LOSSLESS AUDIO DATA COMPRESSION

by

Kan Zhao

Thesis
submitted in partial fulfillment of the requirements for
the Degree of Master of Science (Computer Science)

Acadia University
Fall Convocation 2007

© by Kan Zhao, 2007
NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni les extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.
This thesis by Kan Zhao was defended successfully in an oral examination on August 22nd, 2007.

The examining committee for the thesis was:

Dr. Nelson O’Driscoll, Chair
Dr. Stavros Konstantinidis, External Reader
Dr. Darcy Benoit, Internal Reader
Dr. James Diamond, Supervisor
Dr. Danny Silver, Director

This thesis is accepted in its present form by the Division of Research and Graduate Studies as satisfying the thesis requirements for the degree Master of Science (Computer Science).
Contents

Abstract x

Acknowledgments xi

1 Introduction 1
  1.1 Basic Ideas of Data Compression 1
  1.1.1 Encoder and Decoder 2
  1.1.2 Lossless Codecs and Lossy Codecs 3
  1.1.3 Performance Criteria for Compression Systems 4
  1.2 Lossless Audio Compression 4
    1.2.1 Digital Audio 5
    1.2.2 Digital Audio Format – WAVE 8
    1.2.3 History of Lossless Audio Compression 9
  1.3 Thesis Layout 9

2 Background 11
  2.1 Information Theory and Entropy 11
  2.2 Huffman Coding 13

3 Principles of Lossless Audio Compression 16
  3.1 “Prediction/Residual” Structure 16
  3.2 Linear Prediction Model 19
    3.2.1 Digital Filters 19
    3.2.2 Formulate Predictive Coefficients 20
# List of Tables

1.1 A list of lossless audio codecs ........................................... 10

2.1 Huffman codes for the symbols and corresponding Huffman tree of Figure 2.1 ......................................................... 15

3.1 Golomb codes for $m_G = 5, 8, \text{ and } 10$ .......................... 26

3.2 Golomb-Rice coding for $m = 2$ ........................................... 26

4.1 Test samples [Whi05] .......................................................... 30

5.1 The expectation of Rice code length ..................................... 38

5.2 The average length of codes generated from Golomb-Rice coding, GRHC, and Huffman coding when encoding data with Laplacian distribution 45

5.3 The improvements achieved by applying GRHC coding ............ 48

5.4 The encoding time differences between Golomb-Rice coding and Progressive GRHC. CPU time is measured in seconds. .......... 49

6.1 The compression ratio of the codec when applying flexible blocking Method 1 with judgment element $R$ $[0.4: 1.0]$ ...................... 57

6.2 Fixed blocking vs. flexible blocking Method 1 ......................... 57

6.3 Fixed block size vs. flexible blocking Method 2 when initial sizes of block A and B are 1K, 2K, or 4K ................................. 58

6.4 Fixed block size vs. flexible blocking Method 1 and Method 2 ...... 59

7.1 Fixed blocking vs. variable blocking of FLAC .......................... 65
7.2 The comparison of compression ratios achieved by using different encoding schemes; † the first two minutes of the corresponding file . . . 68
List of Figures

1.1 Encoder and decoder ............................................ 3
1.2 Sampling a sound wave [Sal07, p. 720–727] .................. 7

2.1 An example of building a Huffman tree ......................... 14

3.1 Encoding and decoding processes .............................. 18
3.2 FIR and IIR filters .............................................. 20
3.3 Linear predictions with different orders ....................... 23
3.4 Laplacian distribution with $\sigma^2 = 25$ ....................... 24

5.1 Samples, predictions and residuals ............................ 32
5.2 Residual occurrence probabilities vs. its corresponding Laplacian distribution .............................................. 34
5.3 A binary codeword tree of Golomb-Rice code prefixes .......... 35
5.4 The bit lengths of a residual sequence encoded by Golomb-Rice coding with Rice parameter 0 through 8 .................. 36
5.5 The binary codeword tree of Golomb-Rice coding with probabilities marked for each leaf ............................ 37
5.6 The expectation of the frame length vs. the actual frame length ... 39
5.7 Entropy vs. bits per Rice code .................................. 41
5.8 Laplacian distribution vs. optimal Rice code distribution .... 42
5.9 The flow chart of Golomb-Rice combined with Huffman coding ... 43
5.10 The improvements of GRHC over Golomb-Rice when encoding data source with Laplacian distribution .................. 46
5.11 The improvements of GRHC over Golomb-Rice when encoding the residuals output from a linear prediction model ................................................. 47
5.12 The occurrence probabilities of Rice parameter $m$ ............................................ 48

6.1 Local optimum and global optimum ................................................................. 51
6.2 An audio stream processed by fixed blocking and flexible blocking .............. 52
6.3 The flow chart for flexible blocking (Method 1) ............................................... 55
6.4 Example 1 for flexible blocking (Method 1): the breakpoint is set in middle of the initial block B ................................................................. 56
6.5 Example 2 for flexible blocking (Method 1): the breakpoint is specified until the sum of the size block A and block B reaches the maximum buffer size ................................................................. 56

7.1 Extreme block size search example ................................................................. 67
7.2 An example of the breakpoints selected by the extreme block size search scheme ................................................................. 68
Abstract

Lossless audio compression is a rapidly developing field with a strict stipulation: the decompressed sample information must match the original audio data. A common strategy applied by modern compression systems is a “prediction/residual” structure. This strategy generates a series of estimates for audio samples and applies coding schemes to encode the residuals (the differences between the actual values and the estimates).

This thesis studies two main areas of the “prediction/residual” structure: residual coding and blocking, and their respective algorithms.

A common algorithm for residual coding is Golomb-Rice coding, a simple general-purpose integer coding algorithm. It takes advantage of the fact that the distribution of the residuals output from a prediction model generally follows a Laplacian distribution. Based on the study of Golomb-Rice coding, a new residual coding scheme, Golomb-Rice with Huffman coding (GRHC) is designed. GRHC coding can produce codes that can be adapted (to a degree) to the actual distribution of the residuals output from linear prediction model.

Furthermore, a new idea, flexible blocking, is introduced. Instead of blocking the sequence of samples into fixed length blocks, the algorithm can flexibly block samples to produce an improved compression ratio. This thesis introduces two flexible blocking schemes and an extreme block size search scheme, as well as implements two of these methods in an open-source codec – FLAC.
Acknowledgments

My foremost thanks go to my supervisor Dr. Jim Diamond. As a teacher, he introduced me to the world of data compression techniques, and directed my way on the right path of research. As a friend, he helped me start my graduate study life in Canada. I truly appreciate his encouragement and guidance, and would like to say “xie xie” to him.

I would like to extend my thanks to the department faculty and staff. I appreciate all their support and help. It was definitely a pleasant experience studying in the Jodrey School of Computer Science.

Finally, my deepest thanks go to my parents and my friends who have always been very supportive. Their love and care were my driving force to go through all the way.
Chapter 1

Introduction

In the last two decades, we have witnessed a change in the way people share information and communicate: increasing numbers of multimedia data files – image, audio and video – have been created, stored and transferred with digital hard disks and the Internet; some call it a “multimedia revolution” [Say06, p. 1]. As a result of this revolution, a number of multimedia file formats have been specified and standardized: “jpg”, “gif”, “mp3” and “avi” are some well-known data formats.

The main problem with transferring multimedia data is the fact that the number of bytes required to store multimedia information in an original form can be very large; for example, a five-minute CD-quality audio file without compression takes more than 20 megabytes. Even considering the average hard disk space and the Internet speed, storing or transmitting a CD-quality file could still be inconvenient for common uses. Data compression is the key to overcoming this problem and in enabling the multimedia revolution to continue.

1.1 Basic Ideas of Data Compression

Data compression, generally speaking, is an operation that generates a representation of data requiring less space than the original data itself. Although this idea may be unfamiliar to most people, data compression techniques and applications have been spreading into and affecting people's daily lives. They are transforming the way
people live, work and play. For instance, during a phone call, speech is compressed, and its compacted form is transmitted via phone lines to the party at the receiving end of the call. Moreover, without compression, it would be impractical to put large images and background music in web pages. By contrast with a five-minute CD-quality audio file, an MP3 file (an audio file in a particular compression format) with the same length of music may take only one or two megabytes. Data compression techniques not only reduce the amount of space needed for storage, but also reduce the time needed to transfer data. Data compression has become pervasive in modern life.

The compression effect is not produced by some magic inherent within data compression techniques. Generally speaking, data compression techniques take advantage of some essential features and rules underlying human senses or certain types of files. Statistical models, particular patterns or human perceptual features are considered during the design of a compressor. For example, in English texts, as is well known, the character 'e' occurs more frequently than the character 'q'. Normally, each character is stored in one byte (eight bits). If, instead, we use fewer bits to represent 'e', and more bits to represent 'q', generally more bits will be saved with the shorter representation of 'e' than lost with the longer representation of 'q'. Ultimately, this produces a compression effect. (A statistical model for the English alphabet is used in this example.) Another common strategy adopted is to replace some patterns or words commonly seen in English texts, such as "the", "and" and "of". Instead of using three bytes for the word "the", we may replace it by a certain symbol requiring fewer bits to encode, and, as a result, this strategy can save some bits.

1.1.1 Encoder and Decoder

Generally, a compression technique, or codec, is comprised of two components: an encoder and a decoder. The encoder operates on the original data and outputs a file in a compacted form; this operation is called compression or encoding. The decoder, as a reverse process, operates on the compacted file, and outputs a data file; this operation is called decompression or decoding. The relationship between data files
and a codec is illustrated in Figure 1.1.

![Diagram of Encoder and Decoder]

Figure 1.1: Encoder and decoder

1.1.2 Lossless Codecs and Lossy Codecs

The decoded file may or may not be the same as the original data. According to the requirements of reconstruction, compressors can be classified as one of the two following types: lossless codecs and lossy codecs.

- **Lossless codecs** are ones that gain compression effects without loss of information; that is, the decoder reproduces the original data exactly. Text compressors are typically lossless, since a text, before and after compression, is almost always required to be the same. Any small change in reconstruction results in a different meaning; for example, changing “x += 3;” to “x -= 3;” in a computer program likely results in a significant change to the program. Some typical lossless compression methods or systems are: Lempel-Ziv coding (general-purpose compression), FLAC (audio compression), and Portable Network Graphics (PNG) (image compression).

- **Lossy codecs** are ones for which the reproduced data files are not the same as the original files. Multimedia information is usually the target domain for
lossy techniques, since, as mentioned above, multimedia information consumes significant amounts of storage space. With a lossy codec, the decoded data is not identical to the original data; rather, the decoded data is maintained at some level of acceptable quality. The kinds of information which can be eliminated or omitted while minimizing the impact on multimedia quality are considered and judged during the codec design process. An encoder preserves the information that human senses are more sensitive to, and reduces the information that is not so perceptible. Therefore, the resulting multimedia information still may satisfy human senses. Normally, lossy codecs offer different compression levels to satisfy variable output quality requirements. Some typical lossy formats are: JPEG (a lossy image format) and MP3 (a lossy audio format).

1.1.3 Performance Criteria for Compression Systems

The performance of a compressor can be evaluated in a number of different ways. The time consumed by encoding and decoding, the memory required and the CPU time consumption are three common performance criteria. In general, Compression Ratio is the most significant issue involved in measuring a compressor's performance; Compression Ratio is the number of bits required to represent the data after compression divided by the number of bits before compression.

$$\text{Compression Ratio} = \frac{\text{Compressed File Size}}{\text{Original File Size}} \times 100\% \quad (1.1)$$

As can be seen, a compression ratio less than one indicates a compression effect, and the smaller the compression ratio, the better the compression. For example, if an original text file is 4,000 bytes, and after compression, the data in compacted form is 1,000 bytes, we say that the compression ratio is 25%.

1.2 Lossless Audio Compression

Lossless audio compression techniques are a class of techniques which can compress audio data without losing information; that is, the reconstructed information output
CHAPTER 1. INTRODUCTION

from the decoding process is required to match the original audio data. The development of lossless audio compression is initially motivated by the need of audio engineers and audiophiles to preserve a perfect copy of their audio files; it is also inspired by the natural desire of keeping information complete. In the interest of research and analysis for some special audio files, e.g., passive sonar data, it is necessary to preserve the exact original waveforms while obtaining significant compression results to save transmission time. For musicians and audiophiles, maintaining the full quality of an audio file is sometimes the preferred choice.

Prior to the appearance of lossless audio compressors, a number of general-purpose lossless compressors existed. The reason for introducing specific lossless audio compression is due to the fact that general-purpose compressors produce a low compression effect on audio files. The property of audio signals makes them statistically different from some other types of structured data, such as English texts [Say03, chap. 12]. The ASCII codes for English characters can be represented in a particularly narrow range, which are from 65 to 122 for characters 'A' to 'Z' and 'a' to 'z'. Moreover, there are some essential structures within English texts, such as certain words or characters repeatedly occurring. Therefore, a codebook for English could be relatively small and usually constant. By contrast with that, 16-bit signed audio samples have a wider range of values, from 32767 to −32768, and it is unforeseeable that some sample values would have higher occurrence probabilities. Therefore, audio engineers and technicians started investigating lossless audio compressors.

Before exhibiting lossless audio compression techniques, a basic introduction to digital audio is given: the representation of digital audio files, the way they are constructed, and some main features are explained.

1.2.1 Digital Audio

Digital audio is a type of data that stores sound information in digital form. Digital audio tape (DAT), compact disk (CD) and Minidisk (MD) are some common mediums for storing digital audio files. Besides those, there are some portable digital audio players that can store, organize, and play digital audio files, such as MP3 players and
iPods.

Sound, as a familiar phenomenon, can be considered as a wave propagating in a medium, such as air [Sal07, p. 720–727]. Every wave has two attributes: its frequency and amplitude. The frequency is the number of periods that occur in one time unit, such as one second [Sal07, p. 720–727]; it is measured in Hertz (Hz). The amplitude is a nonnegative scalar measure of a wave’s magnitude of oscillation.

