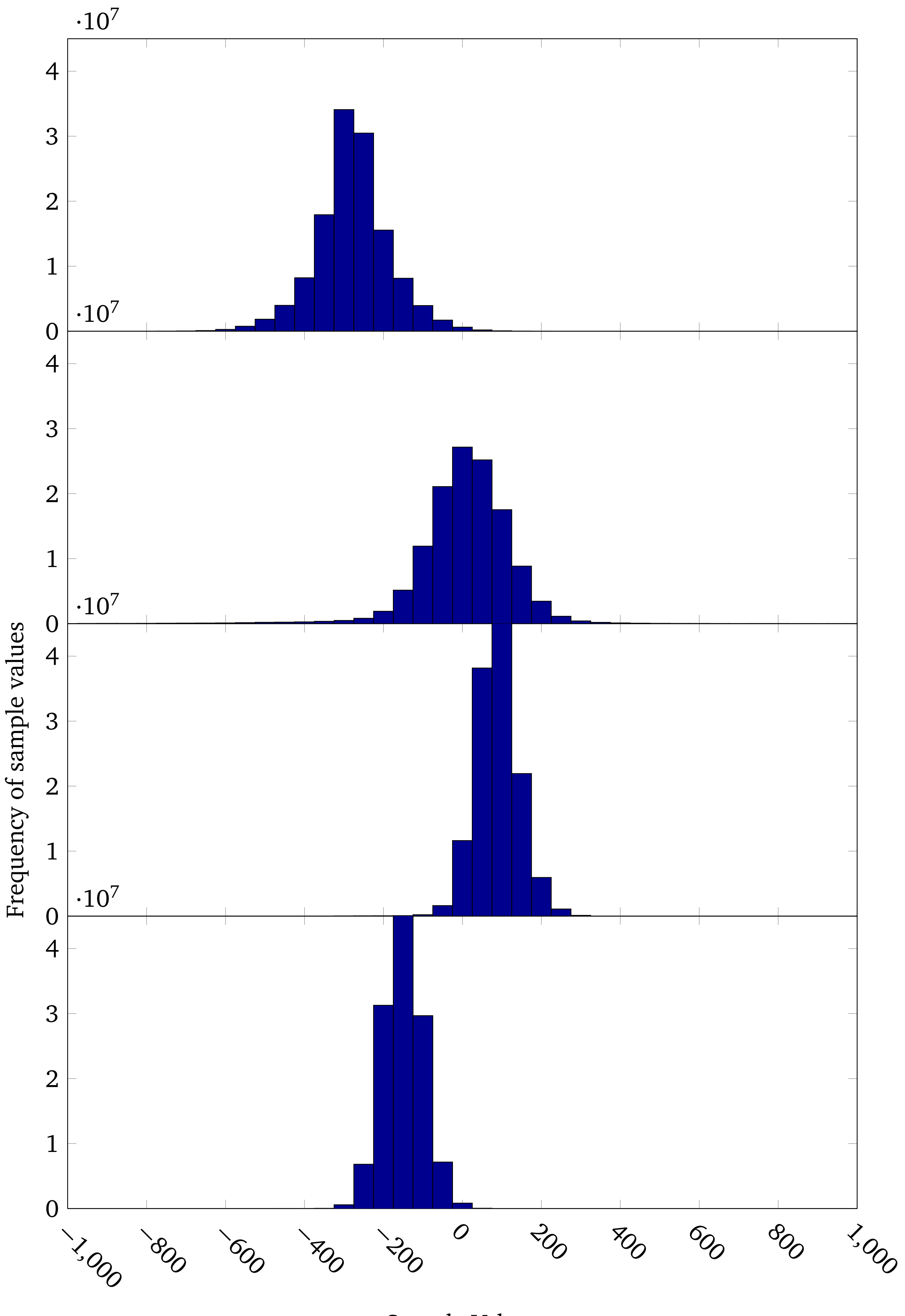


Figure 2.1: Histograms of the data channels provided by the ape experiment.

3 Introduction to Data Compression

Data compression is the modification of the binary representation of a data sequence aimed at lowering the resources required to store or transmit the data. Two fundamental approaches can be distinguished:

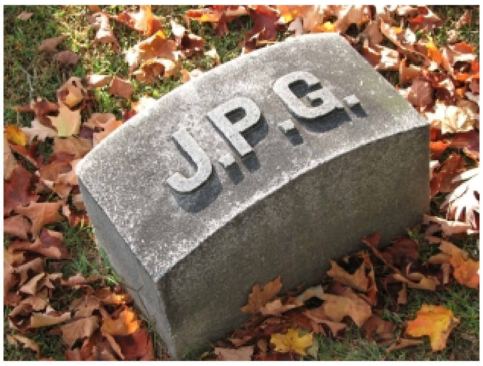
1. **Lossless compression** reduces the data representation size without losing any of the original information. This is commonly used for textual data not tolerating any deviations after decoding. transmissions can not be tolerated.

2. **Lossy compression** reduces the data representation size utilizing the limited human perception. This is very useful for visual and audio data, as many parts of the signals captured by cameras and microphones are not perceivable by humans and can be removed without changing the experience for the consumer.

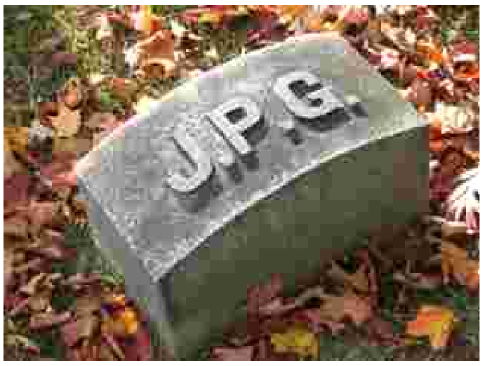
Typically, lossy compression is able to achieve much higher compression rates compared to lossless compression. Figure 3.1 shows a picture compressed with a lossy compression method at high (Figure 3.1a) and low (Figure 3.1b) quality levels. While most of the information can be perceived at just 5.7 % of the size, the effects of the compression algorithm is clearly visible in Figure 3.1b. Hard edges are blurred, and many artifacts can be observed.

Compression algorithms can be divided in two main groups. The first group exploits statistical properties of the source to reduce redundancy. This is done by converting from the source alphabet into another alphabet, often referred to as symbol alphabet. This group will further be referenced as source encoding. The second group of algorithms tries to encode the symbols of an alphabet with the least possible symbols in the target alphabet (for example bits). This group will be referenced as symbol encoding.

Typically, a compression method uses both types of encoding. First a symbol encoding is used to convert from the source alphabet into a symbol alphabet with less



(a) Highest quality (Q = 100) ~83 kB[16]



(b) Low quality (Q = 10) ~5 kB[15]

Figure 3.1: Two JPEG compressed images with different encoding settings.

Notation	Meaning	Definition
Alphabet	A set of distinguishable symbols. A capital letter in fractal font indicates an alphabet.	$\mathfrak{A}$
Letter, Symbol	Smallest piece of data carrying information. Element of an alphabet	$a \in \mathfrak{A}$
Word	Limited series of letters of an alphabet	$(a_1, a_2, \ldots, a_n), a_i \in \mathfrak{A}$
Compression rate	Size of the compressed bit string in percent of the original bit string.	$\frac{\text{length compressed}}{\text{length uncompressed}} \cdot 100$
$P(a)$	Probability for the occurrence of letter a.	$P : \mathfrak{A} \to [0,1]$
$B(a)$	Binary representation of letter a.	$B : \mathfrak{A} \to \{0,1\}^*$
$B_l(a)$	Binary representation of letter a with length l.	$B_l : \mathfrak{A} \to \{0,1\}^l$
$\bullet$	Concatenation of two bit strings.	$B(a) \bullet B(b) = B(a)B(b)$
$u(n)$	Unary code of n (see Section 3.3.3)	$u : \mathbb{N} \to \{0,1\}^*$

Table 3.1: Table of the notations used throughout this chapter.

redundancy. Afterwards, the symbol alphabet is conviërted into a (typically binary) target alphabet.

3.1 Overview of Compression Algorithms

Over the years various compression algorithms evolved. In the following section, a brief description of the basic ideas of the algorithms used in Chapter 4 is given. The information theoretical background will only be scratched as this work is mainly interested in the output performance of the algorithms. Thus, the description of the algorithms itself will focus on the elementary ideas behind each approach, their requirements as well as their advantages and disadvantages. For further information, the readers may refer to [13] and [14]. Table 3.1 describes notations, functions and fundamental terms used throughout this chapter.

3.2 Source Encoding

The algorithms described in this section exploit the statistical properties of the source to reduce redundancy. The alphabet of the resulting symbols typically differs from the source alphabet.

3.2.1 Run-length Encoding (RLE) $\mathfrak{A}^* \rightarrow (\mathbb{N} \times \mathfrak{A})^*$

Run-length encoding is a very simple algorithm which works best on very repetitive data. It represents reoccurring sequences of single letters (a a . . . a) as a tuple (n, a) containing the sequence length n and the repeated letter a. The decoding process reads the number n and outputs the following letter n times. To limit the amount of output symbols, the length of the sequences must be restricted to an upper bound N:

$$\mathfrak{B} = \{(n, a) | a \in \mathfrak{A}, n \in \mathbb{N}, n \leq N\} \tag{3.1}$$

The following examples use the two letter source alphabet $\mathfrak{A} = \{a_1, a_2\}$. The string

$$\begin{array}{l}
a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1 \\
a_1a_1a_1a_2a_2a_2a_2a_2a_2a_2a_2a_2a_2a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_1a_2a_2a_2a_2 \\
a_2a_2a_2a_2a_2a_2
\end{array}$$

takes up 64 letters. For $N \geq 32$, this string can be run length encoded as $32a_1 10a_2 12a_1 10a_2$, which are only four symbols long. With $N = 16$ the string will be encoded as $16a_1 16a_1 10a_2 12a_1 10a_2$. Run length encoding of short letter sequences increase the symbol sequence length (e.g.: $\text{RLE}(a_1a_2a_1a_2a_1) = (1, a_1)(1, a_2)(1, a_1)(1, a_2)(1, a_1))$. More sophisticated RLE approaches address issues like the efficient representation of non repeated letters to further improve the compression performance.

A big advantage in memory constrained systems is that the RLE needs little to none additional memory during encoding and decoding.

RLE is used in various image formats and is very effective for pictures with large homogeneous colored areas such as (black and white) fax transmissions.

This simple algorithm is very ineffective for noisy data like that received by real world sensors, because noise will intercept most of the consecutive sequences. As proposed in [4, p. 102], noisy sensor signals can be prepared for RLE by reducing the quantization accuracy of the sampled data. This scheme results in loss of information and is therefore no viable option for this work.

As this approach is very inefficient for lossless sensor data compression it will not be used standalone in the comparison of Chapter 4 but maybe as part of a different algorithm.

3.2.2 LZ-Family

Lempel and Ziv published the dictionary based algorithm LZ77 in May 1977 [17] and an improved version called LZ78 one year later [18]. There are many variants of these dictionary approaches such as LZW, LZSS and LZMA which may greatly vary in data representation but share the same basic principles.

LZ77 is described in the following paragraphs as described in [13].

LZ77 Compression $\mathfrak{A}^* \rightarrow (\mathbb{N} \times \mathbb{N} \times \mathfrak{A})^*$

LZ77 compression exploits the repetitive structure of textual data. To represent a prefix of the text to be encoded, LZ77 searches the already encoded text for the longest matching sequence. Every encoded symbol is a triple (p, l, a) indicating the position p and the length l of the matching sequence. The position is specified relative to the current encoding position. The letter a is the one following the actual sequence. Its inclusion in the output symbol ensures the progression of the encoding process whenever no matching sequence could be found ($l = 0$). The detailed algorithm is presented in Algorithm 1.

Algorithm 1 LZ77 encoding steps.

Require: s is a string, searchspace is an integer, lookahead is an integer

```
 1: function LZ77_ENCODE(s,searchspace,lookahead)
 2:     $r \leftarrow []$                          ▷ List of tuples to hold the encoded input string
 3:     $i \leftarrow 0$
 4:     while $i \leq$ LEN(s) do
 5:         $p, l \leftarrow$ FINDLONGESTSUBSTRING(s[i:i+lookahead],s[i-searchspace:i-1]) ▷ Find the longest substring that matches the start of the first argument in the second argument.
 6:         PUSH(r,($p$, $l$, s[$i + l$]))            ▷ Add the found reference to the output.
 7:         $i \leftarrow i + l + 1$
 8:     end while
 9:     return r
10: end function
```

To reduce the amount of memory and computing power used to find matching character sequences, a sliding window implementation is used. This window limits the search space to an area around the current character instead of searching the whole previous string.

