Low Complexity Text and Image Compression for Wireless Devices and Sensors

vorgelegt von
Dipl.-Ing. Stephan Rein
aus Bergisch Gladbach

Von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr. -Ing. –

genehmigte Dissertation

Gutachter:	Prof. Dr.-Ing.	Clemens Gühmann
	Prof. Dr.-Ing.	Thomas Sikora
	Prof. Dr.-Ing.	Peter Eisert
Vorsitzende:	Prof. Anja Feldmann, Ph.D.	

Tag der wissenschaftlichen Aussprache: 27.01.2010

Berlin 2010
D 83

Acknowledgments

I would like to thank Prof. Clemens Gühmann for the supervision of this thesis. I am grateful for the contributions of Stephan Lehmann, who redesigned the sensor hardware and developed a sensor filesystem. Yang Liu solved the wireless interface problems and programmed a basic communication protocol. I am grateful to Prof. Frank Fitzek, who had the idea of the smsZipper. Dr. Stefan Lachmann helped me with proofreading. Last I would like to thank my parents Steliana and Joseph for their encouraging support to pursue this thesis.

Abstract

The primary intention in data compression has been for decades to improve the compression performance, while more computational requirements were accepted due to the evolving computer hardware. In the recent past, however, the attributes to data compression techniques have changed. Emerging mobile devices and wireless sensors require algorithms that get along with very limited computational power and memory.

The first part of this thesis introduces a low-complexity compression technique for short messages in the range of 10 to 400 characters. It combines the principles of statistical context modeling with a novel scalable data model. The proposed scheme can cut the size of such a message in half while it only requires 32 kByte of RAM. Furthermore it is evaluated to account for battery savings on mobile phones.

The second part of this thesis concerns a low-complexity wavelet compression technique for pictures. The technique consists of a novel computational scheme for the picture wavelet transform, i.e., the fractional wavelet filter, and the introduced wavelet image two-line (Wi2l) coder, both having extremely little memory requirements: For compression of a 256x256x8 picture only 1.5 kBytes of RAM are needed, while the algorithms get along with 16 bit integer calculations. The technique is evaluated on a small microchip with a total RAM size of 2 kBytes, but is yet competitive to current JPEG2000 implementations that run on personal computers. Typical low-cost sensor networks can thus employ state-of-the-art image compression by a software update.

Contents

Chapter 1

Introduction

In the ancient Rome, symbols were listed in an *additive* manner to represent numbers. Using the Roman numerals, the number 9 is written as VIIII=5+1+1+1+1=9. Another form of number representation is called *subtractive notation*. In that notation the position of the symbols is of importance. If a smaller numeral precedes a larger one, it has to be subtracted. The number 9 would thus be represented as IX=-1+10=9. IX can be regarded as a shorthand of VIIII. Benefits of a shorthand may be that it takes less time to write it down, that it allows to write more numbers on a piece of paper, and that it even allows the reader to interpret the number faster.

The desire of storing large amounts of information and not to wait for retrieving the information is ongoing. Today information is transformed using its digital representation, and efficiency is improved by *data compression* techniques. These techniques reduce the size of required binary data to store or to transfer the information, while the information itself is preserved. Table 1.1 lists some devices of daily use that employ data compression together with the corresponding standards, partially set by the Moving Picture Experts Group (MPEG). For all types of data, including text, speech, audio, picture, and video, very efficient compression techniques are available.

In the recent past, mobile and wireless devices, e.g., phones or small sensors, evolved and became widely accepted. For these devices there exist requirements aside from efficiency, including power consumption, scalability, low complexity, and low memory. These attributes are addressed in the two parts of this work while novel compression techniques for text (chapter 2) and images (chapter 3) are introduced.

The first part of this work concerns the compression of short text messages. Short messages are sequences of 10 to 400 bytes, which can be exchanged in cellular communications using the short message service (SMS). The mobile phone here typically conveys the data without using a compression technique. A compression technique, however, may allow the user to transfer more data at the same cost. Second, the cellular network may be relieved. This is of special interest when large numbers of messages have to be transferred in time, as it for example is required for emergency message alert services. Recent studies [1] have revealed that in case of emergency alerts, the networks fail to meet the 10 minute alert goal due to congestion of the network. A compression technique may allow the mobile phone users to receive the emergency message in time and thus to save many lives. The objective of chapter 2 is to develop a technique for compression of short messages to be applicable to mobile phones.

The standard techniques for text compression are designed for usage on personal computers (PC) and are not able to efficiently compress a short message, which is demonstrated in table

	mobile phone	mobile music player	camera	cam-corder	DVD-player, recorder	digital TV	PC transfer, play	PC archive
text							LZ77 (zip)	LZ77 (zip)
speech	GSM voice						Skype codec	
audio		MPEG1 audio layer 3		MPEG4	MPEG2	MPEG2, MPEG4	MPEG1 audio layer 3	MPEG1 audio layer 3
picture			JPEG				JPEG	JPEG
video				MPEG4	MPEG2	MPEG2, MPEG4	MPEG4	

Table 1.1: Established compression standards for different kinds of data and electronic devices. Data compression is a key technology for many modern applications, e.g., cellular communication would not be possible at all without efficient speech coding.

Thanks for the nice plot. I will meet Morten soon and report back how large the compression gain of this SMS will be. Would be nice if you could report the gain in dependency on the order.

a) message (191 bytes)

technique	[bytes]
gzip	163
PPMII	149
PPMN	140
PPMZ2	129
own coder	**89**

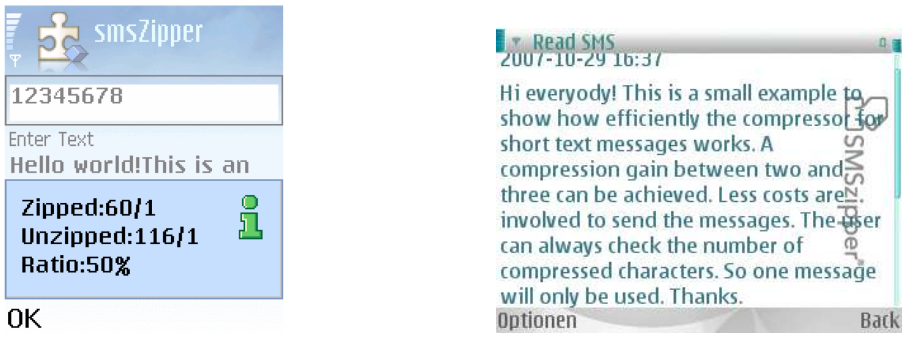
b) compression

Table 1.2: Compression results for the short message in table a). Table b) gives the compressed file sizes in bytes for the programs *gzip* (version 1.3.5), *PPMII* from D. Shakarin (variant I, April 2002), *PPMN* from M. Smirnov (2002, version 1.00b1), *PPMZ2* from C. Bloom (version 0,81, May 2004), and the own coder. Standard lossless compression techniques are designed to compress long files on PCs.

1.2 for a given message. The reasons for this and possible modifications are discussed as follows.

The widely employed tool GNU zip software (gzip) belongs to the class of dictionary coders, where a match between a part of the text and the internal dictionary is substituted by a reference to the dictionary entry, see [2] for details. When the coding process starts, the dictionary is empty, and thus no compression is achieved. To fix this, the internal dictionary would have to be filled beforehand with data. As the dictionary is generally very small, it may not allow for compression of a short message. On the other hand, enlarging the dictionary would enlarge the search time - a critical attribute for limited platforms. Furthermore, the so called *sliding-window* technique is more than 30 years old and does not give state-of-the-art compression, even if it is still in use. The principle thus does not seem to be a promising candidate for short message compression.

The other compression tools in table 1.2 use the method *prediction by partial matching* (PPM). They belong to the class of *statistical* coders, where a statistical model is built *adaptively* throughout the compression. There each symbol is coded separately taking its previous symbols into account. The probability of a symbol is retrieved through the model and encoded using an arithmetic coder. Such a method is not so adaptive than it is with the dic-



a) b)

Figure 1.1: Screen shots of the SMSzipper software. Figure a) illustrates the own cell phone software, which uses the compression method introduced in the first part of this work. Thousands of cell phone users downloaded the tool to save costs when writing messages with more than 160 characters. Today the free software is no longer available, and a commercial software employs the method, illustrated in figure b).

tionary coders, because the model stores all occurrences of symbols. However, as the model is empty at the start of the routine, short messages are not effectively compressed either.

The PPM technique generally gives better compression than the dictionary coders and seems to be conceptually more suitable for the compression of short messages. It is thus selected as a starting point for the first part of this work in chapter 2. A detailed overview of this chapter is given in section 2.1. The chapter describes the steps that were taken for the development of a low-complexity compression scheme. It starts with an own implementation of the method PPM. The implementation is then extended to build a tool which allows for a detailed analysis of the statistical data. The findings lead to the design of a novel statistical context model, which is preloaded with data and evaluated to compress very short messages. A model that only requires 32 kByte cuts the short messages in half. The scalability feature allows for better compression efficiency with the cost of more memory requirements. The novel method is finally applied to a cell phone application, which allows the user to save costs when writing messages that exceed the 160 character limit. Figure 1.1 a) depicts a screen shot of the own software, which was made freely available. Today a commercial tool uses the method, which is illustrated in figure b).