In a digital audio file, the wave information is recorded as a sequence of individual numbers, called samples in audio processing. The process of converting the wave into samples is called sampling or analog-to-digital conversion. It can be described in the following three steps:

- First, when a sound is sensed by a microphone, the wave or vibration is analogously represented or converted into a voltage which varies continuously with time [Sal07, p. 720–727].

- Then, an analog-to-digital converter digitizes the voltage at a given sampling rate and bit resolution, and generates the series of individual sample values. Figure 1.2(a) demonstrates a typical sound waveform; Figure 1.2(b) presents a sequence of samples digitized from that waveform. There are two features of digital audio that need further explanation:

  - **Sampling rate**: the rate at which samples are captured at equal intervals. The sampling rate is measured in samples per second.
  
  - **Bit resolution**: the number of bits in the digital representation of each sample. Eight bits, 16 bits and 24 bits per sample are common resolutions.

- Finally, the sequence of samples is saved into a digital audio file.

As a reverse process, digital-to-analog conversion is the process that converts the sequence of individual samples back into voltages. In order to play digital audio files through speakers, a digital-to-analog converter, which can usually be found in CD players and PC sound cards, transforms the data in digital audio form back into the analog signal waveform.
CHAPTER 1. INTRODUCTION

(a) A sound wave

(b) A sequence of samples digitized from the wave shown in Figure 1.2(a)

(c) A sequence of samples digitized from the wave shown in Figure 1.2(a) at a lower sampling rate than the sampling rate in Figure 1.2(b); the reproduced waveform (in dashed line) is not identical to the original waveform

Figure 1.2: Sampling a sound wave [Sal07, p. 720–727]
CHAPTER 1. INTRODUCTION

There is one point to consider: the sampling rate cannot be arbitrary. Figure 1.2(c) shows another sequence of samples which is also digitized from the waveform shown in Figure 1.2(a) at a lower sampling rate. The reproduced waveform (dashed line) is different than the original signal waveform. Harry Nyquist proved that the sampling rate should be a little over the Nyquist frequency [Nyt28], which is twice the maximum frequency contained in a sound [Sal07, p. 720–727]. Sampling at or above the Nyquist frequency rate avoids this distortion.

Studies show that the human ear is sensitive to a wide range of sound frequencies, normally from 16–20 Hz to 20,000–22,000 Hz [Sal07, p. 720–727]. The CD quality standard defines its sampling rate as 44,100 Hz, which is slightly larger than twice of the maximum frequency a human ear can hear. That means a digital audio file with CD quality preserves the same range of frequencies that the human ear can hear.

1.2.2 Digital Audio Format – WAVE

There are a number of file formats for storing audio data. The WAVE file format is one of the most widely supported uncompressed digital audio formats. It is one subtype of RIFF (Resource Interchange File Format), a tagged structure for multimedia resource files. The WAVE format defines a few chunks, each of which contains its own header and data. Simply, a basic WAVE file is comprised of two parts:

1. At the front of the file, there is a block of data containing information about the audio file, including the file length, sampling rate, bit resolution, number of samples, etc.

2. Following this overhead, there is a block of data containing all the individual samples, which are ordered in time.

For a more detailed description of the structure and inner workings of this format, see [Sal07, sec. 7.4].
1.2.3 History of Lossless Audio Compression

Lossless audio compression has been a rapidly developing field over the last 10 years or so. Compared with lossy audio compression techniques, lossless compression techniques have been considerably later identified and accepted. In 2006, the MPEG-4 Audio Lossless Coding (ALS) became an ISO/IEC standard, which is 15 years later than the MP3 format, which became an ISO/IEC standard in 1991. The postponed development can be explained by two main factors: the strict requirement of lossless compression itself and the availability of suitable hardware capacity. The amount of intrinsic information inherent within an audio waveform limits the minimal compression ratio lossless compressors can reach. As expected, a losslessly-compressed file would be larger than a lossily-compressed one. To facilitate the development of lossless audio compression, it is important to have abundant storage space and higher-speed Internet available. Nowadays, both of these two elements are becoming ubiquitous.

Currently, a number of lossless audio compressors have been published. Table 1.1 presents a list of codecs which are usually used for comparing lossless audio compression performance. The compression ratio of a good lossless audio compressor is normally around 45% to 55% on stereo files [Coa07, Mal07], which is roughly three times larger than the compression ratio of lossy compressors.

1.3 Thesis Layout

This thesis is focused on investigating lossless audio data compression techniques as well as designing new techniques to enhance their compression performance, including:

- Analyzing the performance of a "prediction/residual" structure.
- Exploring the performance of Golomb-Rice coding (a structured entropy coding) when encoding the residuals of the prediction model.
- Introducing a new technique, Golomb-Rice combined with Huffman coding.
- Designing a new scheme to generate optimally-sized blocks.
<table>
<thead>
<tr>
<th>Codec</th>
<th>Author</th>
<th>Source</th>
<th>Last Release</th>
</tr>
</thead>
</table>
| FLAC        | Josh Coalson          | Open
http://flac.sourceforge.net | 1.1.4         |
| Monkey’s Audio | Matthew T. Ashland  | Open
http://www.monkeysaudio.com  | 4.01b2       |
| OptimFROG   | Florin Ghido          | Closed
http://www.losslessaudio.org | 4.600        |
| WavPack     | David Bryant          | Open
http://www.wavpack.com       | 4.41         |
| TTA (True Audio) | Alexander Djourik  | Open
http://www.true-audio.com    | 3.3          |
| LA          | M. Bevin              | Closed
http://www.lossless-audio.com/index.htm | 0.4          |
| TAK         | Thomas Becker         | Closed
http://www.thbeck.de/index.html(German) | 1.0.1        |
| MPEG-4 ALS  | the MPEG audio subgroup | Closed
[LMH+05, LR05, Lie04, LRMY04, MYL04, LR04, Lie03] | 1.00         |

Table 1.1: A list of lossless audio codecs

This thesis is organized as follows. Some general background information, information theory as well as the principles of lossless audio compression, is introduced in Chapters 2 and 3. In Chapter 4, we describe a simplified lossless audio compressor implemented to statistically analyze the performances of the “prediction/residual” structure. Chapters 5 and 6 present new methods applied to residual coding techniques and the flexible blocking scheme. A related result derived by introducing flexible blocking ideas into an open-source lossless audio codec, FLAC, is presented in Chapter 7. Finally, Chapter 8 concludes with some remarks regarding this research work, as well as suggestions for future works.
Chapter 2

Background

In this chapter, a basic mathematical prerequisite for quantifying both lossy and lossless compression – entropy – is explained. It is a fundamental concept of information theory.

2.1 Information Theory and Entropy

In 1948, Claude E. Shannon established that, on average, the number of bits needed to represent the result of an uncertain event is given by its entropy, denoted by $H(X)$ [Sha48]:

$$H(X) = - \sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$$  \hspace{1cm} (2.1)

where $X$ is a discrete random variable with possible outcomes $x_1$, $x_2$, \ldots, $x_n$, and $p(x_i)$ is the occurrence probability of outcome $x_i$. For a given (memoryless) source, the compression performance of any lossless algorithms can approach its entropy value. Herein, $-\log_2 p(x_i)$ is defined as the self-information of outcome $x_i$; it represents the amount of information within the outcome $x_i$ [Sha48]. The lower the probability of an event, the greater the associated information.

The entropy value depends on the statistical nature of a given source. For instance, when we toss a fair coin, the probability of obtaining a head or a tail is the same.
CHAPTER 2. BACKGROUND

There is a 50% chance that any given toss will give a head, \( p(\text{head}) = 0.5 \), and a 50% chance of a tail, \( p(\text{tail}) = 0.5 \). The result is a head or a tail, which can be expressed as either 0 or 1 using one bit. Theoretically, the entropy of tossing a coin is \( H(\text{Tossing Coin}) = -2 \times 0.5 \times \log_2 0.5 = 1 \) bit. The result is acceptable for a "0 or 1" or "yes or no" question. The purpose of introducing entropy is to help solve many practical problems, and to be able to express the results numerically in terms of bits [Say06, sec. 2.2].

In the standard model of data compression, we consider compressing data from a source which outputs symbols from a finite alphabet. These symbols may have a known probability distribution. For example, in text compression, the alphabet is normally the set of 128 ASCII codes [Sal07, p. 1041].

Entropy encoding is a coding scheme which assigns codes to symbols so as to vary code lengths in proportion to the probabilities of symbols. According to Shannon's theorem, an optimal code length for an input symbol (in bits) should be equal or close to its self-information value. Two of the most common entropy coding techniques are Huffman coding [Huf52] and arithmetic coding [HJH03].

The compression efficiency of a compressor can be evaluated by two measurements relating to source entropy: redundancy and code efficiency. Let \( L(x_i) \) be the length of the code associated with input event \( x_i \), and the average code length \( L_{av} = \sum_{i=1}^{n} p(x_i) \times L(x_i) \). Redundancy is the difference between the source entropy and the average code length:

\[
\text{redundancy} = L_{av} - H(X).
\]

The smaller the redundancy value, the better the compression performance. Alternatively, the compression efficiency can be represented by a ratio called code efficiency:

\[
\text{code efficiency} = \frac{H(X)}{L_{av}} \times 100\%.
\]
2.2 Huffman Coding

In 1952, David Huffman designed an algorithm called Huffman coding [Huf52]. Huffman coding was the first scheme to give an exact, optimal algorithm to code symbols from an arbitrary probability distribution, and is still used as the basis of many compression techniques, or one step in multi-step techniques [Say06, p. 41–45]. The codes generated by this algorithm are called Huffman codes.

Huffman codes can be represented as a binary tree in which each leaf corresponds to a symbol. Each code is composed by traversing the binary tree from the root node to a leaf, adding a 0 to the code every time the traversal goes to the left branch and a 1 every time the traversal goes to the right branch [Say06, p. 41–45]. The leaves representing more frequently-occurring symbols are optimally set to lower depths; correspondingly they are encoded with fewer bits than infrequently-occurring symbols.

To create a Huffman tree, initially start with a list of one-node trees, each representing one symbol with a data value indicating its occurrence probability. At each step of a loop, all these trees are sorted in descending order of their occurrence probabilities (ties can be broken arbitrarily). Two trees with the smallest probabilities are selected, and they are added as two child nodes of a new tree. The value assigned to this new tree is the sum of the probabilities of its children. The loop stops when the list is reduced to only one tree, which is the constructed Huffman tree.

For example, suppose we have five symbols \{a_1, a_2, a_3, a_4, a_5\} with occurrence probabilities \(p(a_1) = p(a_2) = 0.1, p(a_3) = 0.15, p(a_4) = 0.3, \) and \(p(a_5) = 0.35.\) The process of building the Huffman tree for this example is illustrated in Figure 2.1. Note that at step 3, there are two trees having the second smallest value. The tie is arbitrarily broken in this case.

Huffman coding can be classified as a dictionary-based compression algorithm. Generally, a dictionary-based compression algorithm maps every symbol in an input source to a corresponding bit string, called a codeword; the dictionary or codebook is the function mapping symbols to codewords [Say03, p. 80]. The codewords for the above example are listed in Table 2.1.
The average code length for this example is $L_{av} = 0.1 \times 4 + 0.1 \times 4 + 0.15 \times 3 + 0.3 \times 2 + 0.35 \times 1 = 2.2$ bits/symbol. The source entropy is

$$H = \sum_{i=1}^{5} - \log p(a_i) \times p(a_i) = 2.126 \text{ bits/symbol}.$$ 

In this case, the redundancy is $L_{av} - H = 0.074$ bits/symbol, and the code efficiency is $H/L_{av} \times 100\% = 96.64\%$. 

Figure 2.1: An example of building a Huffman tree
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Probability</th>
<th>Huffman Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>0.1</td>
<td>1110</td>
</tr>
<tr>
<td>$a_2$</td>
<td>0.1</td>
<td>1111</td>
</tr>
<tr>
<td>$a_3$</td>
<td>0.15</td>
<td>110</td>
</tr>
<tr>
<td>$a_4$</td>
<td>0.3</td>
<td>10</td>
</tr>
<tr>
<td>$a_5$</td>
<td>0.35</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.1: Huffman codes for the symbols and corresponding Huffman tree of Figure 2.1
Chapter 3

Principles of Lossless Audio Compression

In this chapter, we explore a strategy, the “prediction/residual” structure, applied to lossless audio compression. This strategy can eliminate the redundancy existing in audio data, and consequently reduce the apparent amount of information that needs compression. The algorithms of the prediction model and the residual coding are also explained in this chapter.

3.1 “Prediction/Residual” Structure

In many cases of practical interest, audio waveforms exhibit a high degree of continuity or sample-to-sample correlation, i.e., the tendency of samples to be similar to their neighbours [Kim03]. The reason is that audio samples are digitized from continuous waveforms, and the sampling rate is usually higher than the rate needed at any particular time. To take advantage of this correlation, prior to the encoding process most audio compressors apply a preprocessing component called a predictor. The predictor can reduce the sample magnitude by making a prediction of the current sample based on the knowledge of some given preceding samples, and then subtracting the prediction from the current sample value. As a result, the predictor eliminates the correlation inherent in samples before encoding.
There are several prediction methods available for exploiting the correlation between adjacent samples. A universal expression of the predictor for sample \( x_n \), which is denoted as \( \hat{x}_n \), can be formalized as:

\[
\hat{x}_n = f(x_{n-1}, x_{n-2}, x_{n-3}, \ldots, x_{n-M})
\]

where \( x_{n-1}, x_{n-2}, x_{n-3}, \ldots, x_{n-M} \) are the \( M \) samples preceding the one to be processed (\( M \) is referred as the prediction order). The prediction function, \( f(*) \), forms the basis of a given prediction method. When the prediction \( \hat{x}_n \) is close to its corresponding value \( x_n \), the difference or residual, \( e_n = x_n - \hat{x}_n \), is small.