The encoding of a single character is very wasteful (As two zeros and the character itself have to be encoded) in the original LZ77 algorithm. The following example describes the encoding of the string “012120122102120” over the alphabet $\{0, 1, 2\}$ using LZ77 three character wide search buffer. The first element the algorithm reads is 0. As the search buffer is empty, the it is encoded as the tuple $(0, 0, 0)$. The next two letters are encoded the same way, as no match can be found in the search buffer. After the third input letter, the output symbols $[(0, 0, 0), (0, 0, 1), (0, 0, 2)]$ have been produced. With the fourth letter the first time a reference is saved as the sequence 12 is found at position 1 which corresponds to a relative position of 2, with length 2. Thus, $(2, 2, 0)$ is added to the list with 0 being the letter following 12. The search position is increased by 3. The next characters are encoded as a reference to position 4 length 2. The same string 12 occurs at position 1 but it is out of the sliding-window of 3. Thus letters 7, 8 and 9 are encoded as $(4,2,2)$. This process continues till all chars are encoded. The final list is

$$[(0,0,0),(0,0,1),(0,0,2),(2,2,0),(3,2,2),(3,1,0),(3,2,2),(0,0,0)].$$

LZ77 Decompression

The list of tuples produced by the compression step can be decoded easily. For each tuple (p,l,a), the sequence of length l located at position p in the already decoded text is suffixed by a and copied to the output. This process is illustrated in Algorithm 2.

Algorithm 2 Basic LZ77 decoding steps.

Require: r is a list of tuples

```
1: function LZ77DECODE(r)
2:     $s \leftarrow 0$
3:     $p \leftarrow 0$
4:     while !EMPTY(r) do
5:         $l, p, c \leftarrow$ POP(r)
6:         APPEND(s,s[p:p+s]+c) ▷ Copy the referenced chars to the end of the string.
7:     end while
8:     return s
9: end function
```

To resume the previous example, the following tuples have to be decoded:

$$[(0,0,0),(0,0,1),(0,0,2),(2,2,0),(3,2,2),(3,1,0),(3,2,2),(0,0,0)].$$

Each tuple is processed one after the other. The first element $(0,0,0)$ corresponds to a single character as well as element two and three. These are simply inserted into an empty string yielding “012”. Element four references a two letter string starting two positions prior to the current position which is 12. Thus 12 is suffixed by 0 and added to the output string yielding “012120”. This procedure is repeated until the final string is is “012120122102120”.

Improved Variants of LZ77

One of the main problems of LZ77 is the need to encode a single character as a triple symbol. Improved versions like LZSS ($\mathfrak{A}^* \rightarrow (B \times \mathfrak{A} \cup B \times \mathbb{N} \times \mathbb{N})^*$) solve this problem. LZSS does not replace a string by a dictionary reference if the reference tuple would be longer than the string itself. Furthermore, one bit in each tuple indicates weather the tuple carries a letter from the source alphabet or a reference to a dictionary entry.

While LZ77 checks the previous characters in the stream for reoccurring sequences, LZW ($\mathfrak{A}^* \rightarrow (\mathbb{N} \times \mathfrak{A})^*$) creates a dictionary during processing. The dictionary is already initialised with all letters of $\mathfrak{A}$. Instead of storing the reference as a triple, only a dictionary index and the character following the referenced dictionary entry is included in the output symbol. The string decoded by the new symbol is inserted into the dictionary afterwards.

```
['^abraca|',
 'abraca|^',
 'braca|^a',
 'raca|^ab',
 'aca|^abr',
 'ca|^abra',
 'a|^abrac',
 '|^abraca']
```

(a) Step one: Create list of all rotations of the word to encode.

```
['^abraca|',
 'abraca|^',
 'aca|^abr',
 'a|^abrac',
 'braca|^a',
 'ca|^abra',
 'raca|^ab',
 '|^abraca']
```

(b) Step two: Sort the list alphabetically.

```
'|^rcaaba'
```

(c) Step three: Save the last letter of each element of the list as the encoded word.

Figure 3.2: Steps of the BWT encoding of "^abraca|". This example is taken from [3].

3.2.3 Burrows-Wheeler Transform (BWT) $\mathfrak{A}^* \rightarrow \mathfrak{A}^*$

The Burrows-Wheeler Transform, proposed 1994 by Burrows and Wheeler in [3], itself is not a compression algorithm, but BWT prepares the data such that the output is easier to compress. This is achieved by transforming the input into a permutation of itself an increased probability of character repetitions. BWT is reversible, the input stream can easily be reconstructed from the output [1]. The stream itself has to be divided into blocks. The size of a block can be chosen in order to trade-off between the memory and processing power requirements and the character repetition probability.

To perform the BWT encoding, all rotations of the input string i must be computed and sorted alphabetically. The BWT function then outputs the last letters of the respective rotations. This algorithm is summarized in Algorithm 3.

Algorithm 3 Basic encoding steps for the Burrows-Wheeler Transform.

Require: s is a string

```
1: function BWT_ENCODE($s$)
2:     $v \leftarrow$ ROTATIONS($s$)        ▷ Returns all rotations of string $s$ in a list.
3:     SORT($v$)                          ▷ Sorts the list of strings $v$ alphabetically.
4:     return LASTLETTER($v$)   ▷ Returns a string of all last letters of the strings in list
       $v$, while keeping the order.
5: end function
```

To increase the decoding performance, an improved version was proposed in [3]. It sourrounds the input block by a start and ana end mark not included in the input alphabet (for example '^' and '|'). Figure 3.2a shows the permutations of the input block abraca and Figure 3.2b the sorted ones. The resulting BWT output is presented in Figure 3.2c. To decode this sequence, it is split into a list of characters as shown in Figure 3.3a. This list is saved for further use and poses also the first step of the decoding process. The list is sorted alphabetically. This sorted list is prepended by

```
['|',
 '^',
 'r',
 'c',
 'a',
 'a',
 'b',
 'a']
```

(a) Step one: Create a list of strings containing each letter of the encoded string.

```
['|^',
 '^a',
 'ra',
 'ca',
 'ab',
 'ac',
 'br',
 'a|']
```

(b) Step two: Sort the list alphabetically and append the letters of the encoded string to the front.

```
['^abraca|',
 'abraca|^',
 'aca|^abr',
 'a|^abrac',
 'braca|^a',
 'ca|^abra',
 'raca|^ab',
 '|^abraca']
```

(c) Step three: All letters have been decoded. The result is the one string that ends with the string end mark '|'.

Figure 3.3: Steps of the BWT decoding of "^abraca|".

the saved original list, see Figure 3.3b. The process of sorting and preprending is continued till the original string length has been reached. The solution is the string with the string end mark at the last position, see Figure 3.3c. Without the end of

Algorithm 4 Basic decoding steps for the Burrows-Wheeler Transform.

Require: e has to be a Burrows-Wheeler Transformed string

```
1: function BWT_DECODE($e$)
2:     $r \leftarrow$ SPLIT($e$)            ▷ Create a list of characters from a string
3:     $s \leftarrow r$
4:     for $i \leftarrow l, 0$ do
5:         SORT($s$)
6:         APPENDFRONT(s,r)    ▷ Append list of characters $r$ to front of list of strings $s$
7:     end for
8:     return Element of list s that ends with '|'
9: end function
```

string mark the index of the original string has to be saved in the encoding step. The algorithm is presented in Algorithm 4.

When looking at the result, it seems like the first letter of each word would be a better choice for the encoded word as more letters are repeated. However, the BWT transformation would not be reversible in this manner.

Realistic block sizes can easily reach up to 900 KByte. For large blocks a considerable amount of processing power is needed to sort the lists. Furthermore, the memory requirements are growing quadratically with the block size.

3.2.4 Differential Pulse Code Modulation (DPCM) $\mathfrak{A}^* \to \mathfrak{A}^*, \mathfrak{A} \subseteq \mathbb{R}$

Differential pulse code modulation is a technique to reduce the variance of the input stream enabling subsequent symbol encoding algorithms to achieve better compression rates. The input alphabet is restricted to real numbers.

DPCM is commonly used in speech, image and similar encoding. The typically included quantizers cause a loss of information and will not be discussed. Instead, this work focuses on the prediction techniques of DPCM.

For typical sensor data, the sequence of inter-sample differences is zero-mean and has a smaller variance than the sample sequence itself [13, p. 325].

This can be illustrated by the following exemplary sample sequence

$$s = (-309, -306, -294, -274, -266, -264)\,. \tag{3.2}$$

The variance of the sequence is 403.9 and the mean is −285.5. For the sequence of differences

$$s_d = (3, 12, 20, 8, 2) \tag{3.3}$$

the variance decreases to 54 and the mean to 9.

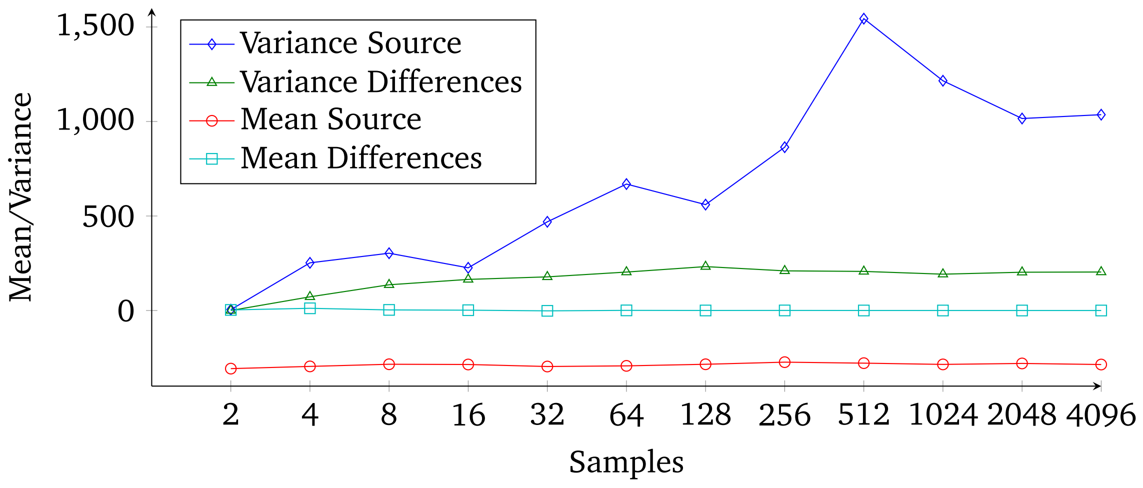


Figure 3.4: Comparison of the mean and the variance between a sequence and its corresponding sequence of differences over the course of different amounts of samples from the primate experiments described in Chapter 2.

Figure 3.4 shows the mean and variance of sequences of different lengths and the same statistical properties of their sequences of differences. As can be seen, the features proposed earlier pose right for this data. The mean of the differences is close to zero. The variance is significantly lower than the variance of the source.

When the sequence of differences and the first element of the sequence is known to the decoder, then the original series can be recovered by simple additions, as long as no quantizer is used.

The performance of this encoding step can be increased by a more sophisticated prediction of the original sequence. The conversion can be written as

$$d_j = x_j - \sum_{i=1}^{n} a_i \cdot x_{j-i}, \tag{3.4}$$

where a_i are prediction coefficients, that determine the impact of previous samples, and n is the prediction order. It is possible to optimise the coefficients $a_1, \dots, a_n$ for best prediction results. Mathematically, the optimisation problem can be written as

$$RA = P \tag{3.5}$$

$$R = \begin{pmatrix} R_{xx}(0) & R_{xx}(1) & R_{xx}(2) & \dots & R_{xx}(n-1) \\ R_{xx}(1) & R_{xx}(0) & R_{xx}(1) & \dots & R_{xx}(n-2) \\ R_{xx}(2) & R_{xx}(1) & R_{xx}(0) & \dots & R_{xx}(n-3) \\ \vdots & \vdots & \vdots & & \vdots \\ R_{xx}(n-1) & R_{xx}(n-2) & R_{xx}(n-3) & \dots & R_{xx}(0) \end{pmatrix} \tag{3.6}$$

$$A = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{pmatrix} \tag{3.7}$$

$$P = \begin{pmatrix} R_{xx}(1) \\ R_{xx}(2) \\ R_{xx}(3) \\ \vdots \\ R_{xx}(n) \end{pmatrix} \tag{3.8}$$

where $R_{xx}(k)$ is the autocorrelation function of x

$$R_{xx}(k) = \frac{1}{N-k} \sum_{i=k}^{N} x_i x_{i-k} \tag{3.9}$$

[13, pp. 333,334]. The solution of this equation poses the optimal coefficients a_i for this sequence. The prediction order has to be selected manually and application specific.

As the prediction coefficients are calculated before they are applied to the prediction process, this scheme is known as forward adaptive prediction (ADPCM-APF). It assumes two passes through the input thus increasing the compression latency. In contrast, the backward adaptive prediction (ADPCM-APB) adopts the coefficients after they where applied to the prediction. For this purpose the least mean square algorithm are commonly used. It can pose quite a challenge to receive good results because the LMS algorithm is very unstable with badly chosen initial parameters.

Apart from these two adaptive approaches many DPCM forms evolved for different applications.