Whereas the first part of this work concerns lossless compression, the second part introduces a novel system for wavelet based lossy image compression on limited platforms. Such a platform can be a wireless sensor in a network. Figure 1.2 illustrates an example of a sensor network in space. These networks consist of very small *pico-satellites*, which are employed for earth observation, weather prediction, and atmospheric monitoring. The satellites are smaller than 10x10x10 cm and only employ very limited hardware. The intention of the considerations in chapter 3 is thus to introduce a low-complexity image compression technique to allow for more effective image data transmission from pico-satellites or small camera sensors. Such a system needs to perform a line-based wavelet transform and to apply an appropriate image coding technique both with respect to the little memory resources of a low-cost microcontroller.

An extensive overview of chapter 3 is detailed in section 3.1. The short outline is given as follows. As a first step an own sensor network platform is designed in order to understand the features of sensor networks and to allow for future verifications of novel algorithms. The platform employs a low-cost signal controller to conform with the typical limitations of sensor networks, as they are detailed in [4]. Figure 1.3 illustrates the own prototype that is employed

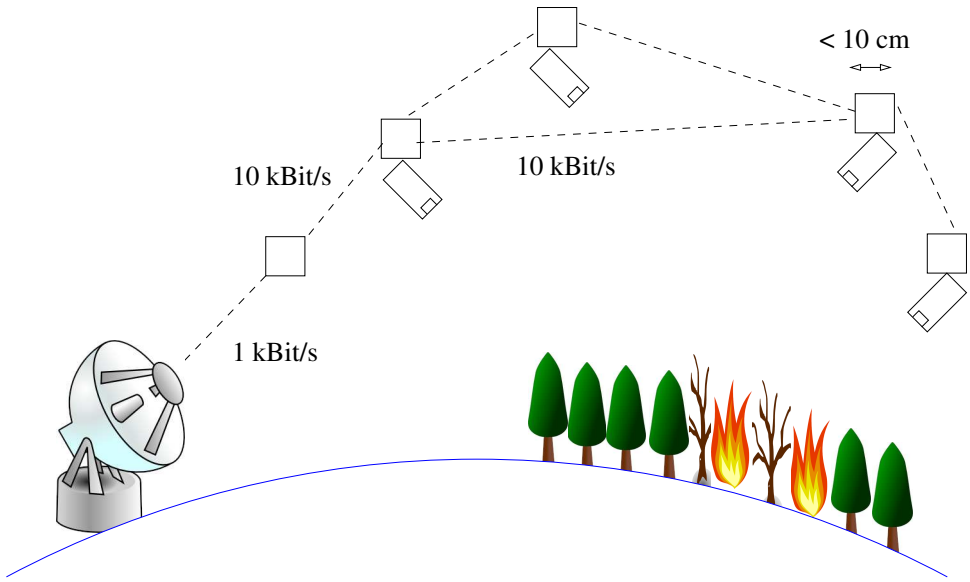


Figure 1.2: Pico-Satellites in space building a camera sensor network for earth observation. These satellites are smaller than 10x10x10 cm and have a weight lower than 1 kg. They are much cheaper than conventional satellites and are currently employed in many research projects worldwide, see for instance the UWE-project of the Universität Würzburg [3]. The low-complexity wavelet transform and image coder introduced in this work may be a candidate for such a satellite to allow for effective picture transmission over a very limited link.

in a wireless network for the distributed computation of a matrix product – a primary step to estimate the feasibility of the upcoming signal processing computations. Then the principles for the novel wavelet transform to be introduced are given. The basics concern the computation of the picture wavelet transform with fixed-point numbers. The problem of the transform is its large memory requirements – a reason why wavelet techniques are generally only employed on more complex and expensive hardware, as for instance, on a digital signal processor (DSP). The fixed-point arithmetic is necessary to compute the wavelet transform with real numbers using only 16-bit integers. The integer numbers are obligate on the here considered 16-bit processors to allow for fast computations.

Then the *fractional wavelet filter* is introduced - a novel algorithm to compute the picture wavelet transform. The algorithm only requires 1.2 kByte of memory for transforming a picture of 256x256 pixels - a novelty in the field of low-memory picture wavelet transform. Figure 1.4 illustrates the result of a typical transform for two levels computed on the own sensor network platform. The picture to be transformed is stored on a multimedia card, from which the data is read line by line. A library for the data access is developed with the help of student work in [5].

The wavelet transform itself does not yet compress the image data. The patterns in the transform have to be exploited by a specific wavelet coding technique. The second part of chapter 3 gives an introduction on wavelet image coding with the standard technique *set partitioning in hierarchical trees* (Spiht) and its basics. The introduction ends with the review of the coding technique *backward coding of wavelet trees* (Bcwt) [6], which is a very recent approach for low-memory wavelet coding. This technique is selected as a starting point for the own investigations - with the aim to develop an algorithm with same compression efficiency while the memory requirements shall be reduced. The own implementation of Bcwt is verified

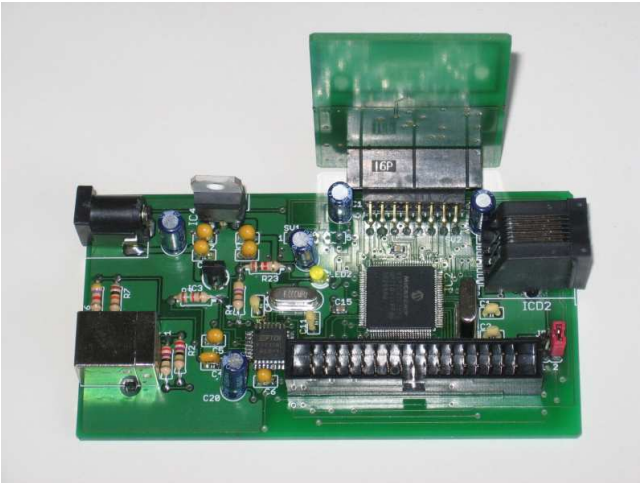
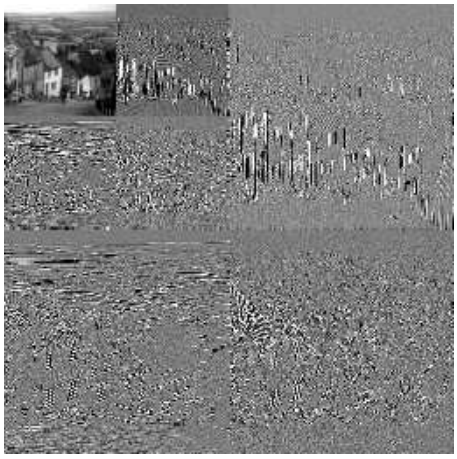


Figure 1.3: First wireless sensor developed in this thesis to conduct elementary signal processing computations in a wireless network. The platform was later extended with a digital camera and a multimedia card (MMC) to allow for the evaluation of a novel picture compression algorithm.

a)



b)

Figure 1.4: Wavelet transform computed on the own sensor hardware with the introduced scheme *fractional wavelet*. Figure a) shows the original image, and figure b) the two-level transform of this image. The scheme requires less than 1.2 kByte of memory and makes the transform applicable to low-cost microcontrollers. The transformed picture is further processed by the wavelet image two-line coder (Wi2l), which is finally introduced in this work.

a) original b) reconstructed

Figure 1.5: Original image (figure a)) and reconstructed image (figure b)) using the novel wavelet image two-line coder (Wi2l). The recursive algorithm only reads two lines of a wavelet subband to encode a 256x256 picture with 1.5 kByte of memory. As the Wi2l coder employs the introduced fractional filter to compute the wavelet transform, all steps can be performed on a microcontroller with integer calculations. In the past these platforms were considered to be not sufficient for wavelet compression. The system further gives state-of-the-art image compression. It here achieves 34.79 dB for a bitrate of 0.7374 bits per byte (bpb), the same quality than Spiht. (The JPEG codec for instance only achieves 31.815 dB for this example.) The Spiht codec was designed for PCs, whereas the novel Wi2l coder runs on a small microchip.

to give the same results than Spiht.

The outcome of these investigations is finally a completely novel recursive algorithm - the *wavelet image two-line* (Wi2l) coder. The novelty of this coder is that it only requires memory for two lines of a wavelet subband to compress an image, which gives a requirement of 1.5 kByte for a 256x256 picture. Compression rates are the same as with Bcwt and Spiht. Figure 1.5 illustrates the quality of a picture that was coded and decoded by Wi2l on the own sensor. For the wavelet transform the fractional filter is employed. Thus all computations for the image compression can be performed with 16-bit integer calculations. The Wi2l coder is finally evaluated on the given hardware to compress a picture in approximatively 2 seconds.

Chapter 2

Text Compression

2.1 Chapter Overview

For the development of a low-complexity compression scheme the method *prediction by partial matching* (PPM) was selected as a starting point, see chapter 1. In section 2.2 literature related to PPM is reviewed. There exists a lot of work that aims to reduce the memory requirements of PPM, however, a system for short messages with extremely low memory requirements was not yet proposed.