Figure 3.1(a) demonstrates the "prediction/residual" structure of the encoding process. As shown in the figure, there are three main components:

- Blocking is a component that divides and groups an audio data stream into several chunks of appropriate length. It is a component normally included by compression techniques to restrict the amount of information processed at a time. The standard terminology refers to the chunk of original data as a "block" whereas the corresponding compressed data is referred to as a "frame". More detailed information is given in Chapter 6.

- The predictor, as mentioned above, is a component with the ability of abstracting a relationship between adjacent sample values. By building a model representing this correlation, it can generate a sequence of predictions. The residuals, which are the differences between the actual sample values and the predictions, will be encoded by the following component: the residual coding component.

- Residual coding is a component which employs entropy coding algorithms to losslessly encode residuals. Its output, a sequence of codes for the residuals, is wrapped in a frame and is stored in a compacted digital audio form.

Generally, an audio file in compacted form is comprised of a header and a sequence of frames. The file header contains properties of the audio signal stream.
Each frame consists of its own frame overhead and a sequence of residual codes. The frame overhead provides enough information so that the decoding process can start working without the knowledge of other frames. The frame overhead contains the corresponding block size, the prediction model, the residual coding algorithm, and all relevant parameters. Decoding, the reverse process, retrieves information from the compacted form and reconstructs audio samples. Samples can be reconstructed by using the equation $x_n = \hat{x}_n + e_n$, where predictions are output from the prediction model, and residual values are directly decoded from the compressed data. Figure 3.1(b) illustrates the decoding process.

Figure 3.1: Encoding and decoding processes
3.2 Linear Prediction Model

The goal of the prediction component is to reduce the variance and amplitude of a data stream. The closer a predictor can predict samples, the smaller the variance of the residuals. Most encoders adopt the linear prediction coding (LPC) concept, a method which has been widely employed in different areas of signal processing. LPC's are a type of finite impulse response (FIR) filter used in audio data compression to estimate future samples based on given previous samples [Say06, chap. 11].

3.2.1 Digital Filters

A mathematical form representing the input-output relationship of a filter is given by

\[ e_n = x_n - \left( \sum_i a_i x_{n-i} + \sum_i b_i e_{n-i} \right) \]  

(3.2)

where the values \{a_i\} and \{b_i\} are filter coefficients, \{x_i\} are the inputs, and \{e_i\} are the filter outputs [Say06, chap. 11].

An impulse response is defined as the response of the system to an impulse input with respect to time [Orf95, sec. 3.3]. Based on the existence of internal feedback, a digital filter can be classified as either a finite impulse response (FIR) filter, illustrated in Figure 3.2(a), or as an infinite impulse response (IIR) filter, illustrated in Figure 3.2(b). A FIR filter, the filter with the coefficients \{b_i\} all zero, has a fixed-duration impulse response [Say06, chap. 11]; its representation form can be simplified to \( e_n = x_n - \sum_i a_i x_{n-i} \). In contrast to FIR filters, IIR filters, where some \{b_i\} coefficients are nonzero, have internal feedback; the response to an impulse theoretically may continue forever [Say06, chap. 11].

Lossless audio compression techniques typically employ FIR filters.
3.2.2 Formulate Predictive Coefficients

A linear prediction is the weighted sum of the preceding $M$ samples, given by this formula:

$$\hat{x}_n = a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3} + \cdots + a_M x_{n-M}$$  \hspace{1cm} (3.3)

where $a_1, a_2, a_3, \ldots, a_M$ are the linear prediction coefficients. An optimal predictor is a model with a prediction function that minimizes the errors, since the function of the predictor is to reduce the amplitudes of the data to be compressed. In this section, a brief introduction is given to formulate the optimal predictive coefficients; the formulation follows [Say06, chap. 11].

A common measurement used to minimize the energy in the residual signal is the sum of the squared errors [Kim03]:

$$E = \sum_n e_n^2 = \sum_n \left( x_n - \sum_{i=1}^{M} a_i x_{n-i} \right)^2.$$  \hspace{1cm} (3.4)

To calculate the coefficients $\{a_i\}$ so as to minimize the value of $E$, we set the derivative of $E$ with respect to each $a_i$ equal to zero, and get $M$ equations with $M$ unknown values:

$$\frac{\delta E}{\delta a_k} = 2 \sum_n x_{n-k} (x_n - \sum_{i=1}^{M} a_i x_{n-i}) = 0, \quad \text{for } k = 1, 2, 3, \ldots, M$$  \hspace{1cm} (3.5)
Each Equation (3.5) can be rewritten as:

$$
\sum_{i=1}^{M} a_i \sum_n x_{n-k}x_{n-i} = \sum_n x_{n-k}x_n, \quad \text{for } k = 1, 2, 3, \ldots, M \quad (3.6)
$$

This set of equations can be approximated with the following matrix equation:

$$
\begin{bmatrix}
R(0) & R(1) & R(2) & \cdots & R(M-1) \\
R(1) & R(0) & R(1) & \cdots & R(M-2) \\
R(2) & R(1) & R(0) & \cdots & R(M-3) \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
R(M-1) & R(M-2) & R(M-3) & \cdots & R(0)
\end{bmatrix}
\begin{bmatrix}
a_1 \\
a_2 \\
a_3 \\
\vdots \\
a_M
\end{bmatrix}
= 
\begin{bmatrix}
R(1) \\
R(2) \\
R(3) \\
\vdots \\
R(M)
\end{bmatrix}
(3.7)
$$

where $R(t) = \sum_{n=1}^{M-t} x_n x_{n+t}$ is known as the autocorrelation function. The reason for saying “it can be approximated” is that there is a difference between the derivative (Equation (3.6)) and the autocorrelation (Equation (3.7)).

To see this, consider an example where a predictor with order $M = 3$ operates on $N = 10$ samples. Equation (3.5) can now be written as

$$
2 \sum_{n=4}^{10} x_{n-k}(x_n - \sum_{i=1}^{3} a_i x_{n-i}) = 0, \quad \text{for } k = 1, 2, 3
$$

That can be written as a matrix like:

$$
\begin{bmatrix}
\sum_{n=3}^{9} x_n^2 \\
\sum_{n=2}^{8} x_n x_{n+1} \\
\sum_{n=1}^{7} x_n x_{n+2}
\end{bmatrix}
\begin{bmatrix}
a_1 \\
a_2 \\
a_3
\end{bmatrix}
= 
\begin{bmatrix}
\sum_{n=3}^{9} x_{n+1}x_n \\
\sum_{n=2}^{8} x_{n+2}x_n \\
\sum_{n=1}^{7} x_{n+3}x_n
\end{bmatrix}
(3.8)
$$
As can be seen, Equation (3.8) is not exactly equal to the autocorrelation Equation (3.7). The approximation is obtained when the number of samples $N$ is large enough, so that the error can be ignored [Par]. Most lossless audio compressors employ this approximation (the autocorrelation function) to generate the prediction coefficients.

Levinson-Durbin recursion [Par] is an efficient algorithm that can solve Equation (3.7) in $\theta(M^2)$ time. Alternatively, Gaussian elimination, the iteration method or QR decomposition are some methods that can also solve Equation (3.7) [Par], but in $\theta(M^3)$ time. An additional advantage of the Levinson-Durbin recursion method is that the sum of the squared errors $E$, formulated by Equation (3.4), can be calculated as a byproduct during the process of calculating the optimal coefficients. The value of $E$ is used to optimize the linear prediction order; the smaller value of $E$, the better the prediction result. By calculating sets of linear prediction coefficients with different order values, for example in a range of zero to twelve, and their corresponding $E$ values, the Levinson-Durbin recursion method can determine the optimal linear prediction order.

Based on the procedure of calculating prediction coefficients, linear prediction coding can be classified as forward linear prediction coding or backward linear prediction coding.

- **Forward linear predictive coding** is a scheme that separates the procedure of computing coefficients from the generation of predictions. The optimal coefficients $\{a_i\}$ are estimated based on the whole block. That is, the autocorrelations are calculated with the knowledge of the whole given sequence of sample values. The coefficient values are constant while coding a given block, or in other words, the prediction model is stable. This scheme is applied by the Shorten and FLAC codecs.

- **Backward linear predictive coding** is a more complex scheme with a variable prediction model. The coefficients $\{a_i\}$ are adjusted based on the preceding samples; hence, the coefficients are variable. A backward scheme can commence generating predictions rapidly, but overall it takes more time to update the
coefficient values in both the encoding and the decoding processes.

Figure 3.3 illustrates the predictions generated with prediction orders from 0 to 3. A 0\textsuperscript{th}-order predictor outputs 0 as its prediction; a 1\textsuperscript{st}-order predictor outputs a value equal to its previous value. Research shows that a predictor with a low prediction order works reasonably well. In practice, the prediction order is less than 12.

![Linear predictions with different orders](image)

Figure 3.3: Linear predictions with different orders

To keep the same prediction model in both the encoding and the decoding processes, the value of the prediction order $M$ as well as the prediction coefficients $\{a_i\}$ must be stored in the frame header. Furthermore, as the startup data for the predictor, a number of beginning sample values are also included. For example, consider the case of a sequence of samples, $x_1, x_2, x_3, \ldots$, with an order $M = 3$ predictor. One possibility is to send samples $x_1, x_2$ and $x_3$ before the encoded residuals. When decoding, the predictor gains the knowledge of the very first three samples, and starts generating predictions. Another technique is called progressive-order prediction [MYL04]. This strategy sends the first sample $x_1$ itself, then predicts $x_2$ based on the knowledge of the one preceding sample $x_1$ and sends the code of residual $e_2 = x_2 - a_1x_1$, and then sends the code of $e_3 = x_3 - (a_1x_1 + a_2x_2)$. 
3.3 Residual Coding

In the area of audio and video coding algorithms, most cases show that the residual sequence output from a general prediction model has a two-sided exponential distribution [Rob94, XX003]. The residual sequences from linear prediction can be closely modeled by a Laplacian distribution with mean value \( \mu = 0 \):

\[
L(x) = \frac{1}{\sqrt{2\sigma^2}} e^{-\frac{|x-\mu|}{\sigma}} = \frac{1}{\sqrt{2\sigma^2}} e^{-\frac{|x|}{\sigma}}
\]

where \( \sigma^2 \) is the variance. Figure 3.4 shows a Laplacian distribution with variance \( \sigma^2 = 25 \).

![Laplacian Distribution](image)

**Figure 3.4: Laplacian distribution with \( \sigma^2 = 25 \)**

**Golomb Coding**, introduced by Solomon Golomb [Gol66], is a simple algorithm that has been widely applied by modern lossless compression systems. The algorithm can generate codebooks that are optimal or nearly optimal for integer sources with Laplacian distributions [Rob94]. Compared with Huffman coding, this algorithm can be applied directly on unbounded integer sources without building a dictionary. With a single integer parameter, denoted \( m_G \), codes can be easily generated. As a structured entropy algorithm, Golomb coding encodes smaller integers (in absolute
value) with shorter codes. The procedure of composing a Golomb code for integer value \( x \) is explored in the following sections.

Initially, if the source values are signed integers, all non-negative integers are mapped to even numbers, and all negative numbers are mapped to odd numbers:

\[
x = \begin{cases} 
2x, & \text{if } x \geq 0 \\
2|x| - 1, & \text{otherwise}
\end{cases}
\] (3.9)

Next, \( x \) is divided by the parameter \( m_G \): the quotient \( q = \left\lfloor \frac{x}{m_G} \right\rfloor \) is encoded in unary; for example, the unary representation of “0” is “”, the unary representation of “1” is “0”, and the representation of “2” is “00”. The remainder \( r = x - q \times m_G \) is written in binary. The remainder \( r \) is represented in \( \lfloor \log_2 m_G \rfloor \) bits, if \( r < 2^{\lfloor \log_2 m_G \rfloor} - m_G \), otherwise, \( r \) is represented by writing \( r + 2^{\lfloor \log_2 m_G \rfloor} - m_G \) in \( \lfloor \log_2 m_G \rfloor \) bits [Rob94]. A Golomb code for input \( x \) would be:

quotient in unary — a terminal bit ‘1’ — remainder in binary

Table 3.1 illustrates the codebooks for unsigned integers [0:19] with Golomb parameter \( m_G = 5, 8, \) and 10. The vertical line of dots separates the codes into two parts: the quotient in unary followed with the terminal bit, and the bit string for the remainder.