Letter	E	A	R	I	O	T	N	S	L	C
Probability	11.2	8.5	7.6	7.5	7.2	7	6.7	5.7	5.5	4.5

Table 3.2: Probabilities of the ten most common letters in words of the English language [11] (case insensitive).

3.3 Symbol Encoding

This section describes algorithms that encode the symbols of an alphabet with a sequence of symbols in a target alphabet. Their aim is to achieve the smallest possible representation of a word of source letters in a word of target letters.

3.3.1 Huffman-Coding

Huffman-Coding exploits the varying occurrence probabilities of different letters in non-random data. For example, the letter 'e' is more than twice as common as the letter 'c' in words of the English language as seen in table Table 3.2. With the commonly used block encoding like ASCII, every letter would take up the same amount of bits. Huffman-Coding encodes the letters of the alphabet $\mathfrak{A}$, with a given probability distribution, such that

1. A more probable letter is encoded with fewer bits than one with lower probability.
2. The two letters with the lowest probabilities are encoded with the same amount of bits.

The generated code can be used to encode any string of $\mathfrak{A}$ but is only optimal if the probability distribution remains unchanged.

The code generation is recursively done starting with the two letters with the lowest probabilities a_n and a_{n-1}. Thereafter, an alphabet $\mathfrak{A}'$ is derived from $\mathfrak{A}$.

$$\mathfrak{A}' = \mathfrak{A} \setminus \{a_n, a_{n-1}\} \cup \left\{a'_n\right\} \tag{3.10}$$

$$P\left(a'_n\right) = P\left(a_n\right) + P\left(a_{n-1}\right) \tag{3.11}$$

and the binary codes of a_n and a_{n-1} are chosen, as

$$B(a_n) = B(a'_n) \bullet 0 \tag{3.12}$$

$$B(a_{n-1}) = B(a'_n) \bullet 1. \tag{3.13}$$

The process proceeds with alphabet $\mathfrak{A}'$ in the same manner. The merging stops when the derived alphabet was reduced to a single element.

Letter	Probability
a_1	0.6
a_2	0.2
a_3	0.1
a_4	0.1

(a) Probabilities of letters in alphabet $\mathfrak{A}$

Letter	Probability
a_1	0.6
a_2	0.2
a'_3	0.2

(b) Probabilities of letters in alphabet $\mathfrak{A}'$

Letter	Probability
a_1	0.6
a'_2	0.4

(c) Probabilities of letters in alphabet $\mathfrak{A}''$

Table 3.3: The different derived alphabets during the Huffman encoding.

Example code generation

Let $\mathfrak{A} = \{a_1, a_2, a_3, a_4\}$ with the probabilities shown in table Table 3.3a.

The first step is searching for the two letters with the lowest probability which are a_3 and a_4 with a probability of 0.1 each. The assigned codes are $B(a_3) = B(a'_3) \bullet 0$ and $B(a_4) = B(a'_3) \bullet 1$. The next alphabet processed is shown in Table 3.3b. Again the two least common letters are substituted which are a_2 and a'_3 with a probability of 0.2 each. They are coded as $B(a_2) = B(a'_2) \bullet 0$ and $B(a'_3) = B(a'_2) \bullet 1$. As there are only two elements left in alphabet $\mathfrak{A}''$ (Table 3.3c) these two are processed. The codes generated are $B(a_1) = B(a'_1) \bullet 0$ and $B(a'_2) = B(a'_1) \bullet 1$. After the final symbol combination, the occurrence of a'_1 carries no information $\left(P(a'_1)\right) = 1$ and is therefore encoded with an empty string: $B(a'1_1) = \epsilon$. Now the recursively defined codes can be expanded to

$$B(a_1) = B(a'_1) \bullet 0 = 0 \tag{3.14}$$
$$B(a_2) = B(a'_2) \bullet 0 = B(a'_1) \bullet 10 = 10 \tag{3.15}$$
$$B(a_3) = B(a'_3) \bullet 0 = B(a'_2) \bullet 10 = B(a'_1) \bullet 110 = 110 \tag{3.16}$$
$$B(a_4) = B(a'_3) \bullet 1 = B(a'_2) \bullet 11 = B(a'_1) \bullet 111 = 111. \tag{3.17}$$

As stated above, this code fulfills the constraints proposed for an optimal code. The element with the highest probability a_1 has the shortest bit representation. The two elements with the lowest probability a_3 and a_4 are encoded with the same amount of bits.

Encoding samples

For comparison, the string $a_3a_1a_2a_1a_1a_1a_2a_4a_1a_1$, which matches the probability distribution in table Table 3.3a, is encoded once with a static encoding B_2 and once with the Huffman-Code created in the preceding section (Table 3.4). Both strings are presented in Table 3.5. The compression ratio

$$\frac{\text{size compressed}}{\text{size uncompressed}} = \frac{16}{20} = 0.8$$

Letter	a_1	a_2	a_3	a_4
B_2	00	01	10	11
Huffman	0	10	110	111

Table 3.4: Letters of $\mathfrak{A}$ encoded with B_2 and the Huffman code generated in Section 3.3.1.

Name	Code	Size in bit
Static	11 00 01 00 00 00 01 10 00 00	20
Huffman	110 0 10 0 0 0 10 111 0 0	16

Table 3.5: The string $a_3a_1a_2a_1a_1a_1a_2a_4a_1a_1$ encoded with a static B_2 and a optimal Huffman encoding.

indicates a size reduction of 20 %.

If the encoded string does not follow the probability distribution used to create the Huffman code, the Huffman-Coding may be more inefficient than block encoding. The String $a_1a_2a_3a_4$ has a distribution of $P(a_1) = P(a_2) = P(a_3) = P(a_4) = \frac{1}{4}$ is now encoded with the same codes as above in Table 3.6.

This is a size increase by 12.5 %.

The Huffman code is a prefix free code. That means that no code is a prefix of another code. Thus, the generated code table can be represented as a binary tree whose leaves are the symbols of the input alphabet $\mathfrak{A}$ (see Figure 3.5). The decoder starts at the root of this tree and steps down to the left or the right subtree depending on the next bit read from the input stream. When reaching a leave, the corresponding symbol is written to the decoder output and the tree walk restarts from the root. This is repeated till the whole input stream is decoded.

A major disadvantage of Huffman codes becomes visible here. Every single bit error may corrupt the whole decoding process.

Adaptive Huffman-Coding

Another major problem of Huffman-Coding is the necessity to know the probability distribution of the data source to achieve the optimal compression ratio. Thus, a two

Name	Code	Size in bit
Static	00 01 11 10	8
Huffman	0 10 110 111	9

Table 3.6: The string $a_1a_2a_3a_4$ is encoded with a suboptimal Huffman and a static B_2 encoding.

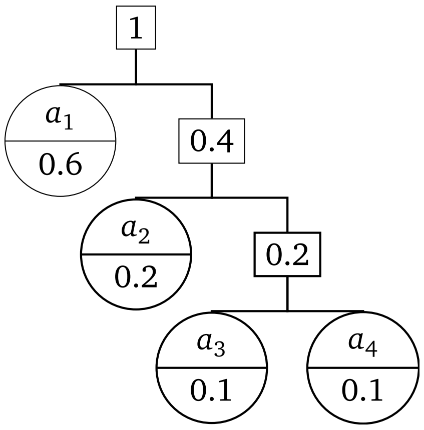


Figure 3.5: Huffman code represented as Tree

pass approach is required where the distribution and the code tree is calculated in the first pass and the second pass encodes the source. This two pass approach requires memory to store the input till the encoding is done. Furthermore, it adds a delay as the input has to be buffered completely before the first output can be generated. To fix this problem, one-pass adaptive algorithms to construct the Huffman code were developed by Gallager [7] and Faller[6] independently [13, p. 58]. These algorithms create the Huffman tree while processing the input data stream. Each letter not yet included in the tree is added at an algorithm-specific position. Furthermore, the letter is transmitted to the decoder. If an input symbol is already in the tree, the corresponding code is transmitted instead.

3.3.2 Arithmetic-Coding

Arithmetic-Coding is a symbol coding well suited for situations where the size of the input alphabet and the probabilities for a single letters are far apart [13, p. 81]. This would result in inefficient encoding using Huffman.

Arithmetic-Coding maps every sequence of letters over an alphabet to an interval in $[0,1)$. Every letter of the source alphabet gets a tag attached corresponding to a specific interval. The tag $T(a_i)$ for letter a_i is generated by the formula

$$T(a_i) = \sum_{k=1}^{i-1} P(a_k) + \frac{1}{2} P(a_i) \tag{3.18}$$

[13, p. 86]. To map the tag to a binary code, the binary representation of $T(a_i)$ is truncated to a length of

$$\left\lceil \log_2 \frac{1}{P(a_i)} \right\rceil + 1 \tag{3.19}$$

bits. Table 3.7 shows the bit representations for the alphabet $\mathfrak{A} = \{a_1, a_2, a_3\}$ using arithmetic encoding. The decoding process consists of looking up the code in a look-up-table. Arithmetic coding can be quite inefficient when encoding just one letter at a time but gets more and more efficient when encoding multiple symbols at once.

Letter	T	Binary	Truncated size	Truncated
a_1	.4	$.0\overline{1100}$	2	01
a_2	.85	$.11011\overline{0011}$	5	11011
a_3	.95	$.1111\overline{0011}$	5	11110

Table 3.7: An example code for the three letter alphabet $\mathfrak{A} = \{a_1, a_2, a_3\}$ with $P(a_1) = 0.8$, $P(a_2) = 0.1$, $P(a_3) = 0.1$ encoded with arithmetic coding

3.3.3 Golomb-Rice-Coding

The Golomb-Rice-Coding is used to encode natural numbers when the probability for higher numbers is lower than the probability for lower numbers. This is often the case for the output of predictive compressions like ADPCM.

Unary code

The unary code is optimal for encoding natural numbers with the probability distribution

$$P(k) = \frac{1}{2^k}$$

It equals the Huffman code for this probability model and the semi-infinite alphabet $\{1, 2, 3, \ldots\}$ [13, p. 66]. The unary code encodes a natural number n as n ones followed by one zero.

$$u(1)010 \tag{3.20}$$

$$u(3) = 1110 \tag{3.21}$$

$$u(11) = 111111111110 \tag{3.22}$$

Golomb code

The Golomb code described in [8] by Golomb is parametrized by a natural number m. It encodes a natural number n as a concatenation of $u(\frac{n}{m})$ and $B(n\%M)$.
Rice code denotes a subset of the Golomb codes where m is a power of two. Only this subset will be presented here. Formally a Rice code is encoded as follows

$$q(n, m) = \left\lfloor \frac{n}{m} \right\rfloor \tag{3.23}$$

$$r(n, m) = n - q(n, m)m \tag{3.24}$$

$$g(n) = u(q) \bullet B_k(r), \tag{3.25}$$

for $m = 2^k$.

Example encoding for $m = 4$ are displayed in Table 3.8.

For Rice-coding shifts and masks can be used instead of divisions and modulo operations. The algorithm greatly simplifies to

$$g(n) = u(n \gg k) \bullet (n\&(m - 1)). \tag{3.26}$$

n	q	r	g(n)
1	0	1	0 01
3	0	3	0 11
5	1	1	10 01
10	2	2	110 10

Table 3.8: Examples of a Golomb code with $m = 4$

3.3.4 Tunstall Codes

The other codes described so far like Golomb-Rice-Coding or Arithmetic-Coding map one letter or a sequence of letters of the source alphabet to a minimal amount of letters in the destination alphabet (here $\{0, 1\}$). Letters with low probability result in longer bit strings and vice versa. Unlike that, Tunstall Codes encode a variable number of letters of the source alphabet into a fixed amount of bits. The main advantage of this scheme is the better handling of errors as they do not propagate through the rest of the decoding [13].

The algorithm itself is easy to implement and use. Before the algorithm starts, a list of tuples $\{(a, P(a))|a \in \mathfrak{A}\}$ is propagated. This list of tuples can be extended to the desired length by a simple algorithm. As long as the list is shorter than desired, the element with the highest probability is removed. Each letter $a \in \mathfrak{A}$ gets appended to this element while updating the probabilities. This increases the code list by $|\mathfrak{A}| - 1$ each time. After reaching the desired length each element in the code list gets a simple binary code. The whole algorithm is presented in Algorithm 5. To encode a message, a simple search for the longest match in the table is sufficient. For decoding, a table lookup is sufficient.

Algorithm 5 Tunstall table generation.

Require: s is a list of elements of an alphabet, b is the desired bit width, a is the alphabet with probabilities, l is the length of the alphabet

```
1:  function GENERATETUNSTALLCODE($s$,$b$,$a$,$l$)
2:      $r \leftarrow a$        ▷ List consists of all letter/probability pairs of alphabet $a$.
3:      $k \leftarrow 0$
4:      while $k \leq \frac{2^b - l}{(l-1)}$ do        ▷ Is the desired bitwidth reached? (See [13, p. 70]))
5:          $t \leftarrow$ POPHIGHESTPROBABILITY($r$)        ▷ Pop the string with the highest probability
6:          APPENDWITHPROBABILITY($r$,$t$,$a$)        ▷ Appends $a$ to $t$ while updating probabilities. Then inserts this list into $r$.
7:          $k \leftarrow k + 1$
8:      end while
9:      return r
10: end function
```

4 Comparison

This chapter describes the steps taken to chose the best fitting algorithm to compress the data described in section Section 2.2 for use in the application described in Section 2.1.