The next two sections concern an introduction to the selected lossless text compression technique and give information on the developed software. More precisely, in section 2.3 the technique of arithmetic coding is surveyed and implemented using integers. Arithmetic coding is needed in PPM to encode the symbols with their entropy. Then in section 2.4 the concept of context modeling for data compression is surveyed. Only a few details on the developed software are given and the interested reader is referred to a technical report. The key point of the software is the design of a data structure for a context model using a hash table technique. The own PPM software is verified to perform correctly on a PC, where the memory is not limited.

In section 2.5 the developed software is extended to a tool that allows for an analysis of the gathered statistics and the problems that arise through the design of the data model. A series of measurements that inspect the internal state of the context model is conducted and the required memory is estimated for the given training files.

The findings of these measurements allow for the development of the low-complexity coder in section 2.6, which fulfills the requirements of allocating less than 100 kByte of memory while the compression is better than 3.5 bits per byte (bpb). The novel coder for short messages is evaluated to give promising compression while the memory requirements are very low. It is finally applied to a cell phone application in section 2.7. In the last section the chapter is summarized.

2.2 Related Work

The idea of this work is to develop a technique related to PPM that gives good compression for short messages while it is conceptually simple and has low memory requirements. The work on PPM is here categorized into two classes. The first class concerns the reduction of memory requirements while the compression performance may be improved, and the second

class addresses the compression of short messages. In the following, work that relates to the first class is listed.

In [7], the scheme PPM* achieves superior compression performance over PPMC by exploiting longer contexts. In [8], a technique to delete nodes from the data tree without loss of compression performance is detailed. The study in [9] presents an algorithm for improving the compression performance of PPMII (based on the Shkarin implementation) using a string matching technique on variable length order contexts with the cost of additional compression or decompression time. The method described in [10] provides comparable performance as PPMC by only using 20 % of the internal memory. The method can use orders 3,1,0, and -1 by allocating 100 kBytes of memory. (The model order refers to the size of the statistical model and will be explained in section 2.4.) In [11] an order 1 context model for PPMC is simulated by a hardware model. In [12], the scheme PPM with *information inheritance* is described, which improves the efficiency of PPM in that it gives compression rates of 2.7 bits per byte (bpb) with a memory size of 0.6 MBytes. It is shown that the proposed method has lower requirements than ZIP, BZIP2 (Julian Seward), PPMZ (Charles Bloom), and PPMD (Dmitri Shkarin), in that it is verified that these methods require 0.5 to more than 100 MBytes of RAM.

While these works simplify PPM to make it applicable to many more computer platforms and partially improve the compression performance, short message compression and the issue of using not more than 100 kBytes are not yet solved. The second class of literature addresses this problem and is reviewed as follows.

In [13], an optimal statistical model (SAMC) is adaptively constructed from a short text message and transmitted to the decoder. As the statistical model is present from the first byte that is to be compressed, the method can compress short messages. However, the overall compression ratio would suffer from the additional size of the statistical model, which has to be transferred in advance. Thus the compression rates would not be satisfying. Works especially concerning the compression of short files are given in [14] and [15]. In [14], a tree machine as a static context model is employed. It is shown that zip (Info-ZIP 2.3), bzip-2 (Julian Seward version 1.0.2), rar (E. Roshal, version 3.20 with PPMII), and paq6 (M. Mahoney) fail for short messages (compression starts for files larger than 1000 bytes). The employed model is organized as a tree and allocates 500 kBytes of memory. While the problem of short message compression is addressed, the memory requirements are still too high. The paper in [15] uses syllables for compression of short text files larger than 3 kBytes. These files are too long as to be considered as short messages.

Data compression techniques for sensor networks are surveyed in [16]. Most of the sensor data compression techniques exploit statistical correlations between the typically larger data flows of multiple sensors as they all are assumed to observe the same phenomenon, see for instance [17, 18]. A technique that could for instance compress sensor signaling data, which can be considered as a short text message, is not addressed in these references.

To our best knowledge, the problem of lossless short message compression using a low-complexity technique is not yet addressed in the literature. The idea of this work is thus to develop a technique for lossless compression of text files with a range of size from 50 to 300 bytes. The memory requirements should be scalable and much lower than 100 kBytes. As a starting point an own PPM coder is implemented. PPM is selected because short message compression requires a preloaded statistical context model that holds true for a wide range of text data. In the next sections background information on PPM and our implementation is given. As *arithmetic coding* is a key component of PPM, it is explained first in the next section.

2.3 Arithmetic Coding

2.3.1 Introduction

Arithmetic Coding is a way to efficiently encode symbols without loss by using their known probability. A symbol can be an alphabetic character, a word, or a number. While *Huffman* coding encodes symbols by integer number of bits, arithmetic coding can achieve encoding with no more bits than the actual entropy of the symbol, which can be a fraction. One of the first practical descriptions of arithmetic coding was introduced by J. Rissanen and G. Langdon in [19].

The outline of this introduction on arithmetic coding is specified as follows: In the next subsection some references for further reading are given. In subsection 2.3.3 the principle of arithmetic coding is explained. If a statistical model of the symbols' probabilities or counts is known, the symbols can be coded by successively defining an interval between a low and a high value. In subsection 2.3.4 the implementation of arithmetic coding is addressed. The source code and more details on the implementation in C++ are listed in the report in [20]. The verification of the programmed arithmetic coder with an order 0 model and the data files of the Calgary corpus is performed in subsection 2.3.5. In subsection 2.3.6 the introduction to arithmetic coding is summarized.

2.3.2 Related References

More details on arithmetic coding can be found in [21] [22] [23] [24] [25] [26]. For the survey in this thesis the seminar in [27] and the article in [23] were mainly employed.

A useful alternative to the source code of this thesis might be the *range coder* [28], as it provides similar results on the entropy but is much faster than the here detailed method. A follow-up work that addresses the range coder on a wireless sensor is reviewed in section 2.8 as a link to the picture compression chapter.

2.3.3 Principle of Arithmetic Coding

The smallest number of bits to encode a symbol s_1 is given by the *entropy* $H(s_1) = -\log_2 p(s_1)$ of the symbol, where $p(s_1)$ is the probability of s_1. Thus, the entropy of a sequence of n symbols s_i is calculated as

$$H = \sum_{i=1}^{n} -p(s_i) \log_2 p(s_i). \tag{2.1}$$

H can be a fraction, and to achieve optimal compression, it is necessary to code the symbols with their exact entropy. This may not be possible with *Huffman* codes, where each symbol is coded separately by an integer number of bits. Encoder and decoder both employ the same probability model, which are here assumed to be given. With arithmetic coding, the symbols are coded by defining subintervals within the current *encoder interval*, denoted by the values *low* and *high*. Each symbol is assigned an interval between *low* and *high* according to its probability. The principle is given as follows:

1. Initiate the encoder interval (the values *low* and *high*) with $[0, 1)$

symbol s_i	h	e	l	o
count	1	1	2	1
$p(s_i)$	1/5	1/5	2/5	1/5
$[p_{left}, p_{right})$	[0,1/5)	[1/5,2/5)	[2/5,4/5)	[4/5,1)
$[SymbolLeft, SymbolRight)$	[0,1)	[1,2)	[2,4)	[4,5)

Table 2.1: Symbol statistics for the message *hello*. The first line gives the counts of the single characters $s_i, i = 1 \ldots 4$. The second line gives the probabilities $p(i)$ of the single symbols s_i. The third line arranges these probabilities on a cumulative probability line, where each symbol has a left and a right probability p_{left} and p_{right}. The last line gives the integer cumulative probabilities *SymbolLeft* and *SymbolRight* for each symbol, which are employed in subsection 2.3.4.

s_i	count	$p(s_i)$	symbol interval	encoder interval
				[0,1)
h	1	1/5	[0,1/5)	[0,1/5)
e	1	1/5	[1/5,2/5)	[1/25,2/25)
l	2	2/5	[2/5,4/5)	[7/125,9/125)
l	2	2/5	[2/5,4/5)	[39/625,43/625)
o	1	1/5	[4/5,1)	[211/3125,43/625)

Table 2.2: Principle of Arithmetic Coding for the message *hello*. The limits of the symbol's interval are called the symbol's *left* and *right probability*.

2. Define the next current interval as a subinterval of the previous interval in dependence of the probability of the symbol to be coded, see table 2.4 a) for the notation:

$$\begin{aligned} range &= high - low \\ high &= low + range \cdot p_{right}(s_i) \\ low &= low + range \cdot p_{left}(s_i) \end{aligned} \tag{2.2}$$

3. Go to step 2 if symbols are left, otherwise go to step 4

4. Estimate the minimum number of bits to clearly define a number between *low* and *high*

To be more precise in step 4, *low* and *high* are converted to binary numbers and then the shortest binary number between these two binary numbers is estimated. This binary is here called the *final number*, which is passed to the decoder. From the explanation in step 4 pseudo code can be derived. This problem will be addressed in subsection 2.3.4 using integer arithmetic, thus solving the precision problem.

Table 2.2 shows an example for encoding the message *hello* using the cumulative probabilities given in table 2.1. Look for instance at the symbol e, whose corresponding encoder interval is calculated as [0+(1/5-0)·1/5, 0+(1/5-0)·2/5). The final interval is given as [211/3125=0.06752, 43/625=0.0688). The final interval's length corresponds to the product probability of the single symbols. Now the shortest binary number between these two numbers has to be estimated. It is found as $2^{-4} + 2^{-8} + 2^{-9} = 0.068359375$, which is 0.000100011 in binary notation.