Golomb coding was later simplified by Robert F. Rice in 1979 [Ric79], and later extended by Pen-shu Yeh and Warner Miller in 1991 [YRM91]. The first extension, called Golomb-Rice coding, considers cases where the Golomb parameter \( m_G \) is a power of 2. The Rice parameter is denoted as \( m \), and its codes are referred to as Rice codes. Golomb-Rice coding eliminates both the need to have two different ways to represent the remainder as well as the division operation (the division by a power of two is implemented as a shift) [Sal07, sec. 2.5]. The quotient \( q \) is now \( \left\lfloor \frac{|x|}{2^m} \right\rfloor \), and the remainder \( r \) becomes \( r = |x| - q \times 2^m \). Moreover, without folding or mapping signed integer sources into non-negative integer sources, Golomb-Rice coding includes one more bit to express the sign information. A Rice code for \( x \) would be:

quotient in unary — a terminal bit ‘1’ — sign bit — remainder in binary
<table>
<thead>
<tr>
<th>x</th>
<th>$m_G = 5$</th>
<th>$m_G = 8$</th>
<th>$m_G = 10$</th>
<th>x</th>
<th>$m_G = 5$</th>
<th>$m_G = 8$</th>
<th>$m_G = 10$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1:00</td>
<td>1:000</td>
<td>1:000</td>
<td>10</td>
<td>001:00</td>
<td>01:010</td>
<td>01:000</td>
</tr>
<tr>
<td>1</td>
<td>1:01</td>
<td>1:001</td>
<td>1:001</td>
<td>11</td>
<td>001:01</td>
<td>01:011</td>
<td>01:001</td>
</tr>
<tr>
<td>2</td>
<td>1:10</td>
<td>1:010</td>
<td>1:010</td>
<td>12</td>
<td>001:10</td>
<td>01:100</td>
<td>01:010</td>
</tr>
<tr>
<td>4</td>
<td>1:111</td>
<td>1:100</td>
<td>1:100</td>
<td>14</td>
<td>001:111</td>
<td>01:110</td>
<td>01:100</td>
</tr>
<tr>
<td>5</td>
<td>01:00</td>
<td>1:101</td>
<td>1:101</td>
<td>15</td>
<td>0001:00</td>
<td>01:111</td>
<td>01:101</td>
</tr>
<tr>
<td>6</td>
<td>01:01</td>
<td>1:110</td>
<td>1:110</td>
<td>16</td>
<td>0001:01</td>
<td>001:000</td>
<td>01:1100</td>
</tr>
<tr>
<td>7</td>
<td>01:10</td>
<td>1:111</td>
<td>1:111</td>
<td>17</td>
<td>0001:10</td>
<td>001:001</td>
<td>01:1101</td>
</tr>
<tr>
<td>8</td>
<td>01:110</td>
<td>01:000</td>
<td>1:1110</td>
<td>18</td>
<td>0001:110</td>
<td>001:010</td>
<td>01:1110</td>
</tr>
<tr>
<td>9</td>
<td>01:111</td>
<td>01:001</td>
<td>1:1111</td>
<td>19</td>
<td>0001:111</td>
<td>001:011</td>
<td>01:1111</td>
</tr>
</tbody>
</table>

Table 3.1: Golomb codes for $m_G = 5, 8, \text{ and } 10$

The codewords for the signed integers [-8:8] with Rice parameter $m = 2$ are shown in Table 3.2. The three parts of a Rice code separated by the vertical lines of dots are: (1) the quotient in unary followed with the terminal bit; (2) the sign bit; and (3) the remainder in binary.

<table>
<thead>
<tr>
<th>x</th>
<th>code</th>
<th>x</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1:000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1:001</td>
<td>-1</td>
<td>1:001</td>
</tr>
<tr>
<td>2</td>
<td>1:010</td>
<td>-2</td>
<td>1:110</td>
</tr>
<tr>
<td>3</td>
<td>1:011</td>
<td>-3</td>
<td>1:111</td>
</tr>
<tr>
<td>4</td>
<td>01:000</td>
<td>-4</td>
<td>01:100</td>
</tr>
<tr>
<td>5</td>
<td>01:001</td>
<td>-5</td>
<td>01:101</td>
</tr>
<tr>
<td>6</td>
<td>01:010</td>
<td>-6</td>
<td>01:110</td>
</tr>
<tr>
<td>7</td>
<td>01:011</td>
<td>-7</td>
<td>01:111</td>
</tr>
<tr>
<td>8</td>
<td>001:000</td>
<td>-8</td>
<td>001:100</td>
</tr>
</tbody>
</table>

Table 3.2: Golomb-Rice coding for $m = 2$

Some extensional schemes have been developed for different application purposes.
Hybrid Golomb coding [XXO03] and Exp-Golomb Coding [GLL05] are two algorithms more suited for handling integer sources with distributions more highly peaked than the Laplacian distribution. Adaptive Run-Length/Golomb-Rice (RLGR) coding [Mal06] is designed to generally work well on a wide range of distributions: the Gaussian distribution, Laplacian distribution and even more peaked ones.
Chapter 4

Implementation and Test Set

A simple lossless audio codec was implemented to analyze the performance of the three internal components of the “prediction/residual” structure. The reasons for not directly analyzing some published codecs is that, first, there is a limited documentation offered from their development groups, and second, the algorithms applied by these codecs vary. Therefore, to collect accurate and statistical data related to the processing performed by these three components, a new lossless audio compressor was implemented.

The project was programmed in the C++ programming language, and runs in both Windows and Linux operating systems. It can encode audio files in the WAVE audio format, and it outputs a file structured in its own compressed data format. The project applies the basic “prediction/residual” structure; the respective algorithms of the three components are listed as follows:

- Blocking: fixed blocking and flexible blocking, explained in Chapter 6.
- Predictor: forward linear prediction with optimal prediction order.
- Residual coding: Golomb-Rice coding and Golomb-Rice combined with Huffman coding (discussed in Chapter 5).

Most of the data and figures given in the following chapters were collected and output by this project. As an analyzer, it provides several different output levels,
which directly show internal data information to developers for verifying the correctness of the system and exploring the performance of the prediction model and residual coding algorithms.

4.1 Test set

A test set of audio samples was downloaded from the website: http://www.firstpr.com.au/audiocomp/lossless. The website maintainer, Robin Whittle, collected this audio corpus, and compared the performance of lossless audio compression systems. The corpus consists of eleven music tracks with CD quality, totaling 674 megabytes. The audio files are named “00.wav” through “10.wav”. File sizes vary from 24 megabytes to 177 megabytes. This corpus gives a good range of material, covering a variety of types of musical performances. Table 4.1 lists some information about each file in the test corpus [Whi05].
### CHAPTER 4. IMPLEMENTATION AND TEST SET

<table>
<thead>
<tr>
<th>File</th>
<th>Size (bytes)</th>
<th>Length (sec.)</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>01.wav</td>
<td>177,312,620</td>
<td>1005.173</td>
<td>Janos Starker, J.S. Bach, <em>Suite 1 in G Major</em>. Sefel SE-CD 300A</td>
</tr>
<tr>
<td>04.wav</td>
<td>85,029,548</td>
<td>482.027</td>
<td>Robin Whittle, <em>Spare Luxury</em>.</td>
</tr>
<tr>
<td>05.wav</td>
<td>59,136,380</td>
<td>335.240</td>
<td>Bubbleman (Andy Van), <em>Theme from Bubbleman</em>. Vicious Vinyl Vol 3 VVLP004CD.</td>
</tr>
<tr>
<td>06.wav</td>
<td>43,998,908</td>
<td>249.427</td>
<td>ElBeano (Greg Bean), <em>Ventilator</em>. Earthcore EARTH 001.</td>
</tr>
</tbody>
</table>

Table 4.1: Test samples [Whi05]
Chapter 5

Residual Coding Scheme

In this chapter, we analyze the performance of the Golomb-Rice coding algorithm applied to the residuals generated by the linear prediction model, and based on the analysis results, we introduce a new scheme, Golomb-Rice combined with Huffman coding (GRHC). Analyzing and reformulating the residual coding scheme is motivated by the significance of residual coding on the “prediction/residual” structure; that is, the efficiency of encoding residuals critically affects the compression result of a compressor.

5.1 Residual Analysis

Golomb-Rice coding is a low-complexity algorithm widely used in lossless systems, such as Shorten, FLAC and MPEG-4. The efficiency of the Golomb-Rice coding performance on residuals is dependent on the shape of the residual distribution as well as its parameter $m$. In this section, statistical results are analyzed to investigate the residual distribution.

First, the performance of the linear prediction model is experimentally examined. Figure 5.1 shows the performance of a linear prediction model working on a block of 4K 8-bit samples. The sample sequence, illustrated in Figure 5.1(a), is closely estimated by the predictor; the corresponding prediction sequence is shown in Figure 5.1(b). An enlarged view of a period of samples versus the correlative estimations
is illustrated in Figure 5.1(c), where the original sample stream is in black, and the prediction stream is in gray. The residual sequence is shown in Figure 5.1(d). The prediction model works generally well on estimating this sequence of sample values; it significantly reduces the variance and dynamic range of the data to be encoded. The amplitudes of original sample values range from 100 to −100, while the amplitudes of the errors mostly range from 20 to −20.

![Figure 5.1: Samples, predictions and residuals](image)

Second, we analyze the distribution of residuals generated from a linear prediction model. Figure 5.2 shows the occurrence frequencies of residuals versus a Laplacian distribution with the same mean and variance value for two sets of data. Figure 5.2(a) compares the errors from the data of Figure 5.1(d), which has a mean value of −0.15 and variance of 37.85. Figure 5.2(b) shows an example of 4K 16-bit samples, where
the mean value is equal to $-79.96$ and the variance is equal to $2,224,081.77$. The
distributions of residual sequences match the corresponding Laplacian distributions
quite well. The residuals close to the mean value have the highest frequency of
occurrence, which are around 9\% and 0.05\%, respectively.

We propose a new idea that can be applied to residuals with non-zero mean value.
In Figure 5.2(b), as can be seen, the mean value of the residuals is not equal to the
expected value zero. This happens when a linear prediction model is applied to a
sequence of samples with large variance. Since Golomb-Rice coding is a symmetrical
coding algorithm, compression improvement is possible by horizontally transforming
the residual sequence: subtract the mean value from the residuals, $\hat{r}_n = r_n - \text{mean}_r$,
and encode the sequence of $\hat{r}_n$ values. The mean value of the residuals, $\text{mean}_r$, must
be added into the frame overhead. For the decoding process, residuals can be retrieved
by the equation $r_n = \hat{r}_n + \text{mean}_r$.

5.2 Some Calculations Related to the Golomb-Rice Coding

The length of the Rice code of $x$, which is equal to $\left\lceil \frac{|x|}{2^m} \right\rceil + m + 2$, varies according
to its Rice parameter $m$. The optimal $m$ value is discussed in this section.

As explained in Section 3.3, a Rice code consists of:

quotient in unary — a terminal bit — a sign bit — remainder in binary.

For purposes of discussion, divide a Rice code into two parts: a prefix and a suffix.
The leading quotient in unary and the terminal bit can be regarded as the prefix; the
last $m + 1$ bits, the sign bit and the remainder, are the suffix. The prefix indicates
the range of residuals, and the suffix specifies the residual value. For example, a Rice
code for residual 7 is "01—011" when the Rice parameter $m$ is equal to 2. The prefix
"01" indicates the residual is in the range of $[-7, -4]$ or $[4, 7]$, and the suffix "011"
specifies its actual value, 7. A binary codeword tree representing the prefix codes is
shown in Figure 5.3. Each code, obtained by a path from the root node to a leaf, is
Figure 5.2: Residual occurrence probabilities vs. its corresponding Laplacian distribution

the prefix, and the leaf represents a range of residuals. The prefix code length grows
as a linear function of the residual (except for the smallest length codeword), and the
number of residuals mapped to codewords of any particular length is $2^{m+1}$. 
Figure 5.3: A binary codeword tree of Golomb-Rice code prefixes

Theoretically, the source entropy value can be reached when the codeword tree is such that each pair of siblings has equal probabilities. For example, suppose a binary tree $T$ has root node $n$ with two child nodes $n_l$ and $n_r$, where their probabilities are denoted as $p(n_l)$ and $p(n_r)$. The entropy for node $n$ could be represented by $H(n) = -p(n_l) \times \log_2 p(n_l) - p(n_r) \times \log_2 p(n_r)$. The value of $H(n)$ is minimum when $p(n_l) = p(n_r)$. Therefore, in practice, to reduce the redundancy, $m$ is selected to ensure that 50% of residuals are in the range $[-2^m, 2^m]$. The optimal Rice parameter $m$ can be computed by Equation (5.1); this result is also presented in [Rob94].

\[
\frac{1}{2} = \int_{-2^m}^{2^m} L(x) \, dx \\
= \int_{-2^m}^{2^m} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \, dx \\
= -e^{-2^m \sqrt{2/\sigma^2}} + 1 \\
\]

\[
m = \log_2 \left( \ln 2 \sqrt{\sigma^2 / 2} \right) \tag{5.1}
\]

The Rice parameter $m$ can only be an integer value, whereas the result of Equation (5.1) is a real number. The integer $m$ is set to be the value of the rounded
real value of $m$: $\lceil m + 0.5 \rceil$. For example, the occurrence probabilities shown in Figure 5.2(a) have variance $\sigma^2 = 37.85$; the result of Equation (5.1) for this variance is $m = \log_2 \left( \ln 2 \sqrt{\sigma^2/2} \right) = 1.59$. Figure 5.4 demonstrates the bit lengths of this residual sequence encoded by Golomb-Rice coding with Rice parameter ranging from 0 to 8. When $m = \lceil 1.59 + 0.5 \rceil = 2$, the length of the encoded residual sequence is minimized.

Figure 5.4: The bit lengths of a residual sequence encoded by Golomb-Rice coding with Rice parameter 0 through 8

Furthermore, by taking advantage of the value of the optimal Rice parameter $m$, the expectation of Rice code length can be estimated. As described above, residuals in the range of $[-2^m+1, 2^m-1]$ are encoded with $m+2$ bits, residuals in $[-2^{m+1}+1, -2^m]$ and $[2^m, 2^{m+1} - 1]$ are encoded with $m + 3$ bits, and so on. Let

$$ r_i = \begin{cases} [ -2^m + 1, 2^m - 1 ], & i = 0 \\ [ -2^{m+i} + 1, -2^{m+i-1} ] \cup [ 2^{m+i-1}, 2^{m+i} - 1 ], & i = 1, 2, \ldots \end{cases} \quad (5.2) $$
Then the residuals in range \( r_i \) will be mapped to Rice codes with length \( L(r_i) \):

\[
L(r_i) = m + 2 + i, \quad \text{for } i = 0, 1, 2, \ldots
\]

The corresponding probability of range \( r_i \), denoted as \( p(r_i) \), can be approximated as:

\[
p(r_i) = \begin{cases} 
\frac{1}{2}, & i = 0 \\
\frac{1}{2} \int_{2^{m+i-1}}^{2^{m+i}} L(x) \, dx = \frac{2^{2^{(i-1)}} - 1}{2^{2^i}}, & i = 1, 2, \ldots
\end{cases}
\]

The probabilities of these residual ranges would be \( \frac{1}{2}, \frac{1}{4}, \frac{3}{16}, \frac{15}{256}, \ldots \), which is marked in its binary codeword tree; see Figure 5.5.