The algorithm has to fit different constraints.

1. The experiment depends on energy-efficient usage of the computational hardware. This helps reducing the size and the weight of the required power supply system.
2. To maintain accurate control, the compression latency must be limited to 100 ms. This ensures the controllability of the sensor positioning through the human operator.
3. The memory required by the compression algorithm must not exceed a few kilobytes, depending on the capabilities of the computational hardware.
4. The algorithm should be able to adapt to varying statistical characteristics of the data to be compressed.

4.1 Algorithms

The selection of algorithms has taken place after an intense study of overall performance estimations. The algorithms are selected to give an overview over high performance algorithms that need high processing power like PAQ as well as more modest algorithms like 7z. The selection is motivated by compression algorithms benchmarks [5, 10]. The selected algorithms are described in Table 4.1. As far as possible standardized implementations are used. Most of these add some additional information to the compressed data, such as package format and other meta information. If one of these algorithms proves viable, further evaluation has to be done without the additional overhead. All algorithms are configured to achieve their highest compression rates.

This section described how, why and which algorithms have been chosen for comparison. The following section will describe how the testing process was automated.

4.2 Testing Environment

This section describes the testing environment used to determine the compression ratios of the selected algorithms in various settings. The test suit is realized by a python program. It first creates the data to compress and then applies the chosen compression to this data. The first step operates on a 1.048.576 bytes large file containing interleaved sensor channels as described in Section 2.2. For each channel as many files as possible of the desired block length are created. These block lengths are chosen

Name	Description	Implementation
7z	LZMA (improved LZ77)	7-Zip [64] 9.22 beta
Bz2	BWT	Python Bz2
Huffman	Two pass Huffman with encoded tree (about 68 byte)	shcodec 1.0.1
LZW and Huffman	LZW followed by Huffman encoding	shcodec 1.0.1 and lzw python
LPAQ	Light implementation of PAQ, neural network prediction followed by arithmetic coding	lpaq8
LZW	Improved LZ78 algorithm	Python LZW
PAQ	Prediction of next symbol with various approaches and combination with an artificial neural network followed by arithmetic coding	paq8o
ZPAQ	Prediction of next symbol by various context predictors followed by arithmetic coding	ZPAQ v1.02
Differential Encoding	Encoding the differences with RICE encoding	Manual python implementation

Table 4.1: The algorithms used for comparison of compression performances.

to be $\{2^x \text{ samples}\}$, $x \in [1, 2, .., 12]$, which corresponds to a latency of 168 ms for the longest blocks. The files created in the first step are automatically encoded with the different compression programs/algorithms. For each block size, the compression ratio is averaged over all compressed files. All compression ratios are written to a file in an easy to parse format for further processing. This format is a semicolon separated text file. Each line represents one block size and is organised as follows

```
block size;size uncompressed;size compressed;compression ratio
```

The name of the text file indicates the applied algorithm.

The testing process is illustrated in Figure 4.1. Furthermore the data flow is illustrated in Figure 4.2.

After describing how the results have been determined the following section discusses these results.

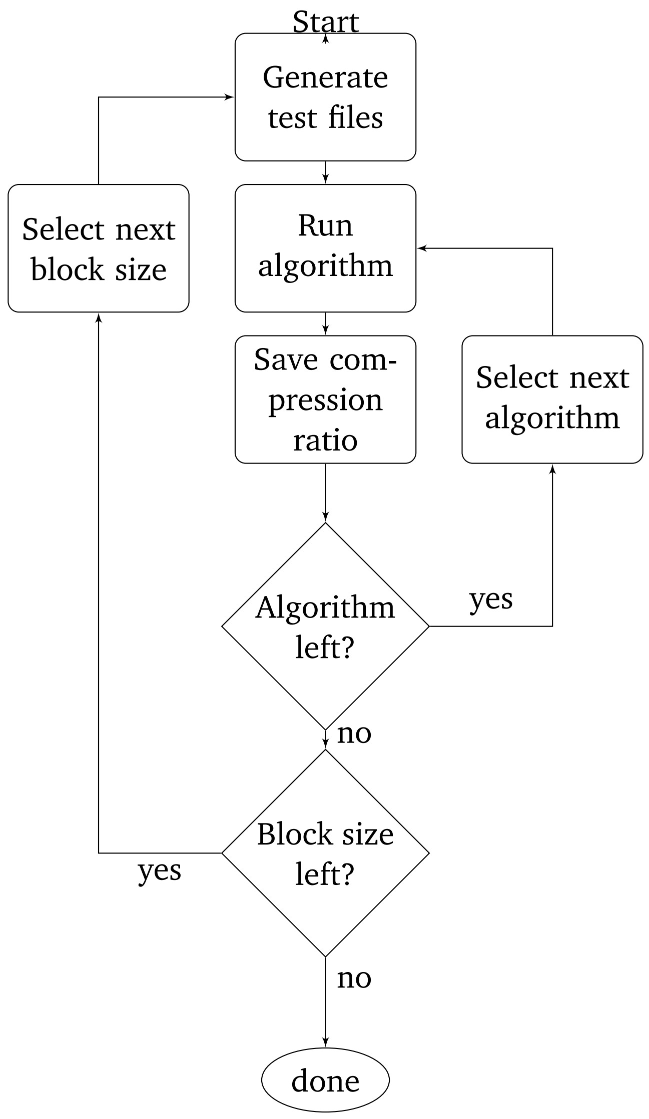


Figure 4.1: This flow chart shows the operations done for testing the various compression algorithms. These operations are implemented as python scripts.

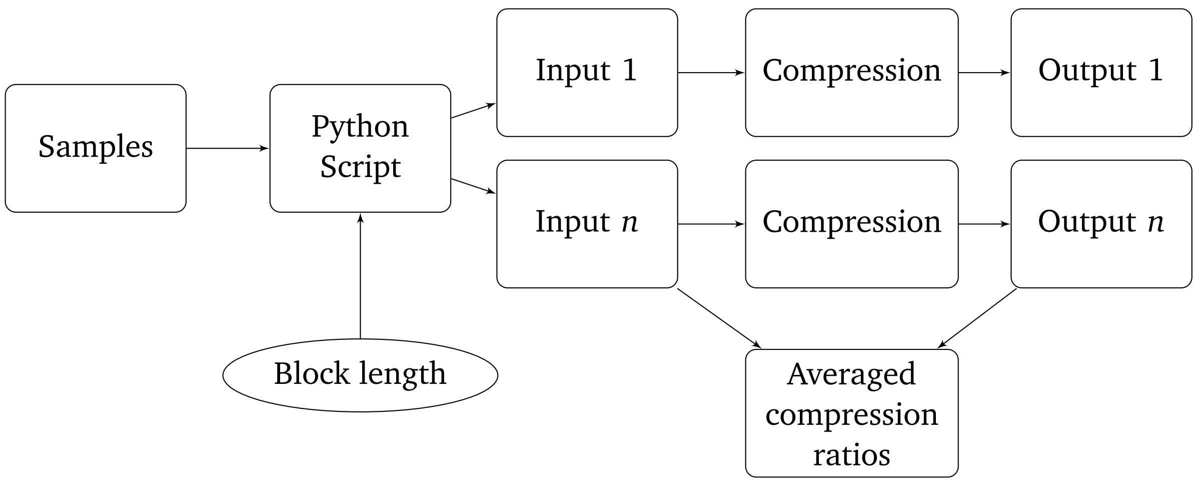


Figure 4.2: Flow of data through the testing process for compression algorithms. The process starts with a file containing measured samples. These samples are split into files of different length corresponding to the block size. Multiple files for each block size get compressed and an average of the compression ratios is saved to file.

4.3 Compression Results

The benchmark results for the compression algorithms are shown in Figure 4.3. All compression algorithms except LZW and Differential result in larger output data than input data for small block lengths and thus achieve a compression ratio greater than one. 7z and ZPAQ require the longest block lengths of at least 256 and 512 samples before actually compressing the data. LZW reduces the data size for blocks with more than 16 samples but does not increase the length for smaller blocks. All compression algorithms achieve a compression of at least 61 % at 4096 samples per block. Furthermore, the compression ratio improves with increasing block sizes for all tested algorithms. The best compression ratio is achieved by PAQ with 38.9 %, LPAQ with 39.8 % and the differential encoding with 39.2 %.

4.4 Evaluation of the Compression Results

After presenting the results, they are discussed in this section and the algorithm to implement on the target hardware is selected. The tested algorithms were able to achieve 38.9% as best result. For this result nearly 2 GiB of memory has been used which makes PAQ, ZPAQ and LPAQ unusable for the implementation on small memory constrained embedded systems. LZW combined with Huffman places fourth with standalone LZW close after. The difference between the version with and without Huffman is so small that standalone LZW would be the way to go without subsequent Huffman. Still LZW requires memory to create the dictionary. As these benchmarks ignore memory constraints, LZW would most likely perform worse on the real hardware.

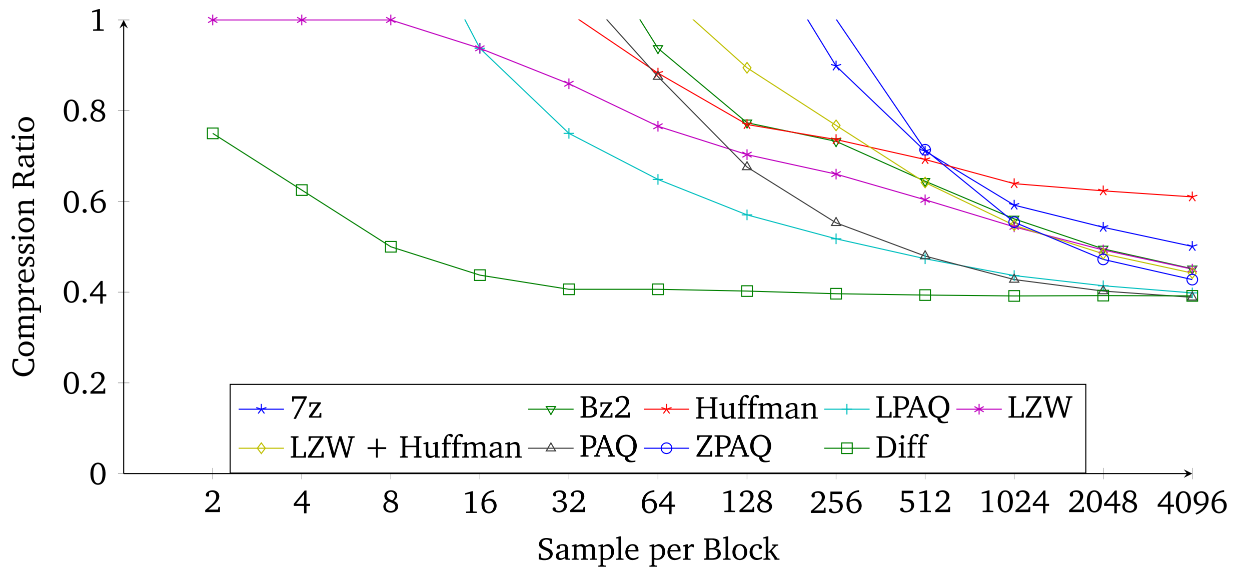


Figure 4.3: Comparison of compression ratios of different compression algorithms on data from sensors used in ape experiments. (See Section 2.2)

Differential encoding on the other hand is resource friendly as only the last sample has to be buffered. Besides the subtraction operation, the involved rice encoder is also not compute intense (See Sections 3.2.4 and 3.3.3). Nevertheless, differential encoding is able to achieve the second highest compression of all algorithms tested here. This differential encoding is an ADPCM encoder with one fixed coefficient ($a_1 = 1$). Due to the superior performance and low operation costs of this algorithm the next section takes a deeper look into ADPCM encoding.

4.5 Testing of ADPCM encoding

Figure 4.4 shows the results of the ADPCM encoding tests. For small block sizes the compression ratio is above one. This behaviour originates from the size of the coefficients that need to be transmitted for each block. For small block sizes below 64 samples one coefficient is the best way to go. Above 64 samples two coefficients perform equally and get even better than one coefficient for larger blocks. Above two coefficients a strange behaviour can be observed. For three coefficients the best block length is 64 samples per block. For higher blocks the compression ratio gets worse. With four coefficients a compression ratio above one and up to 6 can be observed for all tested block length.