For decoding, a decoder number referring to the symbol interval is used, which is here just called *number* or *decoder number*. The decoder number is initiated with the final number and then the following loop is performed:

number	interval p_{left}, p_{right}	symbol
0.068359375	0,1/5	h
0.341796875	1/5,2/5	e
0.708984375	2/5,4/5	l
0.7724609375	2/5,4/5	l
0.93115234375	4/5,1	o

Table 2.3: Decoding of the compressed message *hello*. The number is updated with equation 2.3.

1. Find the symbol interval (table 2.1) in which the number is located:
   $p_{left}(s_i) < number < p_{right}(s_i)$
   The resulting interval denotes the decoded symbol s_i.

2. Update the decoder number using the current symbol probabilities:

$$number = number - p_{left} \tag{2.3}$$
$$number = \frac{number}{p_{right} - p_{left}}$$

3. Go to step 1 until all symbols are decoded

Thereby the number is scaled from a range between *low* and *high* to a range between 0 and 1. In [27] and [25], the symbol probabilities are scaled instead of updating the final number. In this work it was though found more effective to simply update one number, as done in [26], instead of updating the whole array of symbol probabilities. The decoding process of the message is given in table 2.3.

2.3.4 Programming Arithmetic Coding

In practice, arithmetic with integers instead of floats makes sense as programming is simplified and precision problems do not emerge. Furthermore, many micro-computers do not support fast floating-point calculations.

Even though arithmetic coding was already discovered at the end of the seventies, it did not become popular until the invention of specific computation schemes, including a so-called *scaling procedure*, which is a method of incremental output [24]. This method puts out single bits in advance thus preventing the encoder interval to become too small. The procedure is not described in this thesis but in the technical report in [20].

The section first explains how encoding (subsection 2.3.4.1) and decoding (subsection 2.3.4.2) can be realized with integers. Then in subsection 2.3.4.3 the employed statistical model is explained.

2.3.4.1 Encoding with Integers

For implementing arithmetic coding, integer numbers can be used to store the endpoints of the intervals. Instead of using intervals between 0 and 1, an interval between 0 and $MaxNumber = 128$ can be employed, as illustrated in figure 2.2. Instead of the symbol probabilities p_{left} and p_{right}, the symbol counts *SymbolLeft* and *SymbolRight* are defined. Table 2.4 b) gives the

a) Coding with floats

high, *low*	floating point interval for a symbol
number	a) the number employed for decoding a message b) the number which is constructed to be between *low* and *high* of the final interval
p_{left}, p_{right}	left and right floating point probability of one symbol
range	$= high - low$

b) Coding with integers

high, *low*	integer arithmetic interval
count	a) the decoded count which is within the symbol interval b) employed to count scaling type III operations
SymbolLeft, *SymbolRight*	left and right count of a symbol
RANGE	$= [0 \ldots MaxNumber\,]$, where *MaxNumber* denotes the maximum possible number of the calculator
Half	$Range/2$
SymbolIndex	denotes a symbol as a number $\in [0 \ldots 255]$ instead of $\in [-128 \ldots 127]$ (ASCII)
total	cumulative total count of all symbols

Table 2.4: Notation of all important variables for floating-point coding (table a)) and integer coding (table b)).

notation of all important variables. For estimating the endpoints of the intervals, the integer cumulative probabilities given in table 2.1 are employed. For estimating the subinterval, similar equations than for floating point arithmetic can be derived:

$$\begin{aligned} step &= \frac{high - low}{total} \\ high &= low + \lfloor step \cdot SymbolRight \rfloor \\ low &= low + \lfloor step \cdot SymbolLeft \rfloor \end{aligned} \tag{2.4}$$

These actions are taken when a single symbol is to be coded. *high* and *low* are initialized with 0 and *MaxNumber*. The symbols $\lfloor . \rfloor$ round to the nearest integer lower or equal the included element. Note that *step* may still be a floating point number. The integer intervals are thus estimated by rounding to the nearest integer lower than the floating point endpoints. A number within the last interval (denoted by *low* and *high*) is passed to the decoder. The steps for integer arithmetic encoding are given as follows:

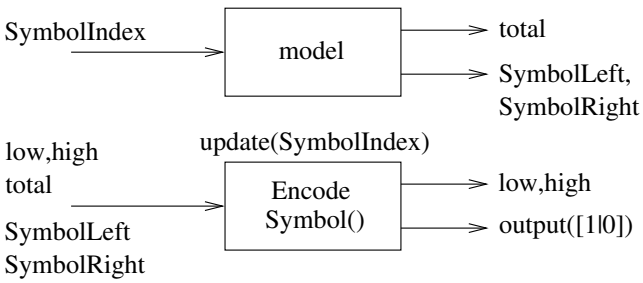


Figure 2.1: Encoding functionality: The two main parts of the arithmetic encoder are a probability model (upper figure) and a function to encode a single symbol (lower figure). The model takes a symbol index as an input variable. It returns a total count, the left probability, and the right probability. The function *EncodeSymbol()* takes the output variables of the model as an input. It updates *low* and *high* of the arithmetic coder and writes the binary compressed data stream.

1. Init variables:
   $low = 0$
   $high = RANGE$
2. Get *SymbolLeft* and *SymbolRight* for the current symbol from the probability model
3. Update the probability model
4. Encode the Symbol:
   (a) $step = \frac{high - low}{total}$
   (b) $high = low + \lfloor step \cdot SymbolRight \rfloor$
   (c) $low = low + \lfloor step \cdot SymbolLeft \rfloor$
   (d) Output binary sequence using a *scaling* procedure
5. Go to step 2.

The step 4.(d) is detailed in [20]. The input and output variables for encoding a symbol and retrieving statistics from the probability model are depicted in figure 2.1.

2.3.4.2 Decoding with Integers

Similarly than for encoding, decoding a symbol requires updating the variables *low* and *high* as follows:

$$\begin{aligned} step &= \frac{high - low}{total} \\ count &= \frac{number - low}{step} \\ [SymbolLeft, SymbolRight] &= GetSymbol(count) \\ high &= low + \lfloor step \cdot SymbolRight \rfloor \\ low &= low + \lfloor step \cdot SymbolLeft \rfloor \end{aligned} \tag{2.5}$$

low and *high* are again initiated with [0,*MaxNumber*). With table 2.1, the symbol decoding function can check the interval corresponding to *count*, and thereby retrieves the symbol. The

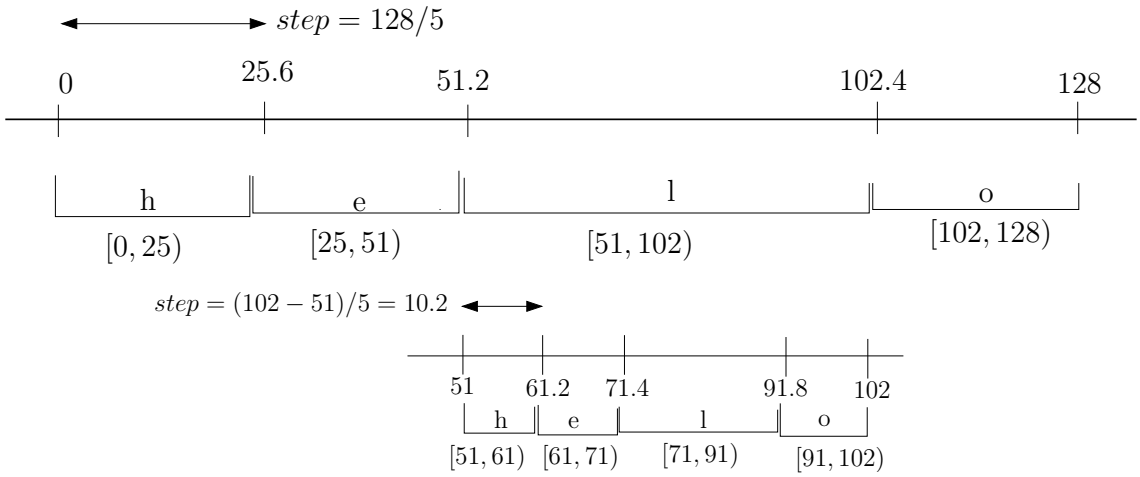


Figure 2.2: Integer Arithmetic Coding using the probability model from table 2.1. The figure illustrates all possible subintervals for the first symbol and all possible subintervals for the second symbol in case of *l* as the first symbol.