Figure 5.5: The binary codeword tree of Golomb-Rice coding with probabilities marked for each leaf.
CHAPTER 5. RESIDUAL CODING SCHEME

By knowing the length of residuals in each range \( r_i \) and their respective probabilities, the expectation of the Rice code length, \( E(\text{Rice}) \), can be easily calculated:

\[
E(\text{Rice}) = \sum_{i=0}^{16} p(\text{r}_i) \times L(\text{r}_i)
\]

\[
= \frac{1}{2} \times (m + 2) + \sum_{i=1}^{16-i} \left( \frac{2^{2^{(i-1)}} - 1}{2^i} \times (m + i + 3) \right)
\] (5.4)

By coding a small program to calculate Equation (5.4), we obtain the expected value of Rice code length with different values of \( m \) for the cases of samples with 8-bit and 16-bit resolutions. The results, presented in Table 5.1, show the expected length of Rice codes, when the value of \( m \) is optimal. In general, the expected value of Rice code length is equal to its \( m \) value plus (roughly) 2.82.

<table>
<thead>
<tr>
<th>( m ) value</th>
<th>( E(\text{Rice}) ) (bits/symbol)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8 bits/sample</td>
</tr>
<tr>
<td>0</td>
<td>2.82 = m + 2.82</td>
</tr>
<tr>
<td>1</td>
<td>3.82 = m + 2.82</td>
</tr>
<tr>
<td>2</td>
<td>4.82 = m + 2.82</td>
</tr>
<tr>
<td>3</td>
<td>5.82 = m + 2.82</td>
</tr>
<tr>
<td>4</td>
<td>6.78 = m + 2.78</td>
</tr>
<tr>
<td>5</td>
<td>7.19 = m + 2.19</td>
</tr>
<tr>
<td>6</td>
<td>8.82 = m + 2.82</td>
</tr>
<tr>
<td>7</td>
<td>9.82 = m + 2.82</td>
</tr>
<tr>
<td>8</td>
<td>10.82 = m + 2.82</td>
</tr>
<tr>
<td>9</td>
<td>11.82 = m + 2.82</td>
</tr>
<tr>
<td>10</td>
<td>12.82 = m + 2.82</td>
</tr>
<tr>
<td>11</td>
<td>13.82 = m + 2.82</td>
</tr>
<tr>
<td>12</td>
<td>14.75 = m + 2.75</td>
</tr>
</tbody>
</table>

Table 5.1: The expectation of Rice code length

As a further advantage, the expectation of frame length can be calculated. It should be approximately equal to the sum of the length of the frame overhead, the length of the uncompressed symbol information to start the prediction process and the expectation of Rice code length multiplied by the number of samples to be compressed, as shown in Equation (5.5).
\[ E(\text{frame size}) = \text{length of frame overhead} + \text{bit resolution} / 8 \times \text{number of unpredicted samples} + E(Rice) \times \text{number of predicted samples}. \]

(5.5)

This low-complexity estimation of frame length can be placed prior to the process of generating predictions. As previously discovered, the Levinson-Durbin recursion calculates the sum of squared residuals \[ E = \sum_n e_n^2. \] Since the mean of the residuals, in theory, is close to zero, the residual variance should be close to \( E/N \), where \( N \) is the number of samples in the frame. The estimation of frame length can be computed by Equation (5.4) and Equation (5.5).

Figure 5.6 shows the comparison of the estimation of frame length versus the actual frame length for the test data “00.wav”; the X axis is the block index and the Y axis is the byte count. The average estimation error is approximately 4%. This difference is due to the fact that the distribution of actual residuals does not completely match the theoretic Laplacian distribution.

![Figure 5.6: The expectation of the frame length vs. the actual frame length](image)
5.3 Golomb-Rice combined with Huffman coding

In this section, we illustrate a new scheme: Golomb-Rice combined with Huffman coding (GRHC). Golomb-Rice coding is a low-complexity algorithm with fast encoding and decoding speed which has been applied by a number of lossless audio codecs. The major disadvantage is that its simple mapping scheme reduces the flexibility of obtaining optimal codebooks on source data with arbitrary distributions. The GRHC coding scheme can produce codes that can be adapted (to a degree) to the actual distribution of the residuals, while maintaining the real-time encoding and decoding enjoyed by the Golomb-Rice coding.

Figure 5.7 shows a comparison between the entropy and the average length of Rice codes with residuals from a sequence of blocks. This result was found experimentally by encoding one test file, "00.wav". The X axis indicates the block index number, and the Y axis is the number of bits per code. The code efficiency displayed is around 96.9%, and the average difference between entropy and the average Rice code length is approximately 0.183 bit/code. This means that Golomb-Rice coding performs efficiently on encoding residuals, and the improvement which can be made by introducing a new coding scheme is limited.

The idea of introducing the Huffman coding scheme to the area of residual coding techniques is motivated by the fact that Huffman coding can generate optimal codewords from given symbols with an arbitrary probability distribution, whereas the Golomb-Rice coding is optimal for the geometric distribution\(^1\) only. Huffman coding ensures that symbols occurring more frequently (or having higher probability of occurrence) will be assigned to shorter codes than those symbols that occur less frequently. The average length of Huffman codes is theoretically closer to the source's entropy value than the average length of Rice codes.

The difficulty of employing Huffman coding to encode residuals is that a codebook is required in the decoding process. Without the information of the codebook or probability distribution, it is impossible to map codes back to the original dataset.

\(^1\)A random variable \(X\) has a geometric distribution if \(p(X = k) = q^{k-1} \times p, \) for \(k = 1, 2, 3, \ldots\)
However, the space for storing the codebook can negate the advantage of the compression effect. For instance, when encoding a block of 4K 8-bit samples, roughly 512 additional bytes would be needed to store the dictionary (using a straight-forward encoding). To overcome this problem, we take advantage of the assumption that the probability distribution of residuals can be closely modeled by the Laplacian distribution. The probability of residual $e$ can be estimated as a uniformly integrated Laplacian distribution, $\int_{e-0.5}^{e+0.5} L(x) \, dx$. A Huffman tree can be built based on these assumed probabilities, both in the encoding and the decoding processes. It only requires a floating point number, the variance $\sigma^2$, added into the frame overhead to indicate the shape of the corresponding Laplacian distribution.

Aside from the space of the dictionary, there is another consideration: the size of a Huffman tree. Building a binary tree requires a significant amount of memory space. The residuals, for 16-bits samples, would range from $-2^{16}$ to $2^{16} - 1$. The corresponding binary tree has $2^{17}$ leaves, which is a tree with a total of $2^{18} - 1$ nodes. Each node needs some tree structure information, e.g. the item value (in our case it would be the probability), and pointers upwards and/or downwards. Building such a
large Huffman tree is expensive in both time and space, and would cause considerable delays with current computing hardware.

Figure 5.8 shows two probability distributions: a Laplacian distribution and a distribution corresponding to the lengths of the Rice codes for \( m = 1 \). The central region, \([-2^m + 1 : 2^m - 1]\), has the largest differences. This observation motivated the design of GRHC. It applies different methods on the central and tail regions: residuals in the center area, \([-2^m + 1 : 2^m - 1]\), are encoded by Huffman coding, and the rest, the two-side tails, are still processed by the Golomb-Rice coding algorithm.

![Laplacian distribution vs. optimal Rice code distribution](image)

**Figure 5.8: Laplacian distribution vs. optimal Rice code distribution**

In GRHC, first, a limited-size Huffman tree is built based upon the sequence of assumed probabilities of residuals from \(-2^m + 1\) to \(2^m - 1\), and the corresponding codewords are generated.

Second, the residuals are assigned to corresponding codes. As mentioned above, the Rice codes for residuals \(|e| < 2^m\) would start with "1", and Rice codes for the tail parts, \(|e| \geq 2^m\), start with "0". The GRHC algorithm takes advantage of this difference in the first bit of a Rice code. GRHC assigns the symbols in the central region to a bit string starting with "1" and followed with its Huffman code, and the
code for the tail regions remains the same. For the decoder, the different coding schemes can easily be distinguished by examining the first bit of the code. Figure 5.9 demonstrates a flow chart of its encoding process.

![Flow chart of Golomb-Rice combined with Huffman coding](image)

Figure 5.9: The flow chart of Golomb-Rice combined with Huffman coding

Table 5.2 shows the theoretic average bits per symbol of three coding algorithms: Golomb-Rice coding, GRHC coding, and Huffman coding working on a dataset with Laplacian distribution, when $m$ ranges from zero to eight. The table also lists the entropy value of the Laplacian distribution, which is:

$$H = - \sum_{n=-2^{16}}^{2^{16}-1} p(n) \log_2 p(n),$$

for 16-bit data, where $p(n) = \int_{n-0.5}^{n+0.5} L(x) \, dx$.

In theory, the relation between the optimal Rice parameter $m^*$ and the Laplacian variance $\sigma^2$ can be formalized as follow:

$$m^* = \log_2 \left( \ln 2 \sqrt{\sigma^2 / 2} \right).$$
In practice, the Rice parameter \( m \) must be an unsigned integer, thus we assign \( m = \lfloor m^* + 0.5 \rfloor \). The value of the variance \( \sigma^2 \) in any given line of Table 5.2 is the optimum value for the corresponding value of \( m \). Since \( \sigma^2 \) can not be equal to zero in the function for \( m^* \), we approximately set it to a small value, 0.24, whose corresponding \( m^* \) value is almost equal to zero.

Figure 5.10 demonstrates the percentage improvement of Huffman coding, GRHC coding, and the entropy over Golomb-Rice coding. As displayed in this figure, the differences are concentrated in a small range, and GRHC performs reasonable close to the compression limitation (the entropy value). The overall average improvement of GRHC over the possible improvement, \((\text{Rice} - \text{GRHC}) / (\text{Rice} - \text{Entropy}) \times 100\%\), is \( 33.52\% \), which is a significant amount of improvement. When the Rice parameter is small, GRHC produces between 3% and 20% improvement over Golomb-Rice coding \((\text{GRHC} - \text{Rice}) / \text{Rice} \times 100\%\). For instance, when \( m^* = 0.99 \), rounded \( m = 1 \), the average code length of GRHC is 4.0301 bit/code, while Golomb-Rice performs at 4.1892 bit/code, Huffman performs at 4.0150 bit/code, and the entropy is 3.7302 bit/code. The improvement of GRHC over Golomb-Rice is 3.8% in this case.

The bumps in Figure 5.10 between integer numbers (for example, the region between 1.0 and 2.0) are due to the difference between the optimal value of \( m \) and its rounded integer value. All values of \( m \) ranging from 0.5 to 1.4999... are rounded to 1. The compression performance of GRHC and Golomb-Rice is reduced by the rounding operation. Their performance varies from not so close to optimal to very close to optimal (when \( m \) is equal to 1.0), and back to not so close to optimal.

The results shown above demonstrate the theoretical compression performance of these three entropy coding algorithms. Since there would be differences between the academic experiment and real cases, we applied the GRHC coding to encode the test audio set, and compared GRHC's result with Golomb-Rice coding. Figure 5.11 shows the improvement of GRHC over Golomb-Rice coding working on sequences of blocks with different Rice parameter \( m \), and the relative occurrence probabilities of the value of \( m \) are plotted in Figure 5.12. GRHC performs more optimally than Golomb-Rice coding when \( m \leq 4 \). When \( m > 4 \), the performance of GRHC is close to and a little worse than Golomb-Rice coding. The reason is that when the variance (and hence
\[\begin{array}{|c|c|c|c|c|c|}
\hline m^* & m \ (\text{int}) & \sigma^2 & \text{Golomb-Rice} & \text{GRHC} & \text{Huffman} & \text{Entropy} \\
\hline 0.00 & 0 & 0.24 & 2.2517 & 1.4892 & 1.3846 & 1.1071 \\
0.25 & 0 & 5.89 & 3.6916 & 3.4388 & 3.2745 & 3.2379 \\
0.49 & 0 & 8.33 & 4.0199 & 3.8026 & 3.5181 & 3.4834 \\
0.75 & 1 & 11.77 & 3.9598 & 3.7736 & 3.7736 & 3.7302 \\
0.99 & 1 & 16.65 & 4.1892 & 4.0301 & 4.0150 & 3.9778 \\
1.25 & 1 & 23.55 & 4.4622 & 4.3266 & 4.2526 & 4.2260 \\
1.50 & 2 & 33.30 & 4.6788 & 4.5635 & 4.5071 & 4.4748 \\
1.75 & 2 & 47.10 & 4.8659 & 4.7680 & 4.7586 & 4.7238 \\
2.00 & 2 & 66.60 & 5.0905 & 5.0075 & 5.0038 & 4.9732 \\
2.25 & 2 & 94.19 & 5.3595 & 5.2892 & 5.2484 & 5.2227 \\
2.50 & 3 & 133.21 & 5.6385 & 5.5768 & 5.5024 & 5.4723 \\
2.75 & 3 & 188.38 & 5.8224 & 5.7722 & 5.7554 & 5.7221 \\
3.00 & 3 & 266.42 & 6.0443 & 6.0019 & 6.0009 & 5.9719 \\
3.75 & 4 & 753.54 & 6.8015 & 6.7754 & 6.7544 & 6.7216 \\
4.00 & 4 & 1065.66 & 7.0219 & 7.0005 & 7.0002 & 6.9716 \\
4.25 & 4 & 1507.07 & 7.2872 & 7.2692 & 7.2470 & 7.2215 \\
4.50 & 5 & 2131.32 & 7.6098 & 7.5850 & 7.5009 & 7.4715 \\
4.75 & 5 & 3014.14 & 7.7912 & 7.7768 & 7.7543 & 7.7215 \\
5.00 & 5 & 4262.64 & 8.0109 & 8.0001 & 8.0001 & 7.9715 \\
5.25 & 5 & 6028.29 & 8.2755 & 8.2665 & 8.2468 & 8.2215 \\
5.50 & 6 & 8525.29 & 8.6052 & 8.5867 & 8.5009 & 8.4715 \\
5.75 & 6 & 12056.60 & 8.7861 & 8.7775 & 8.7542 & 8.7215 \\
6.00 & 6 & 17050.60 & 9.0054 & 9.0000 & 9.0000 & 8.9715 \\
7.00 & 7 & 68202.30 & 10.0027 & 10.0000 & 10.0000 & 9.9715 \\
7.25 & 7 & 96452.60 & 10.2668 & 10.2646 & 10.2468 & 10.2215 \\
7.50 & 7 & 136405.00 & 10.5840 & 10.5821 & 10.5009 & 10.4715 \\
7.75 & 8 & 192905.00 & 10.7823 & 10.7782 & 10.7542 & 10.7215 \\
8.00 & 8 & 272809.00 & 11.0014 & 11.0000 & 11.0000 & 10.9715 \\
\hline
\end{array}\]

Table 5.2: The average length of codes generated from Golomb-Rice coding, GRHC, and Huffman coding when encoding data with Laplacian distribution

\(m^*) are large, the residuals are not well modelled by the Laplacian distribution. Since the Huffman codes are based on an assumed Laplacian distribution, this mismatch
CHAPTER 5. RESIDUAL CODING SCHEME

Figure 5.10: The improvements of GRHC over Golomb-Rice when encoding data source with Laplacian distribution

degraded the performance of Huffman coding. Golomb-Rice coding is more general as it does not make assumptions about the residual distributions, and is thus more suited for such cases.