In figure Figure 4.5 the variation of the compression ratio can be observed over different block lengths. While working with less than 256 samples per block the size of the different encoded blocks varies greatly. The more samples are taken into account the more predictable the compression rate gets. This trend holds till 1024 samples per block. For larger blocks the compression ratio variance is slightly higher. Thus, 1024

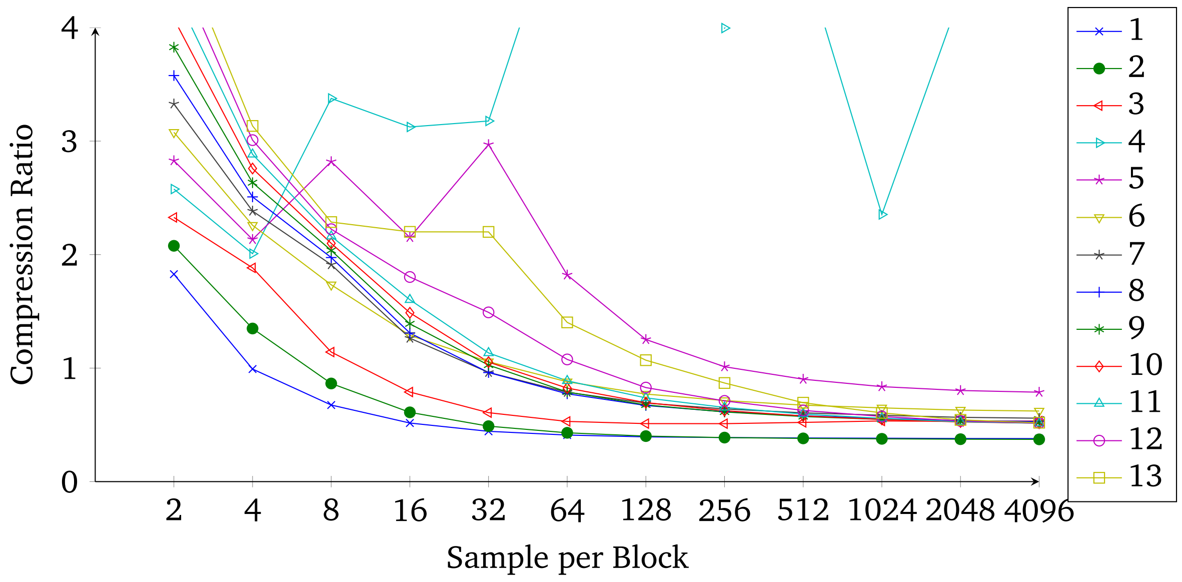


Figure 4.4: Comparison of compression ratios of an ADPCM coding followed by rice code with $M = 16$ with different amounts of coefficients on data from sensors used in ape experiments. (See Section 2.2) Each coefficient takes up 8 bits.

samples per block pose optimal this configuration yields the lowest minimum as well as the lowest variance over all block lengths.

Figure 4.6 shows a comparison of ADPCM encoding with two samples and different golomb parameters. For larger blocks $M = 2$ performs worst by far. $M = \{8, 16, 32\}$ are close together with $M = 16$ being slightly ahead for block sizes above 128.

4.6 Evaluation of ADPCM Test Results

The results from the ADPCM testing presented in the last section will be evaluated here. ADPCM does not require much memory as only the current block has to be kept. The calculation of the autocorrelation coefficients can be done as the samples are received. Fortunately, the best compression can already be achieved with only two prediction coefficients. Thus the coefficient calculations can be limited to two divisions and four multiplications (see Equation (3.8)). The calculation of the prediction value requires two multiplications per sample. The use of rice code allows hardware efficient encoding of each block using only one 'shift' and one 'and' operation. Apart from hardware advantages this algorithm achieves the best compression of all tested algorithms with a compression ratio of 37.7%.

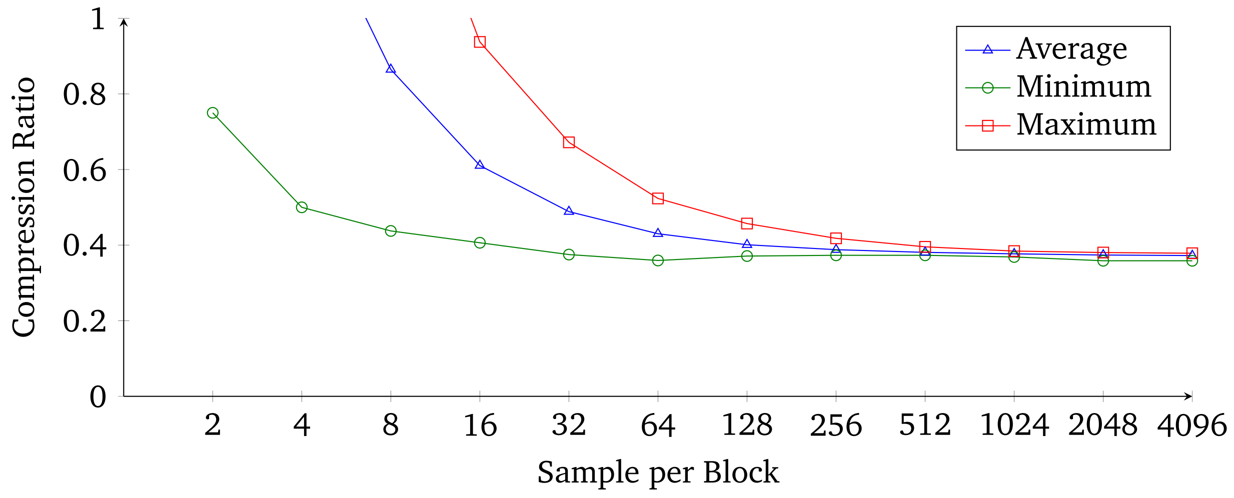


Figure 4.5: Variability of the compression ratios for with two coefficient ADPCM and rice code with $M = 16$. Data samples stem from ape neural experiments (See Section 2.2).

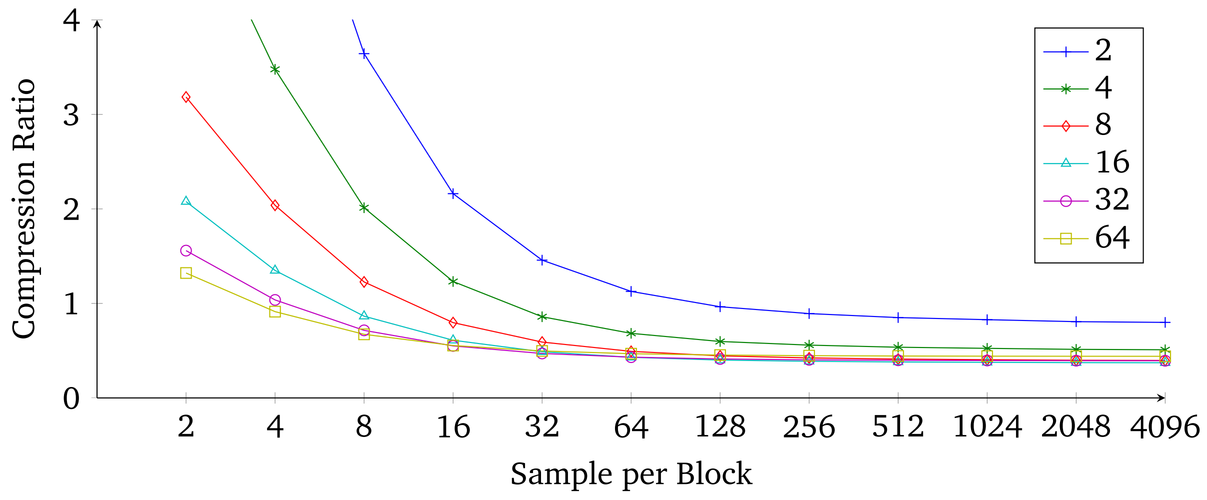


Figure 4.6: Compression ratios for two coefficient ADPCM and different Rice parameters. Data samples stem from ape neural experiments (See Section 2.2).

5 Implementation

After choosing ADPCM with a second order predictor and a subsequent rice-encoder as best suited for the application presented in Chapter 2, this chapter deals with the implementation. Apart from a comparative implementation on a 32 bit Atmel microcontroller, the algorithm is implemented on an Actel IGLOO low power FPGA for improved performance.

5.1 Algorithm Design

The ADPCM algorithm can be divided into three parts.

1. Calculate the autocorrelation coefficients $R_{xx}(0), R_{xx}(1)$ and $R_{xx}(2)$ of the current block of samples.
2. Calculate ADPCM coefficients a_1 and a_2.
3. Apply the ADPCM and the rice encoder to the current block of samples.

5.1.1 Autocorrelation Calculation

The discrete autocorrelation for lag j is defined as

$$R_{xx}(j) = \sum_{n=j}^{N} x_n x_{n-j}, \tag{5.1}$$

with N being the block size [13, p. 545]. For the calculation of the two prediction coefficients $R_{xx}(0)$, $R_{xx}(1)$, $R_{xx}(2)$ are needed. These can be iteratively calculated during sample reception. Therefore, after initializing $r_i(0) = 0 \forall i \in \{0, 1, 2\}$, with each new sample x_n the three calculations

$$r_0(n) = r_0(n-1) + x_n \cdot x_n \tag{5.2}$$
$$r_1(n) = r_1(n-1) + x_n \cdot x_{n-1} \tag{5.3}$$
$$r_2(n) = r_2(n-1) + x_n \cdot x_{n-2} \tag{5.4}$$

have to be executed, where r_i accumulates the value for the corresponding $R_{xx}(i)$. Therefore, the last two samples have to be buffered.

With 1024 samples per block and 16 bit per sample the autocorrelation values possible are limited to 42 bits. This bit width can be reduced by observing that dividing by the block length does not change the compression result. Thus, 32 bit wide autocorrelation coefficients are used.

After the autocorrelation coefficients have been determined, the ADPCM coefficients can be calculated.

5.1.2 Coefficient Calculation

To calculate the two coefficients a_1 and a_2, the equation

$$\begin{pmatrix} R_{xx}(0) & R_{xx}(1) \\ R_{xx}(1) & R_{xx}(0) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} R_{xx}(1) \\ R_{xx}(2) \end{pmatrix} \tag{5.5}$$

has to be solved [13, p. 334]. As shown in [2], it is possible to calculate a_2 without a significant performance loss as

$$a_2 = 1 - a_1 \tag{5.6}$$

while determining a_1 with a simplified form of the exact solution

$$a_1 = \frac{R_{xx}(0) - R_{xx}(2)}{R_{xx}(0) - R_{xx}(1)}. \tag{5.7}$$

Figure 5.1 compares the performance of the estimated coefficients and the exact coefficients for different block sizes. Till about 32 samples per block both methods perform equally well. For larger blocks the exact values perform better with up to 6.4% difference at 4096 samples per block.

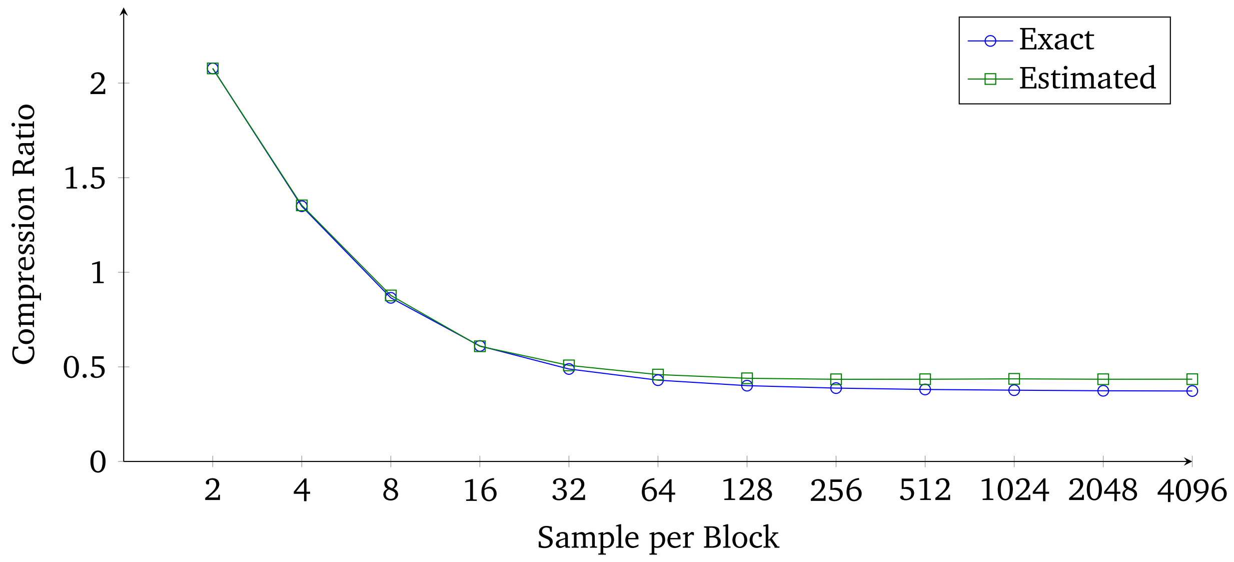


Figure 5.1: Comparison between exact ADPCM coefficents calculated using Equation (5.5) and their estimates calculated with Equations (5.6) and (5.7).