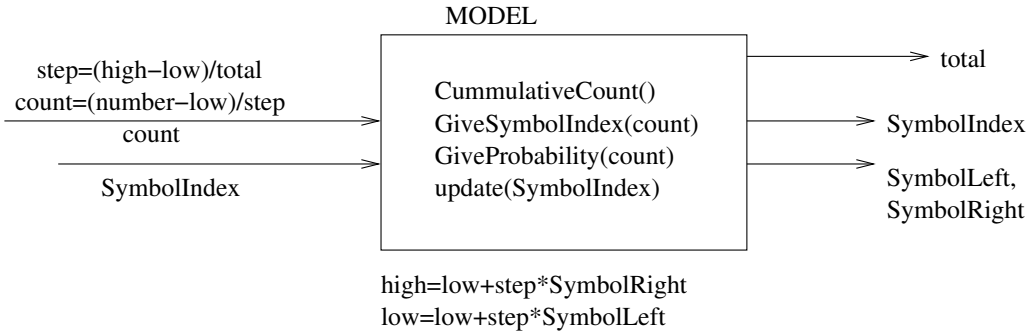


Figure 2.3: Decoding functionality: The main operations are taken by the class *model*, which can return the *cumulative count*, estimate the next decoded symbol with the given variable *count*, give the probabilities *SymbolLeft* and *SymbolRight*, and update the model.

decoding functionality is depicted in figure 2.3. The main steps for the decoder are given as follows:

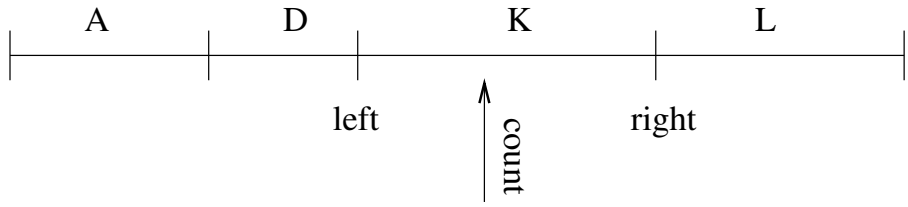


Figure 2.4: Example of finding the appropriate symbol on the probability line when decoding with integers. The function *GiveSymbolIndex(count)* scans the symbols starting from the left most symbol until the cumulative count is larger than the input variable *count*.

1. Init variables:
   $low = 0$
   $high = RANGE$
2. Get *total* from probability model
3. $step = (high - low)/total$
4. $count = (number - low)/step$
5. Use *count* to get the *SymbolIndex* from the model
6. Use *SymbolIndex* to get *SymbolLeft* and *SymbolRight* from the model
7. Update the model
8. Break if the End-Symbol was decoded
9. $high = low + step \cdot SymbolRight$
   $low = low + step \cdot SymbolLeft$
10. Goto 2.

The function *GiveSymbolIndex(count)* from step 5 of the decoder is now discussed in more detail. The function has to find the corresponding symbol interval for *count*. An example is given in figure 2.4. In this case, K would be the symbol to be returned. The function uses an array *SymbolsCount[]* where the counts of the symbols are stored and gets the variable *count* as an input argument. It is realized as follows:

1. $SymbolIndex = 0$
2. $SymbolRight = 0$
3. while (1)
   (a) $SymbolRight$=$SymbolRight$+SymbolsCount[$SymbolIndex$]
   (b) if ($count < SymbolRight$) break
   (c) $SymbolIndex$ ++
4. return $SymbolIndex$

2.3.4.3 The statistical model

SymbolLeft and *SymbolRight* are calculated with the count of the symbol. For encoding and decoding, a simple statistical model is employed that updates the counts of the symbols in dependency of their occurrence. This is done adaptively on the receiver's side as well as on the sender's side. The counts for each symbol are initiated with 1. For each symbol, the state of the model is equal when encoding or decoding. When a symbol is coded or decoded, the count of it is incremented. This kind of model is called *order 0*-model. The order of the model is given as the number of symbols that go into the probability estimation minus one. For instance, if three symbols are encoded as a whole, the model order equals two. In a later section the method *prediction by partial matching* (PPM) is described. This method is actually just an extension from the model order 0 to higher model orders. Such a more complex statistical model achieves better compression performance.

2.3.5 Performance Results

For the performance evaluation, the *Calgary* corpus [29] is employed. This corpus is a collection of text and binary data files, which are commonly used for comparing data compression algorithms. The Calgary corpus was founded for the evaluation in [30] and is further employed in [21]. It consists of 18 files including different data types, as described in the appendix in tables 5.1 and 5.2 on page 146. Bell et al. describe the files in [21] as follows:

> *"Normal" English, both fiction and nonfiction, is represented by two books and papers (labeled book1, book2, paper1, paper2). More unusual styles of English writing are found in a bibliography (bib) and a batch of unedited news articles (news). Three computer programs represent artificial languages (progc, prog1, progp). A transcript of a terminal session (trans) is included to indicate the increase in speed that could be achieved by applying compression to a slow line to a terminal. All of the files mentioned so far use ASCII encoding. Some non-ASCII files are also included: two files of executable code (obj1, obj2), and some geophysical data (geo)- in figure ...- and a "bit-map" black-and-white picture (pic). The file "geo" is particularly difficult to compress, because it contains a wide range of data values, while the file "pic" is highly compressible because of large amounts of white space in the picture, represented by long runs of zeros.*

Figure 2.5 illustrates the compression performance for the files in bits per byte (bpb). The coder uses an order 0 model for the symbol statistics. The compression is very moderate due to the low model order.

2.3.6 Summary

In this section, the principle of arithmetic coding was explained. In conjunction with a statistical model, arithmetic coding can perform efficient data compression. A method for programming arithmetic coding and selected parts of the source code were detailed. The realized statistical model relates to order 0 and can be extended to higher orders to achieve better compression. In the next section, such an extension is described.

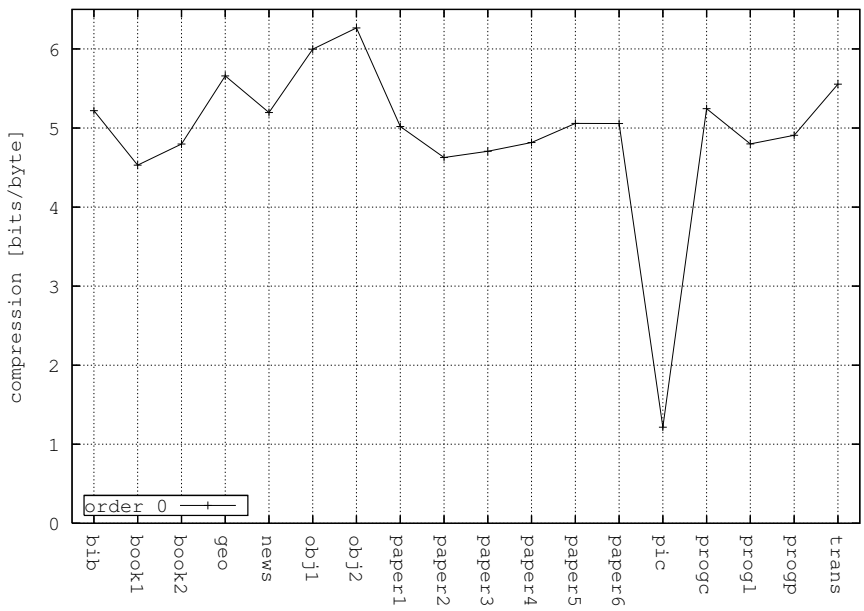


Figure 2.5: Compression results of the arithmetic coder for the files of the Calgary corpus.

2.4 Prediction by Partial Matching (PPM)

2.4.1 Introduction

In this section the method *prediction by partial matching* (PPM) is described and implemented. The idea of PPM is to provide and to exploit a more precise statistical model for arithmetic coding and thus to improve the compression performance.

PPM belongs to the text compression class of statistical coders. The statistical coders encode each symbol separately taking their context, i.e., their previous symbols into account. They employ a statistical context model to compute the appropriate probabilities. The probabilities are coded with a Huffman or an entropy coder. The more context symbols are considered, the smaller are the computed probabilities and thus the compression is improved. The statistical coders give better compression performance than the dictionary coders that employ the *sliding window* method Lempel Ziv 1977 (LZ77), however, they generally require large amounts of random access memory (RAM) [22].

PPM uses the technique of *finite context modeling*, which is a method that assigns a symbol a probability based on the context the symbol appears in. The context of a symbol is defined by its previous symbols. The length of a context is denoted as the *model order*. Similarly than with arithmetic coding, the symbols are coded separately with the difference that now the context of a symbol is taken into account. For instance, when coding the symbol *o* of the message *hello*, the order 4 context is given as *hell*, the order 3 context is given as *ell*, and so on. As the context model allows for prediction of characters with a higher probability, less bits are needed to code a symbol. The here described technique can also be useful for entropy estimation of symbol sequences or sensor data.

The outline of this introduction is given as follows. In the next section related references to PPM are given. In section 2.4.3 the principle of PPM is explained. Section 2.4.4 describes the own implementation. Finally, a summary of this introduction is given in section 2.4.6.

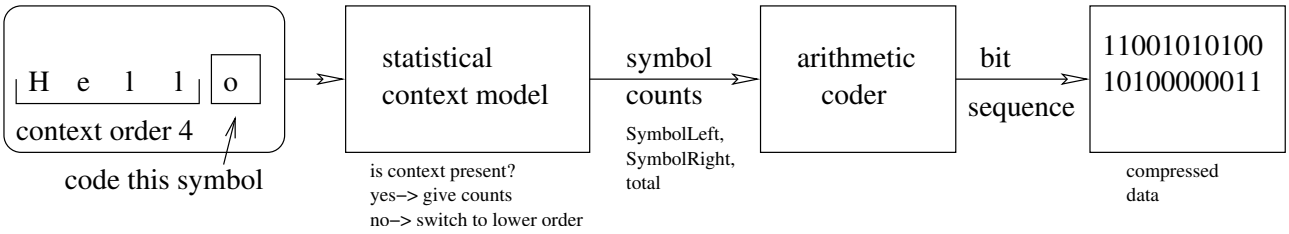


Figure 2.6: Principle of data compression scheme *prediction by partial matching* (PPM). A context model estimates symbol statistics which are passed to an arithmetic coder.