Consequently, a progressive scheme was thus designed: apply GRHC when \( m \leq 4 \), and apply Golomb-Rice coding when \( m > 4 \). This eliminates the negative effect when a block of data has \( m > 4 \). Table 5.3 shows the compression results of the Golomb-Rice coding, the GRHC scheme, and this progressive scheme when encoding the test dataset. Compared with Golomb-Rice coding, the average improvement of the progressive GRHC is around 0.10%, and the maximum improvement it achieved (on this test dataset) is 0.37%.

Table 5.4 shows the encoding time differences between Golomb-Rice coding and
Figure 5.11: The improvements of GRHC over Golomb-Rice when encoding the residuals output from a linear prediction model.

The progressive GRHC coding, when run on a computer with a 2.80 GHz Intel Pentium 4 processor. The difference in run-time is minor, and in some cases progressive GRHC is faster than Golomb-Rice coding.

In this chapter, we analyzed the performance of the Golomb-Rice coding algorithm applied to the residuals output from the linear prediction model, and designed new schemes, the GRHC coding and the progressive GRHC coding, which can produce better compression results for the sequence of the residuals with low variance. In the next chapter, we will introduce a new idea for the blocking component, which can flexibly block the sequence of samples into a number of variable-length blocks.
CHAPTER 5. RESIDUAL CODING SCHEME

Occurrence probabilities of Rice parameter $m$

Figure 5.12: The occurrence probabilities of Rice parameter $m$

<table>
<thead>
<tr>
<th>File</th>
<th>Golomb-Rice Ratio(%)</th>
<th>GRHC Ratio(%)</th>
<th>GRHC Diff.(%)</th>
<th>Progressive GRHC Ratio(%)</th>
<th>Progressive GRHC Diff.(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>38.34</td>
<td>38.23</td>
<td>0.11</td>
<td>38.23</td>
<td>0.11</td>
</tr>
<tr>
<td>01</td>
<td>42.24</td>
<td>42.17</td>
<td>0.07</td>
<td>42.16</td>
<td>0.08</td>
</tr>
<tr>
<td>02</td>
<td>53.08</td>
<td>53.12</td>
<td>-0.04</td>
<td>53.02</td>
<td>0.06</td>
</tr>
<tr>
<td>03</td>
<td>55.71</td>
<td>55.44</td>
<td>0.27</td>
<td>55.35</td>
<td>0.36</td>
</tr>
<tr>
<td>04</td>
<td>41.53</td>
<td>41.19</td>
<td>0.34</td>
<td>41.16</td>
<td>0.37</td>
</tr>
<tr>
<td>05</td>
<td>72.55</td>
<td>72.61</td>
<td>-0.06</td>
<td>72.53</td>
<td>0.02</td>
</tr>
<tr>
<td>06</td>
<td>68.93</td>
<td>68.97</td>
<td>-0.04</td>
<td>68.93</td>
<td>0.00</td>
</tr>
<tr>
<td>07</td>
<td>60.56</td>
<td>60.62</td>
<td>-0.06</td>
<td>60.55</td>
<td>0.01</td>
</tr>
<tr>
<td>08</td>
<td>70.58</td>
<td>70.65</td>
<td>-0.07</td>
<td>70.56</td>
<td>0.02</td>
</tr>
<tr>
<td>09</td>
<td>55.50</td>
<td>55.52</td>
<td>-0.02</td>
<td>55.46</td>
<td>0.04</td>
</tr>
<tr>
<td>10</td>
<td>51.90</td>
<td>51.94</td>
<td>-0.04</td>
<td>51.87</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Table 5.3: The improvements achieved by applying GRHC coding
### Table 5.4: The encoding time differences between Golomb-Rice coding and Progressive GRHC. CPU time is measured in seconds.

<table>
<thead>
<tr>
<th>File</th>
<th>Golomb-Rice</th>
<th>Progressive GRHC</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>27.195</td>
<td>26.891</td>
</tr>
<tr>
<td>01</td>
<td>87.225</td>
<td>88.492</td>
</tr>
<tr>
<td>02</td>
<td>22.946</td>
<td>23.493</td>
</tr>
<tr>
<td>03</td>
<td>13.086</td>
<td>13.226</td>
</tr>
<tr>
<td>04</td>
<td>41.926</td>
<td>42.280</td>
</tr>
<tr>
<td>05</td>
<td>35.155</td>
<td>35.922</td>
</tr>
<tr>
<td>06</td>
<td>25.616</td>
<td>25.667</td>
</tr>
<tr>
<td>07</td>
<td>48.907</td>
<td>50.424</td>
</tr>
<tr>
<td>08</td>
<td>21.269</td>
<td>21.244</td>
</tr>
<tr>
<td>09</td>
<td>38.923</td>
<td>39.195</td>
</tr>
<tr>
<td>10</td>
<td>46.866</td>
<td>47.458</td>
</tr>
</tbody>
</table>
Chapter 6

Flexible Blocking Scheme

Blocking is a common process used in multimedia compressors. This process is placed before the predictor and residual coding components in the encoding process. It breaks a sequence of samples into contiguous units, each of which is processed relatively independently. The advantage of forming the sequence into chunks is that a compressor can obtain more efficient buffer usage and the predictor can produce better prediction performance. Generally, the block size is specified or defaulted before the encoding. Hence, for each block, the codec could load the given amount of information into memory. Furthermore, a carefully-chosen number of samples can maximize the functionality of the prediction model.

In practice, most lossless audio compression systems apply a constant block size to simplify the process of breakpoint selection. (A breakpoint is defined as the last sample of a block.) The block size for encoding an audio file with a 44.1KHz sampling rate typically is set from 2K to 6K bytes. The block length considerably influences the efficiency of the prediction model and the resulting compression ratios [Rob94, Kim03]. A large block size decreases the prediction capacity of the prediction model; the coefficients \(\{a_i\}\), which are generalized for the whole block, may not capture and represent the variance of the waveform so well. On the other hand, if the block size is too small, the frame overhead, which contains information for decoding, lowers the compression effect.
Consequently, a fixed block size is neither the only nor the optimal choice, motivating an investigation of a flexible blocking scheme. The idea is to adjust breakpoints so as to obtain an optimal compression effect. The additional information to be included in the frame overhead is a block size for each frame, which can be represented as a positive integer value. The algorithm is asymmetric: the breakpoints are computed in the encoding process; the decoder only needs to read the frame overhead and get the block size. There is no substantial difference for the decoding process.

For an optimization problem, there are two terms of interest: local optimum and global optimum. A local optimum is a solution from a given domain, optimal within a neighboring set of solutions. A global optimum is the best solution of the whole solution space. For example, Figure 6.1 illustrates a function $F(X)$ representing results of a solution domain $X$. If the question is to look for the minimum result, $x_1$ would be a local optimum whereas $x_2$ is the global optimum.

![Figure 6.1: Local optimum and global optimum](image)

Generally, the globally-optimum solution for a flexible blocking problem would be a sequence of breakpoints which results in the best compression ratio for the entire file; however, compared with the one-dimension solution optimization problem shown in Figure 6.1, flexible blocking is more complex. First, the solution domain is multi-dimensional, represented as $(p_1, p_2, p_3, \ldots)$, where each dimension $p_i$ is one breakpoint. Moreover, $p_1, p_2, p_3, \ldots$ are inter-dependent: each $p_i$'s selection affects the following ones. To simplify the whole problem, we use a greedy algorithm: start from point $p_{i-1}$, and set a breakpoint $p_i$ which obtains a best compression ratio for the block $(p_{i-1}, p_i)$. The algorithm starts at the beginning of an audio file, and the
last breakpoint should be the last sample. The sequence of blocks is specified in order. Generally, the greedy algorithm can only obtain a local optimum for this problem.

6.1 Methods Applied for Flexible Block Size Scheme

We design two ways to flexibly set breakpoints based upon the properties of audio waveforms. The descriptions and test results are described in the following subsections.

6.1.1 Method 1

Method 1 is designed to break up samples based on the nature of audio waveforms; in other words, it blocks the contiguous samples with the same or nearly the same amplitudes into one unit, instead of separating them into two blocks. A linear prediction model tends to perform well on a block of samples with a consistent waveform. Figure 6.2 presents an audio signal separated in two ways, fixed-sized blocking and variable-sized blocking. It would not be unexpected that the second partition is a better solution than the first partition, because the second partition creates blocks of samples which have a degree of self-similarity.

![Fixed-sized blocking and Variable-sized blocking](image)

Figure 6.2: An audio stream processed by fixed blocking and flexible blocking
There are several issues to be considered during the design of a flexible blocking algorithm:

- Avoiding short-term variability: the breakpoints should not be affected by a short run of samples with different characteristics. For example, the algorithm should tolerate a few “inconsistent” sample values in the middle of a block.

- Fast breakpoint selection: a flexible blocking algorithm should be able to pick a breakpoint reasonably quickly. Sample-by-sample picking is inefficient, and may somehow cause a short-term variability effect.

Method 1 works as follows. We define two blocks, block A and B: block A is the required part of the new block; block B is the one following block A and will be considered for concatenation onto block A. The size of block A is denoted as size_A, and the size of block B is denoted as size_B. Build a prediction model based upon samples of block A. Use this model to predict samples of these two blocks, and get two sequences of errors. If block B is correlated with block A, then there should not be a significant difference between the errors of block A and B. We employ the mean of the squared errors to measure the variances of these two sequences, \( \frac{1}{N} \sum_{i=1}^{N} e_i^2 \). They are denoted err_sq_mean_A and err_sq_mean_B.

A variable rate is defined to measure the correlation between err_sq_mean_A and err_sq_mean_B:

\[
rate = \frac{|err\_sq\_mean\_A - err\_sq\_mean\_B|}{err\_sq\_mean\_A}
\]

The smaller the value of rate, the greater the similarity between block A and B. When err_sq_mean_B is much larger than err_sq_mean_A, it means that block B is quite different from block A. On the other hand, when err_sq_mean_B is much smaller than err_sq_mean_A, then it is possible that there is a more suitable prediction model for block B; for example, the case where block B has small variance. A criterion, denoted as \( R \), is defined to judge the correlation level between these two blocks.

First, define several constants: the initial block size for block A and block B is denoted start_size; the maximum block size is denoted max_size; and the size of an un-separable unit is denoted min_size (that is, the algorithm will not consider
adding or removing less than \textit{min\_size} samples to or from any block). The described experiments were done with \textit{start\_size} = 4K, \textit{max\_size} = 2^{16} and \textit{min\_size} = 32 (\textit{min\_size} has to be a power of two).

Next, calculate the value of \textit{rate} with initial block A and block B. If \textit{rate} < \textit{R}, double the size of both block A and block B, which means the original block B is appended to the original block A. Repeatedly double the size of block A and block B until either the sum of the sizes of block A and B reaches the maximal buffer size, or \textit{rate} is smaller than the judgment criterion \textit{R}, whichever happens first. If the case \textit{rate} < \textit{R} happens, a breakpoint in block B is specified. The scheme for specifying the breakpoint in block B is:

1. Define a variable denoted as \textit{step} equal to the size of block B.

2. A loop starts initially with the \textit{rate} of block A and B. Reduce the size of \textit{step} by half. If \textit{rate} < \textit{R}, increase the size of block B by \textit{step}; otherwise, reduce it by \textit{step}. Update the value of \textit{err\_sq\_mean\_B} and the corresponding \textit{rate} value. Continue looping until the value of the \textit{step} is less than or equal to \textit{min\_size}.

The output block size will be the sum of the sizes of block A and B. A flow chart for the algorithm is shown in Figure 6.3. Figures 6.4 and 6.5 illustrate two cases of the breakpoint selection. Figure 6.4 demonstrates the case where the initial block B (the one chosen in the first step of the algorithm) does not have a consistent prediction result as block A does, and the breakpoint is set in the middle of block B. Conversely, if the \textit{rate} is consistently smaller than the value of \textit{R}, both block A and B are repeatedly doubled. The increasing is stopped when \textit{size}_A + \textit{size}_B reaches the maximum buffer size; such a case is shown in Figure 6.5.