This approach has been discarded because another scheme termed Levinson-Durbin algorithm proves advantageous for transmitting the coefficients. As a byproduct this algorithm computes so called PARCOR coefficients which can be represented with a smaller number of bits then the predictor coefficients itself. As a_1 and a_2 can be retrieved from p_0 and p_1 at the receiver, only the PARCOR coefficients will be transmitted with each encoded block. Thus,

$$\begin{aligned}
p_0 &= \frac{R_{xx}(1)}{R_{xx}(0)} && (5.8)\\
p_1 &= \frac{R_{xx}(0)\cdot R_{xx}(2) - R_{xx}(1)\cdot R_{xx}(1)}{R_{xx}(0)\cdot R_{xx}(0) - R_{xx}(1)\cdot R_{xx}(1)} && (5.9)\\
a_1 &= p_0 - p_1 \cdot p_0 && (5.10)\\
a_2 &= p_1 && (5.11)
\end{aligned}$$

has to be performed [9]. These formulas can be derived from Equation (5.5) by inserting Equations (5.10) and (5.11).

$$\begin{aligned}
\begin{pmatrix} R_{xx}(1) \\ R_{xx}(2) \end{pmatrix} &= \begin{pmatrix} R_{xx}(0) & R_{xx}(1) \\ R_{xx}(1) & R_{xx}(0) \end{pmatrix} \begin{pmatrix} p_0 - p_1 \cdot p_0 \\ p_1 \end{pmatrix} && (5.12)\\
&= \begin{pmatrix} R_{xx}(0)p_0(1-p_1) + R_{xx}(1)p_1 \\ R_{xx}(1)p_0(1-p_1) + R_{xx}(0)p_1 \end{pmatrix} && (5.13)\\
&= \begin{pmatrix} R_{xx}(1)(1-p_1) + R_{xx}(1)p_1 \\ l_2 \end{pmatrix} && (5.14)\\
&= \begin{pmatrix} R_{xx}(1) \\ l_2 \end{pmatrix}, && (5.15)\\
& && (5.16)
\end{aligned}$$

where l_2 is

$$\begin{aligned}
l_2 &= \frac{R_{xx}(1)^2}{R_{xx}(0)} && (5.17)\\
&\quad - \frac{R_{xx}(0)^3 R_{xx}(2) - R_{xx}(0)^2 R_{xx}(1)^2 + R_{xx}(0) R_{xx}(1)^2 R_{xx}(2) - R_{xx}(1)^4}{R_{xx}(0)(R_{xx}(0)^2 - R_{xx}(1)^2)} \\
&= \frac{R_{xx}(0)^2 R_{xx}(2)}{R_{xx}(0)^2 - R_{xx}(1)^2} - \frac{R_{xx}(1)^2 R_{xx}(2)}{R_{xx}(0)^2 - R_{xx}(1)^2} - \frac{R_{xx}(0) R_{xx}(1)^2}{R_{xx}(0)^2 - R_{xx}(1)^2} && (5.18)\\
&\quad + \frac{R_{xx}(1)^4}{R_{xx}(0)(R_{xx}(0)^2 - R_{xx}(1)^2)} + \frac{R_{xx}(1)^2}{R_{xx}(0)} \\
&= \frac{(R_{xx}(0)^2 - R_{xx}(1)^2)R_{xx}(2)}{R_{xx}(0)^2 - R_{xx}(1)^2} - \frac{R_{xx}(0)^2 R_{xx}(1)^2 + R_{xx}(1)^4}{R_{xx}(0)(R_{xx}(0)^2 R_{xx}(1)^2)} + \frac{R_{xx}(1)^2}{R_{xx}(0)} && (5.19)\\
&= R_{xx}(2) - \frac{R_{xx}(1)^2 (R_{xx}(0)^2 + R_{xx}(1)^2}{R_{xx}(0)(R_{xx}(0)^2 - R_{xx}(1)^2)} + \frac{R_{xx}(1)^2}{R_{xx}(0)} && (5.20)\\
&= R_{xx}(2). && (5.21)
\end{aligned}$$

5.1.3 Output Encoding

After the coefficients are available, the actual encoding step is performed. First, the first two samples of each block, that are not going to be encoded, are sent followed by the two PARCOR coefficients. For each subsequent sample the predicted value

$$x_p = a_1 \cdot x_{n-1} + a_2 \cdot x_{n-2} \tag{5.22}$$

has to be calculated. The value to be transmitted is calculated by

$$x_t = x_n - x_p. \tag{5.23}$$

Before transmission this value is rice encoded. The calculations required are

$$r = x_t \& 15 \tag{5.24}$$
$$q = x_t >> 4. \tag{5.25}$$

The quotient q and the reminder r can be calculated efficiently in hardware using 'right shift' and 'and' operations. The rice code is retrieved by concatenating q ones $((1 << q) - 1)$, a zero and the bit representation of r.

As these calculations can be done much faster than the serial output transmits them, a queue buffers the output bitstream.

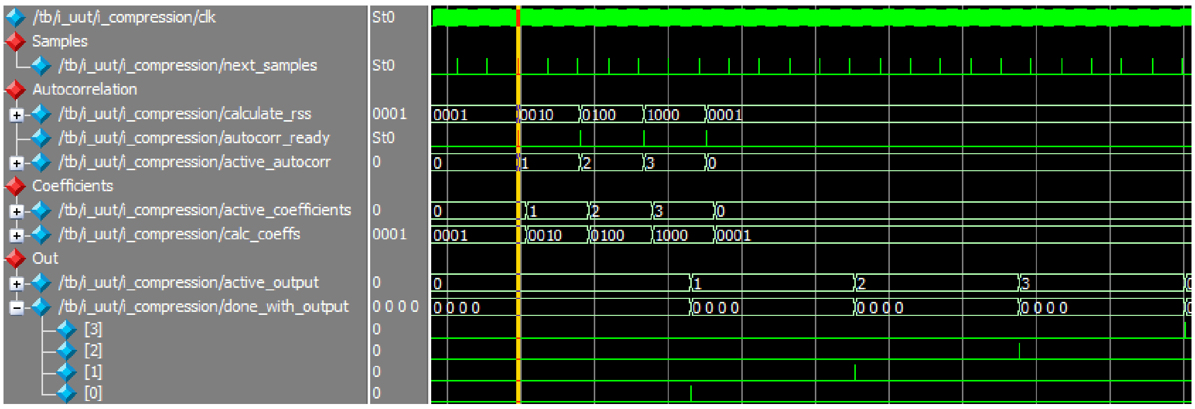
5.2 FPGA Implementation

The FPGA implementation is designed to be deployed on an Actel IGLOO ultra-low-power FPGA. This FPGA offers 144 kbit of RAM divided in 32 4608 bit blocks. The first iteration of the design consisted of a single unit capable of doing all required calculations for a single channel. This solution proved inefficient as it was not possible to fit more than one channel onto the FPGA. Simulations proved the output generation to be the timing bottleneck while the calculation of the autocorrelation and the predictor coefficients required only a small fraction of the computational time.

The second iteration is designed to address this fact by performing the calculations sequentially for each channel. The whole calculation is controlled by a central unit. This unit has different tasks. First of all, each channel has a module for saving the channels samples in BRAM. Each channel must buffer its samples until it accessed the sequentially shared calculation resources. New incoming samples would overwrite the block currently in processing. Thus each sample buffer is dimensioned to hold two subsequent blocks at ones (double buffering). Secondly, the central unit assigns control over the other calculation units to the channels. These comprise one unit for calculating the autocorrelation coefficient, one unit for calculating the ADPCM coefficients and one unit to output the data to the serial output buffer. These resources are assigned to the next channel as soon as the channel currently owning the resource has finished operation. Furthermore, one module is instantiated that encodes the difference between predicted and original value with a RICE code and organises the output in the right order.

The design operates on three clock signals. The first clock is the main operation clock and is used for all calculations. The second clock is used for signal sampling. The last clock is used for the output. FIFOs are used to cross the clock boundaries.

An example operation with four channels is illustrated in Figure 5.2.



/tb/i_uut/i_compression/clk St0
Samples
/tb/i_uut/i_compression/next_samples St0
Autocorrelation
/tb/i_uut/i_compression/calculate_rss 0001
/tb/i_uut/i_compression/autocorr_ready St0
/tb/i_uut/i_compression/active_autocorr 0
Coefficients
/tb/i_uut/i_compression/active_coefficients 0
/tb/i_uut/i_compression/calc_coeffs 0001
Out
/tb/i_uut/i_compression/active_output 0
/tb/i_uut/i_compression/done_with_output 0 0 0 0
[3] 0
[2] 0
[1] 0
[0] 0

Figure 5.2: Waveforms from a simulation of the FPGA's operation with four channels. Control signals are shown. The yellow bar marks reception of the last sample of a block. active_autocorr denotes the channel that currently controls the autocorrelation module. active_coefficients denotes the channel that currently uses the coefficient calculation module. active_output denotes the channel that currently outputs data to the serial connection.

The following module

```
module compression
 #(parameter BIT_PER_SAMPLE   =     16,          ///< sample bit width        [<= 16]
   parameter CHANNELS         =      3,          ///< number of channels      [<=  4]
   parameter SAMPLE_PER_BLOCK = 1024,            ///< sample per block, power of two
   parameter AUTOCORR_LENGTH  = 32,              ///< bit width of the autocorrelation busses
   parameter ACCURACY = 7)                       ///< digits after the point for fixed point numbers
  (input      clk,                               ///< computation clock
   input      rstN,                              ///< computation reset (synchronous, low active)
   input      [BIT_PER_SAMPLE*CHANNELS-1:0] samples, ///< uncompressed input stream
   input      next_samples,                      ///< input valid pulse
   output  [7:0] byte,                           ///< compressed output stream
   output  next_byte);                           ///< output valid pulse
```

provides the interface to the data compression. It takes the clock and distributes it to all other modules. Furthermore, the samples for all channels arrive here. This module contains one `compress_single_channel` module per channel and one of the `calc_adpcm_coefficients` and `autocorrelation` modules. Internally the resources are distributed to the `compress_single_channel` modules by a resource specific counter. This counter counts up each time the attached resource finishes operation. This results in a round-robin resource scheduling.

The following module

```
module compress_single_channel
  (input                                clk,            ///< computation clock
   input                                rstN,           ///< computation reset (synchronous, low active)
```

```
    input [BIT_PER_SAMPLE-1:0]            samples,          ///< uncompressed input stream
    input                                 next_samples,     ///< input valid pulse
    // Coeffs
    input                                 calculate_coeffs, ///< this module may calculate coefficients
    input                                 coeffs_valid,     ///< coefficients are calculated and lay at
                                                            ///< the appropriate ports
    input signed[2:-ACCURACY]             i_a0,             ///< filter coefficient 0
    input signed[2:-ACCURACY]             i_a1,             ///< filter coefficient 1
    input signed[0:-ACCURACY]             i_p0,             ///< PARCOR coefficient 0
    input signed[0:-ACCURACY]             i_p1,             ///< PARCOR coefficient 1
    output                                coeffs_start,     ///< coefficient module may start calculation
    output signed[(2*BIT_PER_SAMPLE)-1:0] r0_o,             ///< R_xx(0) for coeffs
    output signed[(2*BIT_PER_SAMPLE)-1:0] r1_o,             ///< R_xx(1) for coeffs
    output signed[(2*BIT_PER_SAMPLE)-1:0] r2_o,             ///< R_xx(2) for coeffs
    // Autocorrelation
    input                                 calculate_rss,    ///< this module may calculate autocorrelation
    input                                 rss_valid,        ///< inputs r0,r1,r2 are valid
    input signed[(2*BIT_PER_SAMPLE)-1:0]  r0,               ///< R_xx(0) from autocorr
    input signed[(2*BIT_PER_SAMPLE)-1:0]  r1,               ///< R_xx(1) from autocorr
    input signed[(2*BIT_PER_SAMPLE)-1:0]  r2,               ///< R_xx(2) from autocorr
    output                                sample_valid,     ///< sample_o is valid
    output signed[BIT_PER_SAMPLE-1:0]     sample_o,         ///< current sample for autocorrelation module
    // Output
    input                                 allow_output,     ///< this module may output to uart
    input                         increase_read_address,    ///< output encoding module needs next address
    input                                 done_with_output, ///< the output module has finished operation
    output                                start_output,     ///< if set the output module starts operation
    output signed [BIT_PER_SAMPLE-1:0]    sample_output,    ///< current read memory sample
    output signed [2:-ACCURACY]           o_a0,             ///< ADPC coefficient 0
    output signed [2:-ACCURACY]           o_a1,             ///< ADPC coefficient 1
    output signed [0:-ACCURACY]           o_p0,             ///< PARCOR coefficient 0
    output signed [0:-ACCURACY]           o_p1              ///< PARCOR coefficient 1
);
```

is instantiated once for each channel. It administrates all data belonging to a channel. This includes buffering the block of samples as well as the corresponding autocorrelation values predictor coefficients. Furthermore, it manages which calculations have to be done for this channel and executes them as soon as it gets the corresponding resource.