2.4.2 Related References

Techniques for adaptive context modeling are discussed in [31] and [32]. The original algorithm for PPM was first published by Cleary and Witten in [7] and improved by Moffat [33] [34], resulting into the specific method PPMC, which is the reference throughout this introduction. As PPMC has high computational requirements, it is still not widely used in practice. PPMC outperforms Ziv-Lempel coding in compression, thus it is an interesting candidate for complexity-reduction. In the following PPMC will for simplicity be referred as PPM.

A very similar approach to the implementation of PPM described here is given in [35] [36]. As it will be detailed in subsection 2.4.4.3, however, in this work a different concept for the data structure is introduced. For a general introduction into the field of context modeling for data compression, see [37] [21] [22]. In [30], different strategies for adaptive modeling are surveyed, including finite context modeling, finite state modeling, and dictionary modeling.

2.4.3 Data Compression with Context Modeling

PPM consists of two components, a statistical context model and an arithmetic coder, as illustrated in figure 2.6. Each symbol is coded separately taking its context into account. Figure 2.7 illustrates two context models, where figure a) refers to order 0 and figure b) to order 1. The context model stores the frequency of each symbol and arranges them on a so-called *probability line*, as illustrated in figure 2.7 a). Thereby, each symbol in the context tree is assigned a *SymbolLeft* and a *SymbolRight* count (also called left and right count). For a symbol i, these counts are calculated as

$$SymbolLeft\ (i) = \sum_{\forall j<i} count(symbol(j)) \tag{2.6}$$

$$SymbolRight\ (i) = SymbolLeft\ (i) + count(symbol(i)), \tag{2.7}$$

where $count(symbol(i))$ denotes the statistical count of the symbol i. These two statistical counts are needed when the model is queried by a symbol with a given context. Note that each implementation of a statistical model has generally a maximum context order.

When a symbol is to be encoded the model is first checked for the symbol with a given context of this order. If it is in the model, the left and the right count can be retrieved and the symbol is encoded. If not so, an *Escape* symbol is transmitted and the next lower order is checked. The escape symbols are employed to signal the decoder the current model order. Similarly as each symbol with a given context has a left and a right count, an escape symbol also has a $SymbolLeft_{Esc}$ and a $SymbolRight_{Esc}$ count, so that it can be encoded as a regular

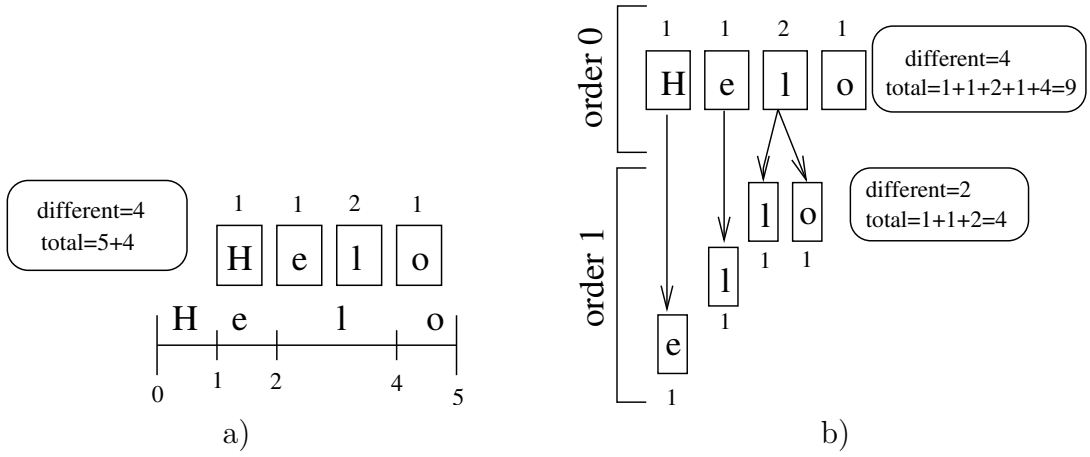


Figure 2.7: Order 0 (figure a)) and order 1 (figure b)) context model after coding the message *hello*. The order 0 model only has one context with four different symbols. A context is typically arranged on a line, where each symbol has a left and a right count according to its frequency of occurrence. The total count of a context is calculated by the sum of the number of different symbols and the statistical counts. The order 1 model in figure b) has four different contexts. A context contains all characters with the same previous symbol(s). The illustrated context *ll, lo* contains the symbols *l* and *o*. It has thus two different symbols and a total count of 4.

symbol. The counts are calculated as

$$SymbolLeft_{Esc} = \sum_{\forall i} count(symbol(i)) \tag{2.8}$$

$$SymbolRight_{Esc} = SymbolLeft_{Esc} + different, \tag{2.9}$$

where *different* denotes the number of different symbols i (and thus the count *esc* of the escape symbol). The left count for an escape symbol is thus given as the sum of all the right counts of the symbols in the context. For the right count the number of different symbols in that context has to be added. Note that in figure 2.7 a) the escape symbol is located outside the depicted probability line at the right-hand side of the symbol "o". As $SymbolRight_{Esc}$ refers to the right count of the last symbol on the probability line, it also gives the *total* count of the context (needed for the arithmetic coder, see section 2.3).

If a symbol is even not in the order 0 model it is coded with the order -1 model, where each symbol has an equal probability, see figure 2.8 a). Both models in figure 2.7 were constructed on coding the message *Hello*. For an order 1 model, the steps for coding the first 2 symbols are given as follows:

1. Is 'H' in the model? No− > update 'H' in order 0, send escape, code 'H' in order -1
2. Is 'He' in the model? No− > update 'He' in order 1, send escape
3. Is 'e' in the model? No− > update 'e' in order 0, send escape, code 'e' in order -1

The context model is employed adaptively on the encoder and on the decoder side. When coding or decoding a single symbol, the model is in the same state on each side. The model can be realized by linked nodes within a data tree. When a symbol with a specific context is not found in the tree, it can result from three cases:

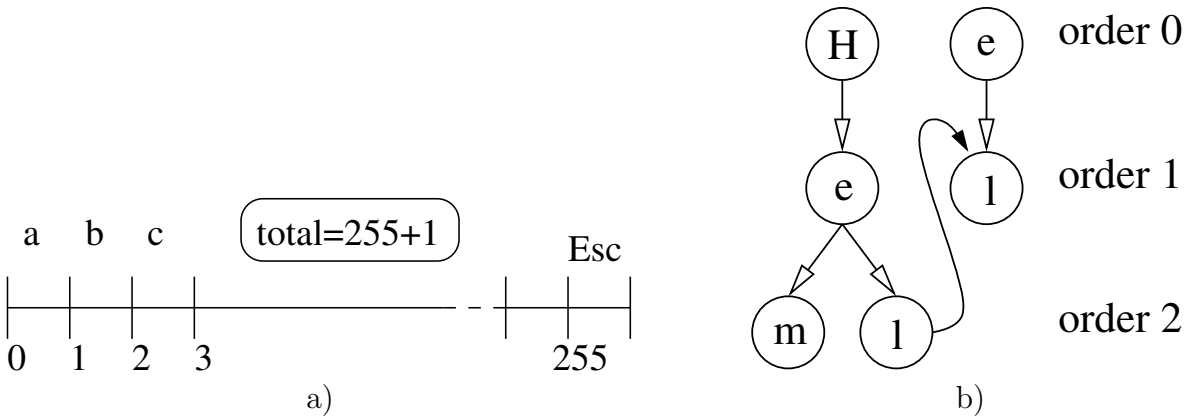


Figure 2.8: Figure a) shows the probability line for order -1: This order is used when no statistical data is in the model. Then each symbol is assigned an equal probability. The escape symbol in order -1 is sent to signal the decoder the end of the message. Figure b) shows the data tree for the symbol "l" in the word "Hel". The model is first asked for the order 2 context, which is done by checking the tree for the string "Hel". The next lower order would require the string "el" to be in the tree. The long arrow in the middle from "l" to "l" is optional. A node can contain a pointer to the context node of the next lower order. Thus, the search through the tree is accelerated when escape symbols occur frequently.

1. The symbol does not exist in the context. A new node is created and an escape symbol is sent.

2. There is no symbol in the context. A new node is created while there is no need for sending an escape. (The decoder can conclude without an escape that it has to switch to the next lower order.)

3. There is not a symbol in the context and the context does not exist. The context and then the symbol nodes have to be created. There is no need for sending an escape.