The block size can be increased quickly in the case of a long series of samples consistently well represented by a similar prediction model. The breakpoint is also easily detected and specified when a waveform starts changing. Table 6.1 displays the minimal (best) compression ratios of the flexible blocking Method 1, when the initial \textit{size}_A and \textit{size}_B are set equal to 4K, and the criterion judgment ratio \textit{R} ranges from 0.4 to 1.0. The best compression ratios which can be achieved for each test audio file is shown in bold in the table. The improvement ratio between the fixed blocking
CHAPTER 6. FLEXIBLE BLOCKING SCHEME

Input: \( \text{size}_A, \text{size}_B \)
Output: Output block size

Buffer Block A:
(index_front, index_front + size\(_A\) - 1)
Calculate predictive coefficients
Predict
Residuals
Calculate the mean of squared errors
err\(_{sq\_mean}_A\)
Compute rate

(size\(_A\) + size\(_B\) <= max\(_size\))?
Yes
No

Output block size = size\(_A\) + size\(_B\)

Buffer Block B:
(index_front + size\(_A\), index_front + size\(_A\) + size\(_B\) - 1)
Predict
Residuals
Calculate the mean of squared errors
err\(_{sq\_mean}_B\)

(Step == size\(_A\), && rate < R)?
No
step = step / 2;

size\(_B\) = (rate < R)
(size\(_B\) + step)
(size\(_B\) - step);

(step == min\(_size\))?

Yes

Output block size = size\(_A\) + size\(_B\)

Figure 6.3: The flow chart for flexible blocking (Method 1)
Figure 6.4: Example 1 for flexible blocking (Method 1): the breakpoint is set in middle of the initial block B

Figure 6.5: Example 2 for flexible blocking (Method 1): the breakpoint is specified until the sum of the size block A and block B reaches the maximum buffer size.

algorithm and the flexible blocking Method 1 ranges from 1.48% to 0.23%, and the average is 0.74%. For more detailed information, see Table 6.2.

6.1.2 Method 2

A simpler and more straightforward blocking method is investigated. It can choose breakpoints even more efficiently. In Section 5.2, we gave an expression for the expected length of a frame, denoted $E(*)$. Using this estimation, we design an easier
### Table 6.1: The compression ratio of the codec when applying flexible blocking Method 1 with judgment element R [0.4 : 1.0]

<table>
<thead>
<tr>
<th>File</th>
<th>R</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>37.91</td>
<td>37.91</td>
</tr>
<tr>
<td>01</td>
<td>41.02</td>
<td>40.76</td>
</tr>
<tr>
<td>02</td>
<td>52.23</td>
<td>52.24</td>
</tr>
<tr>
<td>03</td>
<td>55.31</td>
<td>55.28</td>
</tr>
<tr>
<td>04</td>
<td>40.59</td>
<td>40.54</td>
</tr>
<tr>
<td>05</td>
<td>72.53</td>
<td>72.45</td>
</tr>
<tr>
<td>06</td>
<td>68.84</td>
<td>68.87</td>
</tr>
<tr>
<td>07</td>
<td>59.97</td>
<td>59.83</td>
</tr>
<tr>
<td>08</td>
<td>70.45</td>
<td>70.48</td>
</tr>
<tr>
<td>09</td>
<td>54.84</td>
<td>54.73</td>
</tr>
<tr>
<td>10</td>
<td>50.86</td>
<td>50.82</td>
</tr>
</tbody>
</table>

### Table 6.2: Fixed blocking vs. flexible blocking Method 1

<table>
<thead>
<tr>
<th>File</th>
<th>Fixed Blocking</th>
<th>Method 1</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>00.wav</td>
<td>38.34%</td>
<td>37.91%</td>
<td>0.43%</td>
</tr>
<tr>
<td>01.wav</td>
<td>42.24%</td>
<td>40.76%</td>
<td>1.48%</td>
</tr>
<tr>
<td>02.wav</td>
<td>53.04%</td>
<td>52.23%</td>
<td>0.81%</td>
</tr>
<tr>
<td>03.wav</td>
<td>55.72%</td>
<td>55.28%</td>
<td>0.44%</td>
</tr>
<tr>
<td>04.wav</td>
<td>41.53%</td>
<td>40.54%</td>
<td>0.99%</td>
</tr>
<tr>
<td>05.wav</td>
<td>72.89%</td>
<td>72.45%</td>
<td>0.44%</td>
</tr>
<tr>
<td>06.wav</td>
<td>69.07%</td>
<td>68.84%</td>
<td>0.23%</td>
</tr>
<tr>
<td>07.wav</td>
<td>60.56%</td>
<td>59.83%</td>
<td>0.73%</td>
</tr>
<tr>
<td>08.wav</td>
<td>70.77%</td>
<td>70.45%</td>
<td>0.32%</td>
</tr>
<tr>
<td>09.wav</td>
<td>55.50%</td>
<td>54.43%</td>
<td>1.07%</td>
</tr>
<tr>
<td>10.wav</td>
<td>51.90%</td>
<td>50.74%</td>
<td>1.16%</td>
</tr>
</tbody>
</table>

method to optimize the blocking process.

Suppose we have two adjacent blocks, denoted as block A and block B, where point $p$ is the breakpoint separating these two blocks. Without the breakpoint $p$, we would have a larger block comprised of the samples in block A and block B, which we denote as block A&B. If the sum of the lengths of the residuals for block A and
block B is larger than that of block A&B, it means, without the breakpoint \( p \), we may achieve better compression for the same set of data.

Initially, set the size of block A and B equal to \( s \), which could be 1K, 2K or 4K. Define a loop where, at each step, the expected length of the encoded block A, \( E(A) \), the expected length of the following block B, \( E(B) \), and that of the block A&B, \( E(A&B) \) are calculated. If \( E(A) + E(B) < E(A&B) \), block A is set as an optimal block. Otherwise, update block A and B: the new block A is the block combining the original block A and B, and the new block B is a block with size \( s \), following the new block A. Redo the loop until the size of block A reaches the maximum size or the case \( E(A) + E(B) < E(A&B) \) happens. It is a low-complexity algorithm, but still achieves improved blocking performance.

Table 6.3 demonstrates the compression ratio of Method 2 compared with a fixed blocking algorithm. Roughly, when \( s = 1K \) or 2K, the algorithm achieves better compression ratios. The improvement ratio between the fixed blocking algorithm and the flexible blocking Method 2 ranges from 2.05\% to 0.65\%, and the average is 1.17\%.

| File | Fixed Blocking | \begin{tabular}{c|c|c|c|c|c} 
| File     | Fixed Blocking | Method 2 | Min | Difference  \\
|-----------|----------------|----------|-----|-------------
| 00.wav    | 38.34%         | 37.73%   | 37.67% | 37.67% | 0.67%  
| 01.wav    | 42.24%         | 40.46%   | 40.19% | 40.24% | 0.19%  
| 02.wav    | 53.04%         | 52.29%   | 52.15% | 52.15% | 0.89%  
| 03.wav    | 55.72%         | 55.07%   | 55.02% | 55.12% | 0.70%  
| 04.wav    | 41.53%         | 40.66%   | 40.39% | 40.38% | 0.39%  
| 05.wav    | 72.89%         | 72.15%   | 72.23% | 72.41% | 0.74%  
| 06.wav    | 69.07%         | 68.42%   | 68.50% | 68.80% | 0.65%  
| 07.wav    | 60.56%         | 59.05%   | 59.09% | 59.42% | 0.51%  
| 08.wav    | 70.77%         | 70.12%   | 70.19% | 70.33% | 0.65%  
| 09.wav    | 55.50%         | 53.87%   | 53.57% | 53.65% | 1.93%  
| 10.wav    | 51.90%         | 50.19%   | 49.97% | 50.05% | 0.69%  

Table 6.3: Fixed block size vs. flexible blocking Method 2 when initial sizes of block A and B are 1K, 2K, or 4K

Table 6.4 demonstrates the compression levels of three different blocking schemes: fixed blocking, Method 1 and Method 2; in all test cases Method 2 produces slightly
better compression performance than Method 1.

<table>
<thead>
<tr>
<th>File</th>
<th>Fixed Blocking</th>
<th>Method 1</th>
<th>Method 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>00.wav</td>
<td>38.34%</td>
<td>37.91%</td>
<td>37.67%</td>
</tr>
<tr>
<td>01.wav</td>
<td>42.24%</td>
<td>40.76%</td>
<td>40.19%</td>
</tr>
<tr>
<td>02.wav</td>
<td>53.04%</td>
<td>52.23%</td>
<td>52.15%</td>
</tr>
<tr>
<td>03.wav</td>
<td>55.72%</td>
<td>55.28%</td>
<td>55.02%</td>
</tr>
<tr>
<td>04.wav</td>
<td>41.53%</td>
<td>40.54%</td>
<td>40.38%</td>
</tr>
<tr>
<td>05.wav</td>
<td>72.89%</td>
<td>72.45%</td>
<td>72.15%</td>
</tr>
<tr>
<td>06.wav</td>
<td>69.07%</td>
<td>68.84%</td>
<td>68.42%</td>
</tr>
<tr>
<td>07.wav</td>
<td>60.56%</td>
<td>59.83%</td>
<td>59.05%</td>
</tr>
<tr>
<td>08.wav</td>
<td>70.77%</td>
<td>70.45%</td>
<td>70.12%</td>
</tr>
<tr>
<td>09.wav</td>
<td>55.50%</td>
<td>54.44%</td>
<td>53.57%</td>
</tr>
<tr>
<td>10.wav</td>
<td>51.90%</td>
<td>50.74%</td>
<td>49.97%</td>
</tr>
</tbody>
</table>

Table 6.4: Fixed block size vs. flexible blocking Method 1 and Method 2

In this chapter, we designed two flexible blocking schemes, and generally they can achieve an average 1% improvement on the test data. In the next chapter, the performance of the flexible blocking scheme applied to an open-source codec will be investigated.
Chapter 7

FLAC with Flexible Blocking Scheme

In this chapter, the performance of the flexible blocking scheme is investigated by integrating it into a widely-available compression system. The requirements for this system are as follows: it should be an open-source codec, it should perform reasonably quickly both in the encoding and the decoding processes, and it should be widely supported by both hardware and software.

Currently, there are a number of lossless audio compression techniques for professional and consumer applications under continuous development. The compression ratios they can achieve are close; the difference between the worst and the best is around 5%. Apart from the compression ratio, the following elements are involved in the design and evaluation of an audio compressor:

- Compression and decompression speed. To some extent, the processing time and compression ratios are two competing effects on encoding and decoding processes. Ideally, a compressor for audio data is real-time; that is, the time needed to encode and decode is less than the time that it takes to play the multimedia data at its usual speed. Some adaptive algorithms, e.g., adaptive Huffman [Sal07, sec. 2.9] and adaptive linear prediction [Say06, sec. 11.4], sequentially update their model based on previous processing. The adjustment not only slightly slows down the encoding process, but also requires the same
adaptation operation to be done during decoding. Generally, the efficiency of
decoding is more important than encoding, since once a data file is compressed,
it is possible that the decompressing would be done many times.

- Variable compression levels. To meet different requirements, it is convenient
to provide different compression levels. In general, compressors offers at least
three options: "fast", "best" and "default". The default option gives the
performance with reasonably good compression effect, and can be finished in a
medium amount of time (compared to the "fast" and the "best").

- Portability, copyright of the code, operating system support and device software
support are some other elements to be considered.

Among the large number of lossless audio compressors, some compressors stand
out as more widely accepted:

- Shorten is one of the first lossless audio codecs, whose model, described in
Figure 3.1(a), is considered as a basic structure utilized by most lossless audio
compressors. Shorten, developed by T. Robinson in 1994, was initially designed
to compress speech corpora [Rob94]. It supports two forms for modeling an
input signal: a low order linear predictor; and one of four fixed polynomial
predictors. Shorten generally performs well on files with low amplitude and low
frequency samples [Sal07, sec. 7.9].

- FLAC, Free Lossless Audio Codec, is a popular, free and well-maintained loss-
less audio compressor, specially designed to encode integer audio signals. Josh
Coalson, the developer who had the first version of FLAC published on the
sourceforge website in 1991, has fully opened the source code of FLAC to the
public, and also maintains and updates the project. FLAC's format and refer-
ence implementation are public on its website [Coa07]. FLAC stands out for its
real-time decoding speed, error resistance, and stream seek-ability. As a result,
many software players and devices have started supporting FLAC. The source
code, GUI encoding/decoding front-ends, and player plug-ins are available on
its website.
• *Monkey’s Audio* is a lossless audio codec with a good compression performance. Compared with FLAC, it applies a more complex strategy: an adaptive predictor (the prediction value is adjusted depending on the accuracy of the predictor) and an adaptive residual coding scheme (the parameter of Golomb-Rice coding is adjustable). As a result, Monkey’s Audio achieves a slightly better compression ratio than FLAC, but takes more encoding time and more decoding time. It has two limitations that exclude it from being considered: first, it is available only for the Windows operating system, and second, its format and technical documentation are not openly available.

• *MPEG-4 Audio Lossless Coding (ALS)*, a new extension of MPEG-4 audio coding, has been developed by the MPEG audio subgroup since 2002; the final technical specification was issued in July 2005. The basic technology for MPEG-4 ALS was developed by the NUe Group at the Technical University of Berlin. The official website is [http://www.nue.tu-berlin.de/forschung/projekte/lossless/mp4als.html](http://www.nue.tu-berlin.de/forschung/projekte/lossless/mp4als.html).

### 7.1 The Implementation of FLAC with A Flexible Blocking Scheme

Flexible blocking is one scheme which hasn’t been applied by lossless audio compression techniques. To explore the performance of a flexible blocking scheme working on open-source compressors, we investigated some open compressors, mainly focusing on their overall performance, documentation support, as well as feasibility. FLAC is a codec which meets all these requirements.