The following module

```
module organiseoutput
 (
  input                               clk,              ///< computation clock
  input                               rstN,             ///< computation reset (synchronous, low active)
  input                               start,            ///< all inputs are valid, start the output
  input                               allow_output,     ///< channel may output
  input signed [BIT_PER_SAMPLE-1:0] sample,           ///< sample to encode
  input signed [2:-ACCURACY]          a0,               ///< ADPCM coefficient 1
  input signed [2:-ACCURACY]          a1,               ///< ADPCM coefficient 2
  input signed [0:-ACCURACY]          p0,               ///< PARCOR coefficient 1
  input signed [0:-ACCURACY]          p1,               ///< PARCOR coefficient 2
  output reg                          increase_read_address, ///< next sample at input sample
  output reg    [7:0]                 byte,             ///< output byte
  output reg                          next_byte,        ///< output byte valid
  output reg                          done_with_output  ///< outputted all data
  );
```

handles the output for each module. This includes outputting the data in the right order and applies the rice encoder to the prediction deviations.

The following module

```
module calc_adpcm_coefficients
(input                              clk,   ///< computation clock
 input                              rstN,  ///< computation reset (synchronous, low active)
 input                              start, ///< start calculation
 input  signed[AUTOCORR_LENGTH-1:0] r0,    ///< r_xx(0)
 input  signed[AUTOCORR_LENGTH-1:0] r1,    ///< r_xx(1)
 input  signed[AUTOCORR_LENGTH-1:0] r2,    ///< r_xx(2)
 output reg                         done,  /// done with calculation
 output signed[2:-ACCURACY]         a0,    /// filter coefficient 0
 output signed[2:-ACCURACY]         a1,    /// filter coefficient 1
 output signed[0:-ACCURACY]         p0,    /// PARCOR coefficient 0
 output signed[0:-ACCURACY]         p1);   /// PARCOR coefficient 1
```

calculates the predictor and PARCOR coefficients. This module contains one divider and one multiplier that are used serialised.

The following module

```
module autocorrelation
(input                               clk,          ///< computation clock
 input                               rstN,         ///< computation reset (synchronous, low active)
 input signed [BIT_PER_SAMPLE-1:0]   sample,       ///< input sample
 input                               next_samples, ///< input sample valid
 output signed [AUTOCORR_LENGTH-1:0] r0,           ///< R_xx(0) accumulated
 output signed [AUTOCORR_LENGTH-1:0] r1,           ///< R_xx(1) accumulated
 output signed [AUTOCORR_LENGTH-1:0] r2,           ///< R_xx(2) accumulated
 output reg ready);                                ///< r0, r1 and r2 valid
```

calculates the autocorrelation coefficients. Each time `next_sample` is valid, the current data attached to sample is multiplied by the corresponding delayed sample and added to accumulator registers. This requires this module to save the last two samples used. Furthermore, the current position in the block is observed.

The following module

```
module rice_encode
(input clk,                   ///< computation clock
 input rstN,                  ///< computation reset (synchronous, low active)
 input enabled,               ///< Rice module is enabled and used
 input in_v,                  ///< Input in valid
 input[BIT_PER_SAMPLE-1:0]in, ///< current sample
 input clear,                 ///< Don't wait for next sample but fill output with zeros
 output reg[7:0] out,         ///< Sample to send via UART
 output reg out_v,            ///< Output out valid
 output next);                ///< this module is able to process the next sample
```

calculates the rice code of the data attached to the input `in_v`. If the size of the output is smaller than 8 bit, the module waits till the next sample arrives and concatenates both encoded samples. To remove this behavior and clear the output register after a sample, `clear` can be set to one. This module can be disabled by setting `enabled` to zero.

5.3 Microcontroller Implementation

Contrary to the FPGA implementation the microcontroller implementation is strictly sequential.

This implementation is written in C for the Atmel AT32UC3A Microcontroller. No library functions are needed. This design is build around a C structure that holds all data needed per channel.

```
typedef struct {
        short sample_buffer[2*SAMPLE_IN_BLOCK];
        int readPosition;
        int writePosition;
        long autocorr_accu[3];
        long autocorr[3];
        short delay1;
        short delay2;
        char p0;
        char p1;
        short a0;
        short a1;
        } s_encoder;
```

The sample buffer is again twice as big as the block size as new incoming samples would overwrite samples currently processed. This structure can be manipulated by three functions.

```
void encoder_init(s_encoder *e);
```

This function initialises a `s_encoder` structure with meaningful values and processed called for all newly generated `s_encoder` structures.

```
char encoder_addSample(s_encoder *e, short sample);
```

The samples generated per channel can be fed into the structure by the `encoder_addSample` function. This function calculates the autocorrelation coefficients on-the-fly. After a full block has been received, the ADPCM coefficients are calculated automatically.

```
int encoder_sendSamples(s_encoder *e,
                        uint8_t* fifo,
                        uint32_t fifo_head,
                        uint16_t fifo_length);
```

The last function outputs the data of structure e into the fifo and returns the new fifo head.

5.4 Decoder

To decode the encoded bit stream, a python script is used. This script can operate on arbitrary inputs like files or stdio. It takes the input stream and decodes it into the original sequence. First the two unencoded samples are retrieved by simply getting 16 bit from the sample.

```
t = 0
for i in range(16):
        t <<= 1
    t |= b.getBit() # Retrieve the next bit from the bit stream
if t & (1<<15):     # Negative Integer?
   t = signed(t)
```

The sample is returned to its original sign in lines 5 and 6.

The next 16 bit contain the PARCOR coefficients. These are retrieved in the same way as the first two samples. With these two samples the ADPCM coefficients are calculated.

```
a = []
a.append(p0-adpcm.fixedPointMultiply(p1,p0,7))
a.append(p1)
```

For the rest of the samples in the block the prediction value is calculated and the difference between the predicted and the original value is retrieved by decoding the rice encoded bit stream.

```
nr, b = adpcmbenchmark.golombDecodeNext(b, 16)
pred = ((last * a[0] + next_to_last * a[1]) >> 7)
```

where `last` contains the value of the last sample and `next_to_last` contains the value of the sample before. The original value is determined by addition of `nr` and `pred`.

After all samples have been read the current byte is discarded as the output generated by the implementations is aligned to byte boundaries.

The samples are saved to an array that can be used in the desired way.

6 Evaluation

After selecting and implementing the most suitable algorithm, namely ADPCM and Rice, in the last chapters this chapter deals with the evaluation of the implementation. Section 6.1 describes the testing environment. Section 6.2 finally presents and discusses the results.

6.1 Setup

This section describes the test setup for both the FPGA as well as the MCU implementation. Instead of using a real ADC with a sensor to generate samples, a test file is read from an internal memory of the devices. The test file from Chapter 4 was reused for this purpose. This way, the experiments are repeatable. The FPGA implementation is entirely evaluated in ModelSim. The MCU implementation is connected to an oscilloscope for accurate timing measurement.

The most important aspect to be evaluated is the time needed after a full block of 1024 samples has arrived till all channels have written their output into the output FIFO. Based on this time the maximum possible amount of channels the device can handle is calculated.

The FPGA implementation is evaluated by measuring the time needed for the first channel from the time the block has been received completely till the whole block has been encoded and outputted into the FIFO. Because the autocorrelation of the first channel is calculated while the samples come in, it needs less time after the block has been completed. Thus a second channel is measured, too.

The time needed for all channels to complete must be smaller than the time needed to sample the complete next block. The amount of channels n that can be encoded with the FPGA must fulfill

$$t_1 + (n-1) \cdot t_n \leq \tfrac{1024}{24414}\mathrm{s} \tag{6.1}$$

where t_1 is the time the first channel needs for coefficient calculation and output to FIFO and t_n is the time needed for the output of the following channels as autocorrelation and the coefficients are already calculated after the first channel has freed these resources. To find the maximal number of channels, the greatest n fulfilling the inequality has to be determined. Hence, the amount of channels that can be encoded with the FPGA is given by solving Equation (6.1) for n.

$$n \leq \frac{\tfrac{1024}{24414}\mathrm{s} - t_1}{t_n} + 1, \tag{6.2}$$

The MCU implementation has two points of interest. First of all, the time between two samples has to be large enough to calculate the next iteration of the autocorrelation coefficients for all channels (Equations (5.2) to (5.4)). Secondly, the time after the block has been completed has to be large enough for all channels to output their data into the FIFO. Furthermore, the time needed for calculating the autocorrelation of the following block during the calculation of the predictor coefficients and encoding of the last block needs observation.

The inequality

$$n_c \cdot t_a \leq \tfrac{1}{24414}\mathrm{s} \tag{6.3}$$

$$n_c \leq \frac{\tfrac{1}{24414}\mathrm{s}}{t_a}, \tag{6.4}$$

where t_a is the time needed to calculate the autocorrelation coefficients of a single channel, poses an upper limit for the number of channels the MCU is able to process. The highest number of channels possible for the pure coefficient calculation and output generation is given by

$$n_s \cdot (t_n + 1024 \cdot t_a) \leq \tfrac{1024}{24414}\mathrm{s} \tag{6.5}$$

$$n_s \leq \frac{\tfrac{1024}{24414}\mathrm{s}}{(t_n + 1024 \cdot t_a)} \tag{6.6}$$

due to the sequential MCU processing the time available for the coefficient calculation has to be reduced by the time needed for all autocorrelation calculations in the block. The MCU implementation will work correctly if the amount of channels is chosen to fulfill both Equations (6.4) and (6.6). Thus it holds

$$n \leq \min\left(\lfloor n_c \rfloor, \lfloor n_s \rfloor\right). \tag{6.7}$$

For a real application only a small fraction of the theoretically possible amount of channels can be used for the MCU implementation. Other calculations already take place in the microcontroller and the theoretical maximum would occupy 100% of the calculation capacities.

6.2 Results

For the FPGA the following times have been measured:

$$t_1 = 234.668\,\mu\mathrm{s} \tag{6.8}$$

$$t_n = 224.424\,\mu\mathrm{s} \tag{6.9}$$

$$t_a = 82.912\,\mathrm{ns} \tag{6.10}$$

These times have been determined by post layout simulation in ModelSim ACTEL 6.6d Revision: 2010.11. The synthesis is done by Synplify Pro E-2010.09A−1 and Designer 9.1.5.1 Release v9.1 SP5. The times are averages over ten blocks/samples. The main clock has been set to the largest possible value of 45.1 MHz limited by the design.

For the MCU the following times have been measured:

$$t_a = 4.23\,\mu\text{s} \tag{6.11}$$
$$t_n = 7.90\,\text{ms}. \tag{6.12}$$

These times have been determined by using an oscilloscope. The code was compiled with O2 and the MCU runs at 66MHz.

An interesting aspect is the time used for calculation relative to the time required to sample a whole block as the time not spend on calculations can be used to save energy. This relation is calculated as

$$t_{\text{calc FPGA}} = 1024 \cdot t_a + t_1 + (n-1) \cdot t_n \tag{6.13}$$
$$r_{\text{calc FPGA}} = t_{\text{calc FPGA}} \cdot \frac{24414}{1024}\text{s}, \tag{6.14}$$

with n being the number of channels. The same formula for the MCU is given by

$$t_{\text{calc MCU}} = n \cdot (1024 \cdot t_a + t_n) \tag{6.15}$$
$$r_{\text{calc MCU}} = t_{\text{calc MCU}} \cdot \frac{24414}{1024}\text{s}. \tag{6.16}$$

Figure 6.1 shows this relation. As can be seen the FPGA is significantly faster in calculating the output. The FPGA uses at most 2.9 % of the time for calculating four channels. The MCU implementation on the other hand uses already 29.16 % of the time for calculating one channel. The percentage increases to 116.69 % at four channels.

Unfortunately the FPGA implementation is limited to three channels due to a lack of memory. With one channel already 16 of 32 available block RAMs are used. Each channel requires eight more BRAMs. The implementation requires 40 BRAMs at four channels. Thus a four channel implementation is not supported by the current design. The BRAM utilisation is displayed in Figure 6.2.

For all other resources the FPGA stays below the limit. 91 % of the 24576 available core cells are used at four channels. Figure 6.3 presents the core cell utilisation.

The theoretical maximum of channels for the FPGA can be calculated by Equation (6.2),

$$n \leq \frac{\frac{1024}{24414}\text{s} - 2.346\,68 \times 10^{-4}\,\text{s}}{2.244\,24 \times 10^{-4}\,\text{s}} + 1 \tag{6.17}$$
$$n \leq 186.846787. \tag{6.18}$$

Therefore the FPGA is able to handle 186 channels. Even 187 channels are possible because no required data is overwritten.

Inserting Equations (6.11) and (6.12) in Equation (6.4) yields

$$n_c \leq \frac{\frac{1}{24414}\text{s}}{4.23 \times 10^{-6}\,\text{s}} \tag{6.19}$$
$$n_c \leq 9.688324, \tag{6.20}$$

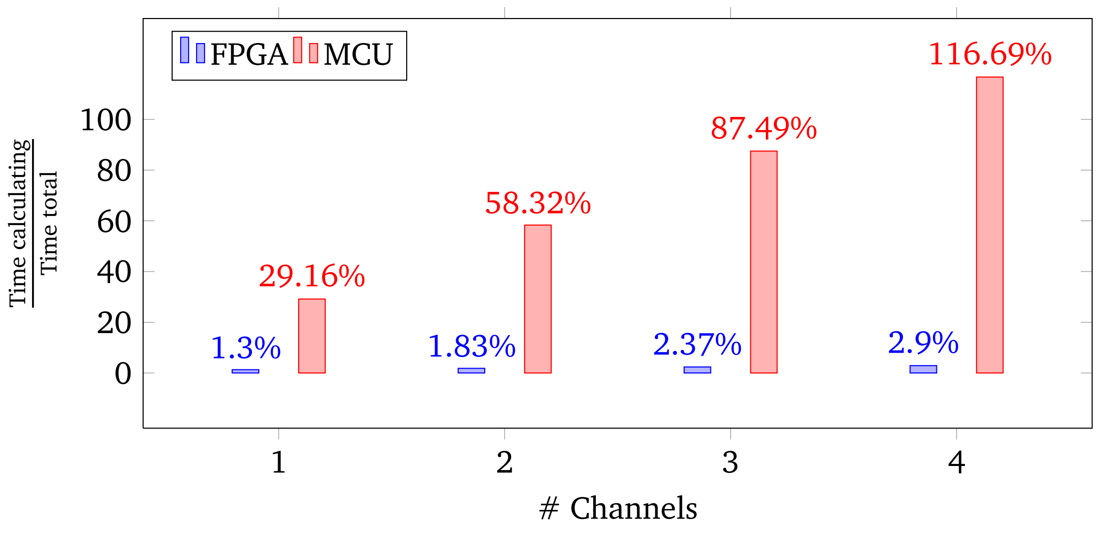


Figure 6.1: Runtime comparison of the FPGA and MCU implementations. For each amount of channels the time needed to complete all calculations for one block is plotted in relation to the total time available for one block. A lower percentage can be advantageous for the power consumption of the device.

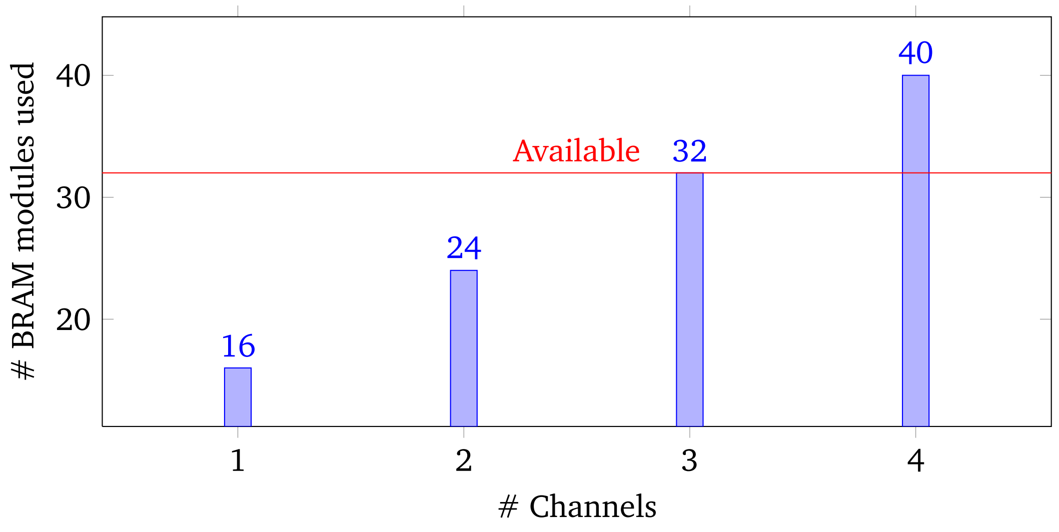


Figure 6.2: Block RAM usage of the FPGA implementation for different amounts of channels. The red line indicates the available BRAM modules. As can be seen at four channels the FPGA can not provide enough BRAM modules to fulfill the requirements.

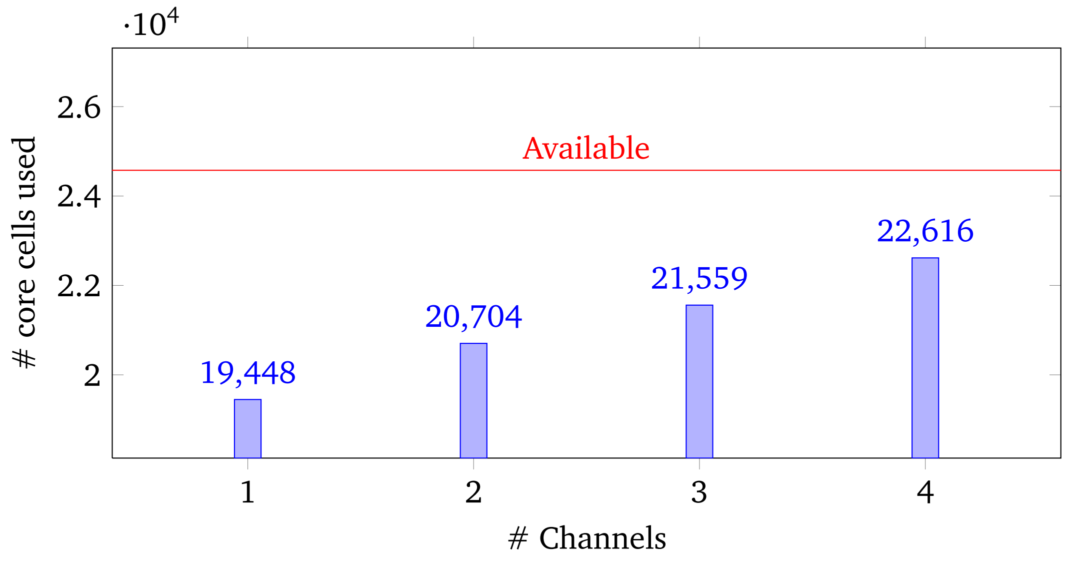


Figure 6.3: Core Cell utilisation of the FPGA implementation for different channel configurations. Mapping done by Synplify Pro E-2010.09A−1.

thus the MCU can calculate the autocorrelation coefficients for a maximum of nine channels. Furthermore, inserting Equations (6.8) and (6.9) in Equation (6.2) yields

$$n_s \leq \frac{\frac{1024}{24414}\,\mathrm{s}}{0.0079\,\mathrm{s} + 1024 \cdot 4.23 \times 10^{-6}\,\mathrm{s}} \tag{6.21}$$

$$n_s \leq 3.429. \tag{6.22}$$

Hence the maximum number of channels for which the coefficients can be calculated and the output can be generated is three.

Following Equation (6.7), it results

$$n \leq \min\left(\lfloor 9.688324 \rfloor, \lfloor 3.429 \rfloor\right) \tag{6.23}$$

$$n \leq \min\left(9, 3\right). \tag{6.24}$$

Therefore the MCU is able to process a maximum of three channels at the same time.

These results indicate that the MCU is too weak to handle four channels. Furthermore, the MCU needs to handle other tasks in the real application as well leaving only a fraction of processing power to the output encoding. The FPGA solution on the other hand can handle all realistic scenarios. Both implementations are further limited by memory requirements that has not been considered yet. Taking memory into account, the FPGA is able to handle three channels at 1024 samples per block. The MCU is still able to handle three channels. The memory problem can be relieved by using less samples per block at the cost of worse compression performance. This tradeoff has to be evaluated for each application.

7 Conclusions and open issues

This work proposes the use of an ADPCM code with two coefficients followed by a rice encoding step to encode sensor data representing neuronal activities of apes. This application requires the encoding algorithm to meet various restrictions. The proposed algorithm is able to achieve a compression ratio of up to 37.1 % with low computational complexity. While the algorithm can be implemented on the already used MCU, the use of an additional FPGA proves advantageous in terms of calculation speed. This additional computing power could be used for more channels in later applications. Furthermore this enables the device to save power.

The compression could be further improved by encoding the resulting already encoded bit stream with a source encoding step like LZW to get rid of repetitions. This requires further research regarding block length, optimal prediction order and rice parameters as well as a further look into suitable (fast) source encoding algorithms.

The implementations have to be tested for the real application. Especially the power consumption of the MCU compared to a combined MCU and FPGA solution has to be evaluated.

The FPGA implementation needs some improvements to be able to handle four channels. Currently the implementation needs too many block RAM module, as each channels needs twice the block size to buffer the input. This buffer is needed to store the incoming samples while the output for the last block is calculated. A possible solution is to handle incoming samples during calculation in a central buffer and forward these samples as soon as the calculation is complete. Another solution is the reduction of the block length which results in a worse compression ratio.

Bibliography

[1] S. Arming, *Data compression in hardware — The Burrows-Wheeler approach*.

[2] C. Boonyakitmaitree, K. Nandhasri, and J. Ngarmnil, “A low computational predictor coefficient algorithm for adpcm implementation of portable recording devices”, in *Circuits and Systems, 2004. MWSCAS ’04. The 2004 47th Midwest Symposium on*, vol. 3, 2004, iii –18790 vol.3–.

[3] M. Burrows and D. J. Wheeler, “A Block-Sorting Lossless Data Compression Algorithm”, Digital Equipment Corporation, Tech. Rep., 1994, pp. 12–4.

[4] M. E. P. Capo-Chichi, J.-M. Friedt, and H. Guyennet, “Using data compression for delay constrained applications in wireless sensor networks”, in *Proceedings of the 2010 Fourth International Conference on Sensor Technologies and Applications*, ser. SENSORCOMM ’10, Washington, DC, USA: IEEE Computer Society, 2010, pp. 101–107, ISBN: 978-0-7695-4096-2.

[5] CompressionRatings.com. (). Audio1, [Online]. Available: `http://compressionratings.com/aud1.html` (visited on 06/15/2012).

[6] N. Faller, “An Adaptive System for Data Compression”, *Record of the 7th Osilomar Conf. on Circuits, Systems and Computers*, pp. 593–597, Nov. 1973.

[7] R. Gallager, “Variations on a theme by Huffman”, *IEEE Transactions on Information Theory*, vol. 24(6), pp. 668–674, 1978. [Online]. Available: `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=01055959`.

[8] S. W. Golomb, “Run-length encodings”, *IEEE Transactions on Information Theory*, vol. 12, pp. 399–401, 1966. [Online]. Available: `http://urchin.earth.li/~twic/Golombs_Original_Paper/`.

[9] ivanov, arokem, agramfort, and miketrumpis. (), [Online]. Available: `https://github.com/nipy/nitime/blob/master/nitime/algorithms/autoregressive.py` (visited on 07/26/2012).

[10] M. Mahoney. (). Large text compression benchmark, [Online]. Available: `http://mattmahoney.net/dc/text.html` (visited on 06/15/2012).

[11] oxforddictionaries.com. (). What is the frequency of the letters of the alphabet in english?, [Online]. Available: `http://oxforddictionaries.com/words/what-is-the-frequency-of-the-letters-of-the-alphabet-in-english` (visited on 05/05/2012).

[12] L. Rafflenbeul, T. Schönbach, and R. Werthschützky, *Drahtloses sensor-aktor-system zur erfassung neuronaler aktivität*. [Online]. Available: `http://www.embedded-world.eu/fileadmin/user_upload/pdf/egm2011/Session_1/05_Rafflenbeul_TU_Darmstadt.pdf`.

[13] K. Sayood, *Introduction to Data Compression, Third Edition (Morgan Kaufmann Series in Multimedia Information and Systems)*, 3rd ed. Morgan Kaufmann, Dec. 2005, ISBN: 012620862X. [Online]. Available: `http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/012620862X`.

[14] C. E. Shannon, “A mathematical theory of communication”, *Bell system technical journal*, vol. 27, 1948.

[15] Toytoy. (2006). Jpeg example jpg rip 010.jpg, [Online]. Available: `http://en.wikipedia.org/wiki/File:JPEG_example_JPG_RIP_010.jpg` (visited on 03/23/2012).

[16] —, (2006). Jpeg example jpg rip 100.jpg, [Online]. Available: `http://en.wikipedia.org/wiki/File:JPEG_example_JPG_RIP_100.jpg` (visited on 03/23/2012).

[17] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression”, *Information Theory, IEEE Transactions on*, vol. 23, no. 3, pp. 337–343, May 1977, ISSN: 0018-9448.

[18] —, “Compression of individual sequences via variable-rate coding”, *Information Theory, IEEE Transactions on*, vol. 24, no. 5, pp. 530–536, Sep. 1978, ISSN: 0018-9448.

List of Figures

List of Tables

List of Algorithms