FLAC is an open-source software package performing lossless compression on audio files. Unlike some audio compressors that support both lossy and lossless compressions, FLAC implements lossless compression only. Its developer and administrator, Josh Coalson, points out that there have been many excellent lossy compressors, so that FLAC is and should stay lossless [Sal07, sec. 7.10]. Several main features, namely its openness, robustness and multi-platform compatibility, make FLAC outstanding:
1. FLAC is asymmetric; it spends less time to decode an audio file than to encode it [Coo07]. FLAC has a reasonably fast encoding speed and outstandingly fast decoding speed.

2. FLAC is an open, free codec. The executable file and source code are available on its website. Developers are encouraged to contribute to the project and supporting software. From a developer's point of view, FLAC has well-maintained documentation: its file format, user API and library API of the current version (1.1.4) are described on its official website. Moreover, aside from the codec itself, there are some tools which make FLAC more usable:
   
   - Metaflac is one tool used to display information of metadata blocks (the metadata block is the overhead defined by FLAC for storing the basic properties of an audio stream).
   - FLAC also provides some testing tools which can be used to verify the correctness of library API functions.

3. FLAC is designed to be seekable and streamable. As described above, the compressed data is saved in a sequence of frames. For each frame, there is a frame header which contains enough information to decode the frame. Therefore, without decoding from the beginning, decoding can be started from any frame. Moreover, to be able to skip any period of audio, FLAC has a list of seek points for indexing samples every few seconds.

4. As announced on its official website, FLAC is the most widely supported lossless audio codec. It provides source code and binaries for Linux, Windows, Mac OSX, *BSD, Solaris, OS/2, BeOS, and Amiga OS. A number of players, such as Winamp and MPlayer, support the FLAC file format or have FLAC plugins. Furthermore, many home stereos, car stereos, portable and handheld audio devices can play music in this format [Coo07].

Unlike many other public lossless audio compressors, the FLAC file format can support flexible blocking. The metadata blocks (see [Coo07] for a complete list) have
two 16-bit integers representing the minimum block size and maximum block size. If they are different, then it means that a variable block size scheme is applied within the compressed “flac” file. Furthermore, each frame header contains the length of the block. It is thus feasible to apply a flexible blocking scheme within FLAC without affecting its decoding process.

FLAC is programmed in C/C++. It offers two sets of API functions: the FLAC C API and the FLAC C++ API. The functions provided by these two sets of libraries are mostly equivalent. The reference for that is listed on sourceforge website: http://flac.sourceforge.net/api/index.html. It gives an overview of FLAC’s structure, and could be a guide for easily finding information.

7.1.1 The Contribution: FLAC with Variable Block Size

In Section 6.1.2, we displayed the comparison of two flexible blocking algorithms. The result shows that the Method 2 can achieve a better compression ratio. To explore the performance of a flexible blocking scheme working on open-source compressors, we added this method into FLAC.

We added one option which encodes a sequence of frame with variable block size; it is named --flexible-blocking. Table 7.1 shows the comparison between the original fixed blocking scheme and the flexible blocking scheme, where the initial sizes of block A and B for Method 2 are 2K, which was experimentally found to be optimal for FLAC. The command lines (for example, to encode 00.wav file) are:

Fixed blocking: flac -8 --no-padding 00.wav -f
Variable blocking: flac -8 --no-padding --flexible-blocking 00.wav -f

The improvement achieved is limited: the average compression ratio improvement on the test set is 0.16%; this results in a saving of 1,083 kilobytes out of 674 megabytes of the original test data size.
### 7.1.2 Extreme Block Size Search

Considering the limited improvement achieved, we designed another algorithm which can discover the globally optimal breakpoint solution set to achieve the best compression ratio of FLAC. We first define the maximum block size and minimum block size; the \( \text{max\_blocksize} \) is \( 2^{16} \), since FLAC stores the block size in 16 bits, and \( \text{min\_blocksize} \) is typically set to a power of two; experiments were performed with \( \text{min\_blocksize} \) set to 512 and 1024. Next, search for the optimal solution, and ultimately compress the audio file with the solution that can generate the smallest encoded file size. The algorithm is computationally very expensive, in both time and space.

Define the value of maximum block size, e.g., 64K samples, and minimum block size, e.g., 512 samples. For instance, if the audio file has 128K audio samples, then we will have \( N = 128K/512 = 256 \) possible breakpoints. Number them from breakpoint 1 to breakpoint 256, and define the beginning of the samples as breakpoint 0; the end of the samples is breakpoint 256. Define a matrix where each item \( (i, j) \) represents the information for a period of samples from breakpoint \( i \) to breakpoint \( j \). For example, \( (0, 1) \) would be a block of samples from \( x_0 \) to \( x_{511} \), when the minimum block size is 512. The size of \( (i, j) \) would be \( (j - i) \times \text{min\_blocksize} \). For each pair \( (i, j) \), we keep...
two numbers: an unsigned integer value representing the expected size of the encoded period of data, denoted as \((i, j).\ bit\_num\), and a major breakpoint \((i, j).\ bp\), which will be explained later.

The matrix is completed by calculating items from its diagonal line up to its rightmost and uppermost item \((0, N)\). The bit number for each item \((i, j)\) would be:

\[
(i, j).\ bit\_num = \begin{cases} 
\min\big(E(block(i, j)), \min_{i < t < j} ((i, t).bit\_num + (t, j).bit\_num)\big), & \text{if } ((j - i) \times \text{min\_blocksize}) \leq \text{max\_blocksize} \\
\min_{i < t < j} ((i, t).bit\_num + (t, j).bit\_num), & \text{otherwise}
\end{cases}
\]

(7.1)

where \(E(block(i, j))\) calculates the size of the encoded block from breakpoint \(i\) to \(j\) without separation. It is computed only if the block size isn’t larger than the maximum allowed size. The value \((i, j).\ bp\) indicates a major breakpoint which may result in an optimal blocking. If its value is equal to \(-1\), then the whole block without separation would be optimal. Otherwise, it would be the value of \(t\) where \((i, t).\ bit\_num + (t, j).\ bit\_num\) is minimized. The optimal solution is found at \((0, N)\) when the algorithm has finished.

By tracing back from \((0, N)\), we can obtain the optimal list of breakpoints. First, create a list to store the breakpoints. Add breakpoint 0 and breakpoint \(N\) into the list. Traverse the list from the beginning to the end. Initially, define a pointer, denoted as \(p\), pointing to the first item of the list. At each step, get the current breakpoint number \(i\), which \(p\) is pointing to, and its next one \(j\). If the matrix item \((i, j).\ bp\) is equal to \(-1\), assign \(p\) to the item \(j\). Otherwise, insert the number of \((i, j).\ bp\) between \(i\) and \(j\). The loop is ended when \(p\) points to the last element in the list.

The matrix shown in Figure 7.1(a) demonstrates an example of the result after analyzing an audio stream with six possible breakpoints. Each item is represented as \((i, j).\ bp\). Using this matrix, we start with a list with two items: 0 and 6. The pointer \(p\) is initially pointing to number 0. Since \((0, 6).\ bp = 1\), insert 1 after 0. Check \((0, 1).\ bp\); since it is \(-1\), assign \(p\) to 1. Check the value of \((1, 6).\ bp\) which is 3, and insert
CHAPTER 7. FLAC WITH FLEXIBLE BLOCKING SCHEME

(a) A matrix calculated from extreme block size search scheme

\[
  \begin{bmatrix}
    (0, 1) : -1 & (0, 2) : 1 & (0, 3) : 1 & (0, 4) : 2 & (0, 5) : 1 & (0, 6) : 1 \\
    (1, 2) : -1 & (1, 3) : -1 & (1, 4) : 3 & (1, 5) : 3 & (1, 6) : 3 \\
    (2, 3) : -1 & (2, 4) : 3 & (2, 5) : 4 & (2, 6) : 3 \\
    (3, 4) : -1 & (3, 5) : -1 & (3, 6) : -1 \\
    (4, 5) : -1 & (4, 6) : 5 \\
    (5, 6) : -1
  \end{bmatrix}
\]

(b) A demonstration of collecting breakpoints

\[
  \begin{array}{c|c|c}
  | & | & | \\
  0 & 6 & (0, 6).bp = 1 \\
  \downarrow & | & | \\
  0 & 1 & 6 & (0, 1).bp = -1 \\
  \downarrow & | & | \\
  0 & 1 & 6 & (1, 6).bp = 3 \\
  \downarrow & | & | \\
  0 & 1 & 3 & 6 & (1, 3).bp = -1 \\
  \downarrow & | & | \\
  0 & 1 & 3 & 6 & (3, 6).bp = -1 \\
  \downarrow & | & | \\
  0 & 1 & 3 & 6 & p \text{ points to the last element}
  \end{array}
\]

Figure 7.1: Extreme block size search example

3 after 1. Keep on going until \( p \) points to the last element in the list. Figure 7.1(b) demonstrates the process of collecting breakpoints. The optimal solution for this example would be breakpoints at locations: \( 1 \times \text{min.blocksize}, 3 \times \text{min.blocksize} \) and \( 6 \times \text{min.blocksize} \).

Figure 7.2 illustrates the breakpoint selection found by the extreme block size search scheme. The vertical lines are the places where breakpoints are set. This example shows that the method can set the breakpoints at the place where the amplitudes of the audio samples change significantly.

Therefore, we may explore the best possible compression ratio that FLAC can achieve by adopting a flexible blocking scheme. The command option to do this
Figure 7.2: An example of the breakpoints selected by the extreme block size search scheme

is named --extreme-blocksize-search. Table 7.2 displays the list of compression ratios achieved by encoding some wave files using FLAC with different schemes, the original fixed blocking scheme (4K audio samples per block), flexible blocking (Method 2), extreme block size search with min.blocksize = 1024, and extreme block size search with min.blocksize = 512. We selected three files, 00.wav (which has the lowest compression ratio of the test set), 05.wav (which has the worst compression ratio of the test set), and 03.wav (which has a medium compression ratio), and compressed the first two minutes of each of these files. The encoding time (on the previously-described computer) of fixed blocking is around 11 seconds, the encoding time of the flexible blocking (Method 2) is around 60 seconds, the encoding time of extreme block search with min.blocksize = 1024 is around 6.5 hours, and the encoding time of extreme block search with min.blocksize = 512 is around 28 hours. The result shows the improvements gained by introducing the variable block size schemes to FLAC is modest, and if we assume the compression ratio of the extreme block size search with min.blocksize = 512 is the best possible compression, then the flexible blocking Method 2 can achieve 57% of the possible improvement with only a relatively small amount of extra running time.

<table>
<thead>
<tr>
<th>File</th>
<th>Fixed Blocking</th>
<th>Flexible Blocking</th>
<th>Extreme Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00†</td>
<td>40.24%</td>
<td>40.09%</td>
<td>40.03%</td>
</tr>
<tr>
<td>03†</td>
<td>58.51%</td>
<td>58.38%</td>
<td>58.29%</td>
</tr>
<tr>
<td>05†</td>
<td>68.92%</td>
<td>68.72%</td>
<td>68.61%</td>
</tr>
</tbody>
</table>

Table 7.2: The comparison of compression ratios achieved by using different encoding schemes; † the first two minutes of the corresponding file
The reason for the small improvements of the flexible blocking scheme within FLAC is due to a strategy applied by FLAC. In FLAC, the residual sequence of a block is broken into several partitions, each of which has an individual Rice parameter. The number of partitions is set to $2^m$, where $m$ is optimized by FLAC. This strategy reduces the improvement effect achieved by the optimized block size scheme.
Chapter 8

Conclusions and Future Work

8.1 Conclusions

This thesis provides an overview of a "prediction/residual" structure applied to lossless audio data compression techniques, as well as the design of new techniques to enhance their compression performance. Two main ideas were proposed:

- A progressive entropy coding, Golomb-Rice combined with Huffman coding (GRHC) was designed. Compared with the Golomb-Rice coding, GRHC offers more flexibility, and, theoretically, it performs 33.5% closer to the entropy of the residuals. When the variance of the residuals is small ($m \leq 4$), GRHC performs better than Golomb-Rice coding.

- A new idea of flexibly blocking the sequence of samples into a number of variable-length blocks. Instead of applying the regular fixed-length blocking, it is possible to obtain a better compression ratio by varying the block length. There are two methods designed to implement this idea, and, generally they can achieve an average 1% improvement when applied to the basic "prediction/residual" structure. We also investigated the improvement by adding this scheme into an open-source codec, FLAC. By trying an extreme search for a globally optimal breakpoint solution, we conclude that the flexible blocking scheme can achieve a better compression ratio, but the improvement is somewhat minor.
8.2 Future Work

In this thesis, we researched two main areas of the “prediction/residual” structure: the flexible blocking scheme and residual coding algorithms. There is one more idea for the flexible blocking scheme: in Chapter 7, we displayed the improvement achieved by the extreme block size search scheme with \( \text{min. blocksize} = 1024 \) and 512. It is conceivable (but unlikely) that when \( \text{min. blocksize} \) is smaller, e.g., 1 or 2, the extreme block size scheme can obtain a significantly better improvement ratio. It would be interesting to figure out what that result would be. With the current algorithm, using such a small \( \text{min. blocksize} \) is computationally infeasible.

Instead of the regular entropy coding algorithms, how does the adaptive algorithm perform on the residuals output from prediction models? As mentioned previously, some compression systems, such as Monkey’s Audio, employ these adaptive algorithms; there may be profitable lines of enquiry in this direction.

As previously discussed, the performance of the flexible blocking scheme and residual coding algorithms is approaching the best possible result. It may be worth investigating the other component, the prediction model. The linear prediction model may not be sufficiently accurate to estimate audio samples. Higher order polynomial functions and/or exponential functions may be worth considering.
Bibliography


BIBLIOGRAPHY