The nodes of the tree each store the count of a symbol. When a symbol is found, *SymbolLeft*, *SymbolRight*, and *total* have to be calculated. This can be done by traversing all symbols of the context. If the symbol to be coded is found, *SymbolLeft* and *SymbolRight* are stored. Then the rest of the symbols of the context are traversed till the last symbol, thus calculating the total count (in the literature, the total count is also referred as cumulative count). When traversing the context's symbols, equation 2.8 is employed concurrently.

For improving the search through the tree, there exist various methods. For example, additional pointers can be maintained by the nodes to find the next lower-order context, as illustrated in figure 2.8 b). These possibilities are not discussed here, as a hash table model is employed instead of a tree-like linked list structure, as detailed in section 2.4.4.

2.4.3.1 Full Exclusion

Full exclusion (in the literature, full exclusion is sometimes referred as *scoreboarding*) is a method for PPM to improve the compression performance. If a symbol is not found in a context of order N, the other symbols that are present in this context are stored to be excluded from the probability calculation in the next lower order $n = N - 1$, where the symbol may be found. Thereby less symbols are taken into account for the probability calculation of the symbol to be coded. A symbol is thus coded with a higher probability causing the arithmetic

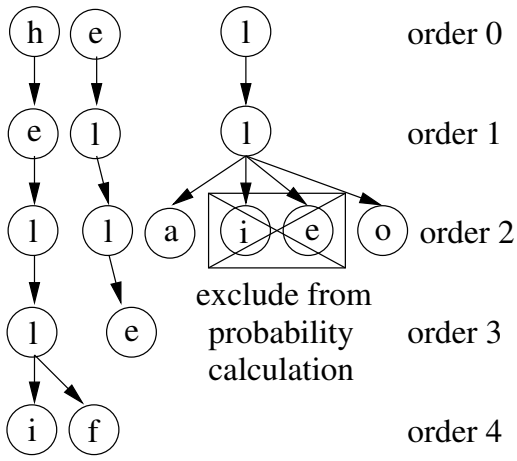


Figure 2.9: Full exclusion: The symbols that occurred in higher contexts are excluded from the probability calculation. The figure illustrates the method for an order 4 model with the message *hello* to be coded.

coder to produce less bits. The method is illustrated in figure 2.9 for the message *hello*. The symbol o is finally coded in the order 2, where the symbols i and e are excluded from the probability calculation.

2.4.3.2 Lazy exclusion

Lazy exclusion (also referred as *update exclusion*) is part of PPMC and is a strategy for updating the single symbols in a context. It means that if the symbol to be coded is found in order N of the model, only the orders $n \geq N$ are updated. The orders $n < N$ are not updated. Take for instance the word *hello* with the symbol o to be coded. If the symbol is found in order 3 and the maximum order of the model is 4, only the counts of o in the context *hello* and *ello* are updated. The lower orders are not updated.

Lazy exclusion gives slightly lower compression performance than full exclusion. As it is faster and easier to implement, it is the choice for the source code in this thesis.

2.4.3.3 Renormalization

To make PPM applicable and to locally detect a change in the statistics of the data, a *renormalization* is performed on the counts of all symbols in a context when incrementing a count would be larger than a previously defined maximum count. A renormalization means dividing all the context's counts by 2. If one byte is selected for each statistical count, the range R for the count of a symbol is defined as

$$R = [0 \ldots 255]. \tag{2.10}$$

Renormalization occurs if the count of a symbol is going to be larger than 255. Instead of incrementing 255, the counts of a context are divided by 2 and the count to be updated is incremented resulting in the number 128. The division is achieved by a left shift of the binary counts. If a count becomes 0 it is incremented to 1.

Another method for local adaption to the statistics and to constrain the memory requirements is to flush the whole model thus building up a new model. A flush routine can be performed when the compression performance drastically degrades or when the memory is exhausted.

2.4.4 Programming PPM

In this section the own PPM implementation is explained, which uses the arithmetic coder and provides a statistical context model. The model consists of a data structure to store the statistical information and the appropriate functions to update or retrieve the statistics. To make this section more readable for the general reader, very specific details and source code extracts are avoided. The interested reader is referred to the report in [38] for more details on the source code itself and its usage.

Teahan and Cleary propose a *trie*-based data structure for fixed-order models in [39]. In computer science, a *trie* or a prefix tree is an ordered tree data structure to store an associative array where the keys of the nodes are strings, see [40] for a survey on data structures. (In the following the term *tree* is used to refer to a trie or a data tree.) A tree requires functions for traversing it to access the queried data. In contrast to a tree, a hash-table technique with a smart hash-function that avoids collisions can be faster. For the own implementation, a hash table is used to manage the string data and collisions are resolved by linked lists.

The code consists of the classes *model* and *hash*. The class *model* is derived from the class *hash*. Note that the data structure just simulates a tree with nodes and branches, because it is realized with a hash table (to simulate a data tree) for fast information retrieval. Therefore, a hash table entry in the data structure may be referred as a *node*. A model can be defined as an object of the class *model*, for which specific functions for updating the model or retrieving the symbol probabilities are available. These functions are designed to fulfill the need of the arithmetic coder detailed in section 2.3. The class *hash* especially provides functions for the inner data structure of the model, concerning for instance the creation of a new data entry or the search of keys.

In the subsections 2.4.4.1 and 2.4.4.2, it is first described how the model can be used to encode and decode a data stream. In subsection 2.4.4.3 the class *hash* is described, which contains the data structure and functions for the statistical model. In subsection 2.4.4.4 the memory management is given.

2.4.4.1 Encoding the data stream

When encoding a data stream, the function *encode()* is called by the main program (*coder.cpp*). The object *MyModel* is defined and then can be employed for context model interactions. The context model sequentially is given a substring of the data stream until the string is coded. The encoding steps are given as follows:

1. Init *low* and *high* for the arithmetic coder, see section 2.3
2. Initiate an object *MyModel* of the class *model*
3. Set the maximum order of *MyModel*
4. Retrieve the left, the right, and the total count for the current symbol with the model class function *GiveProbability()*. Input of this function is the current symbol and its context. The class keeps track of the current model order and thus the function can be called many times. The function returns a flag *SymbolCoded* to indicate if the retrieved context was in the model.
5. Use the function *EncodeSymbol()* to encode the current symbol using *low* and *high* for the arithmetic coder and the retrieved symbol statistics

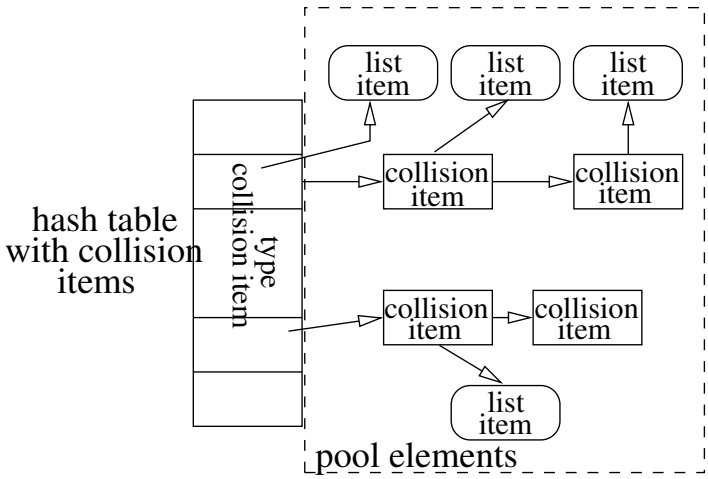


Figure 2.10: Structure of the data model: Collisions are resolved by *chaining*. A more detailed description of the data model is given in figure 2.11.

6. If $SymbolCoded == 1$ move to the next symbol

7. Go to 4.

2.4.4.2 Decoding the data stream

Similarly than for the encoding procedure, the steps for the decoding procedure are given as follows:

1. Init *low* and *high* for the arithmetic decoder

2. Do some other initializations concerning the arithmetic decoder

3. Create an object *MyObject* of the class *model*

4. Set the maximum order of the model

5. Estimate the total count for the current context with the function *GiveTotal()*, a function of the class *model*.

6. Estimate *count* through the arithmetic coder

7. Use the function *GiveSymbol()* - a function of the class *model* - to decode a symbol

8. Break if the decoded symbol signals the end of the stream

9. Store the decoded symbol in the target array

10. Perform the scaling operations of the arithmetic coder

11. Goto 5.

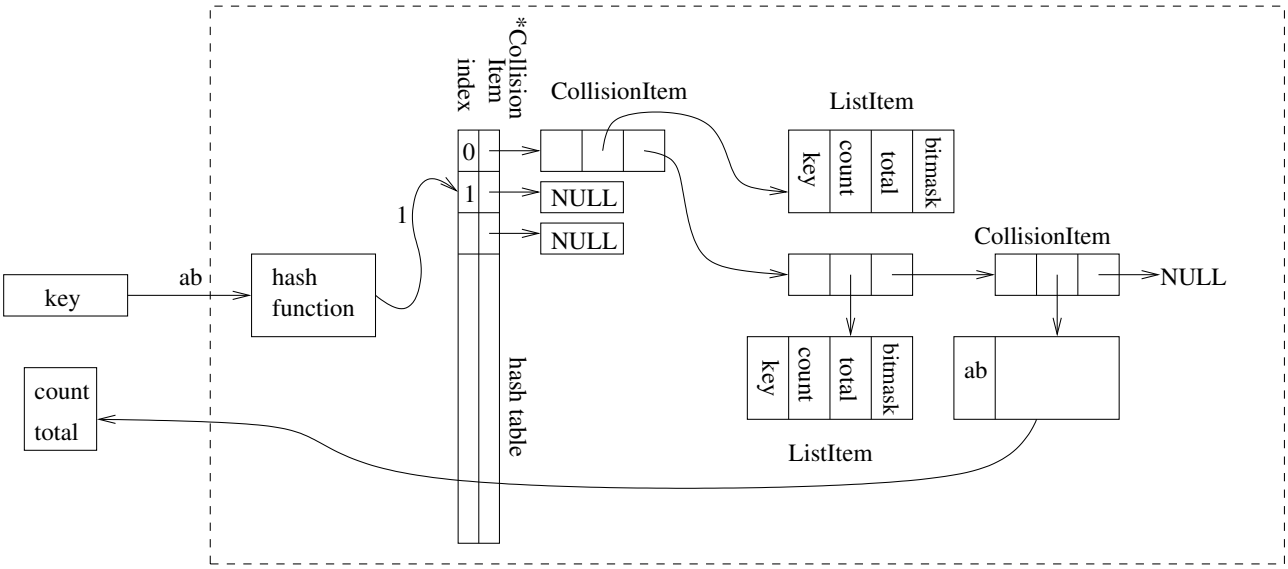


Figure 2.11: Structure and functionality of the data model: A key (a string or word) is mapped onto a hash table item through the hash function. Then the list of collision items is traversed until the correct item is found. The data for each collision item is stored in a separate list item. Such a list item contains the key, the statistical count of the key, the total count of the context where the key is located, and a bitmask, which signals existent successor nodes in the next higher order. The data model returns the statistical count for the key and the total count.

2.4.4.3 Class hash

Hash tables are employed to access a set of symbols/words by a set of keys. In case the hash table is organized as a simple array, a key is a number that indicates a certain hash table entry with the required information. Hashing is of special relevance if the set of possible keys is much larger than the set of symbols/words containing the information. In such a situation, a *hash function* (in the literature, hash functions sometimes are called *hash keys*) is employed to calculate the memory address with the required information from a key.

In context modeling, the keys are character arrays and the information is accessed by a pointer to an object containing the symbol statistics. For low order context models, hashing is not necessarily needed: For order 0, an array with 256 elements is sufficient and for order 1, an array with $256^2 = 65536$ could be allocated. With order 2, however, three characters have to be indexed, resulting in an array size of $256^3 = 16.777.216$. Higher orders soon exceed memory configurations. One possible solution is to organize the complete data tree as a linked list. The drawback of this technique is that nodes of higher orders have to be searched extensively, thus resulting in computationally lower performance.

In [36], for the orders 0, 1 and 2 the array technique is employed – that is, the symbol contexts are accessed by arrays, for each order a separate one. Each array element then contains a pointer to a linked list with the different symbols that are present in that context. The single list elements then contain the statistics. For higher orders, the hash technique is employed, where the contexts are accessed with a hash function, and similarly as for the lower orders, the symbol statistics are stored by linked lists.

In this work, a different hash function concept is applied, because the library shall especially be useful for research on hashing techniques. As illustrated in figure 2.10, for each symbol in

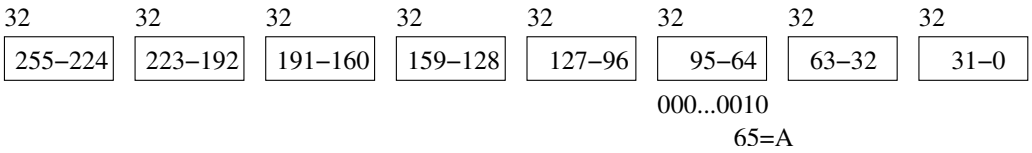


Figure 2.12: Data structure for the bitmask. It consists of eight 32-bit integer variables, where each bit indicates if a symbol is present in the context. A maximum of 256 symbols can be present in a given context. Later a novel technique is introduced where this kind of signaling technique is not needed any more.

a context a hash table entry is reserved. A single hash table is employed for all orders. The selected hash function is detailed in [41] as *One-at-a-Time Hash*, where it is evaluated to perform without collisions for mapping a dictionary of 38470 English words to a 32-bit result. Its source code is given in figure 5.4 on page 161 in the appendix.

The idea of the hash function is to produce a randomly distributed integer number from an arbitrary array of byte characters. The number is located within the table size. To be more precise, the hash function has to equally distribute the set of keys that are expected to appear over the hash table entries. In ideal case, the hash function exactly foresees the set of keys that will be requested. If each requested key is mapped on a distinct hash table entry the hash key performs *perfect hashing*. The time for searching the required statistics would be of order O(1). In practice, perfect hashing is often not achieved. If the hash function matches several keys on a single entry, a specific hash table technique has to be employed to resolve the *collision*. The technique of *chaining* is used here, where collisions are resolved by a linked list, as illustrated in figure 2.10. In case the number of collisions is small, the hash function still performs well.

The data structure as illustrated in figure 2.10 consists of two different data objects, the *CollisionItem* and the *ListItem*. An object of the class *CollisionItem* contains a pointer to a list item and a pointer to its successor.

An object of the class *ListItem* contains the key and the statistics, i.e., a pointer to the character array, the length of the array in bytes, and the symbol count. In addition, such an object also contains a function for key comparison and importantly an object of the class *bitmask*. Note that the total count, which is required by the arithmetic coder, is not included as a variable to reduce the memory requirements. The total count is computed on the fly in that the rest of the symbols on the probability line (starting from the symbol to be coded or from the symbol that was decoded) are traversed.

The class *bitmask* is employed to indicate if a symbol in a given context is present. As illustrated in figure 2.11, each object of the class *ListItem* contains a bitmask. A bitmask represents the branches of the tree. Each node not only includes the symbol statistics but the information of the existent successor nodes in the next higher order. Figure 2.12 depicts the structure of a bitmask. Each node in a data tree has a context in which up to 256 symbols can be present. The bitmask is an array of bits each one denoting whether a symbol in the context is present or not. The bits are stored in eight integer variables.

2.4.4.4 Memory management

The statistical data for the nodes of the data tree is maintained by three data pools, i.e., a pool for the collision items, a pool for the list items, and a pool for the keys (a key is a string with a variable length), which are illustrated in figure 2.13. The memory for these pools has to be

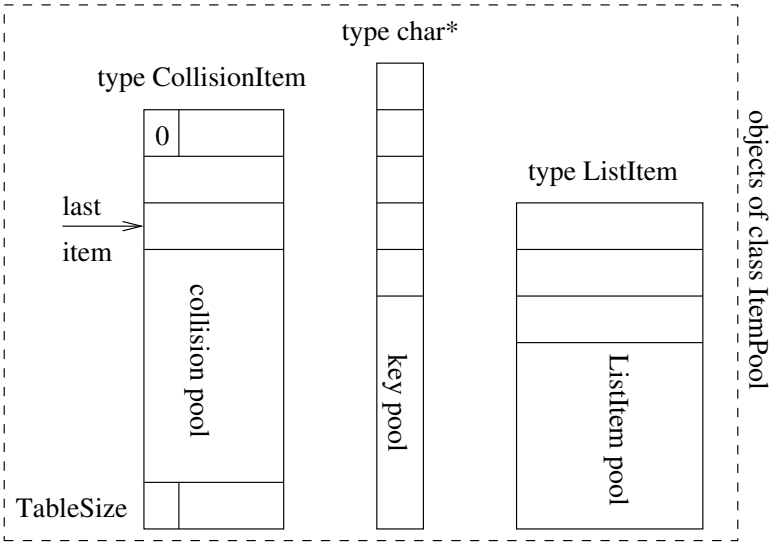


Figure 2.13: The memory is managed by three pools, which are allocated at the beginning of the program with the class *ItemPool*. These pools are arrays of a fixed dimension and store the collision items, the list items, and the keys that belong to each list item. The dimensions are set at the beginning of the program.

allocated at the beginning of the program. For this purpose the class *ItemPool* is employed, which can create three different objects of the types *CollistionPool*, *KeyPool*, and *ListItemPool*. The three pools are created in the constructor of the class *hash*. Thus the pools are created automatically when an object of the class hash is created. As the class model is derived from the class hash, the object of the class hash is created automatically with the definition of the model. Therefore, the default pool sizes are defined in the constructor of the class model and are given as follows:

- number of collision items: 65536
- number of keys: 2097152
- number of list items: 2097152
- length of the hash table: 2097152

The defaults are selected to allow for complete maintenance of the statistical data that can be collected for any file of the training data. In section 2.5 an option for the user to parametrize the pool sizes is added to the program.

2.4.5 Performance Results

Similarly than for the arithmetic coder, for the compression evaluation of the PPM implementation the *Calgary* corpus is employed. The measured compression performance for the orders 0-4 is given in the figure 2.14. The metric for compression is given in bits per byte (bpb). From order 3 to 4, there is only a little compression gain. Higher orders are expected not to improve the compression performance. For the file *geo* the compression performance is even worse for the orders 3 and 4, possibly because of the wide range of data values with small counts, which

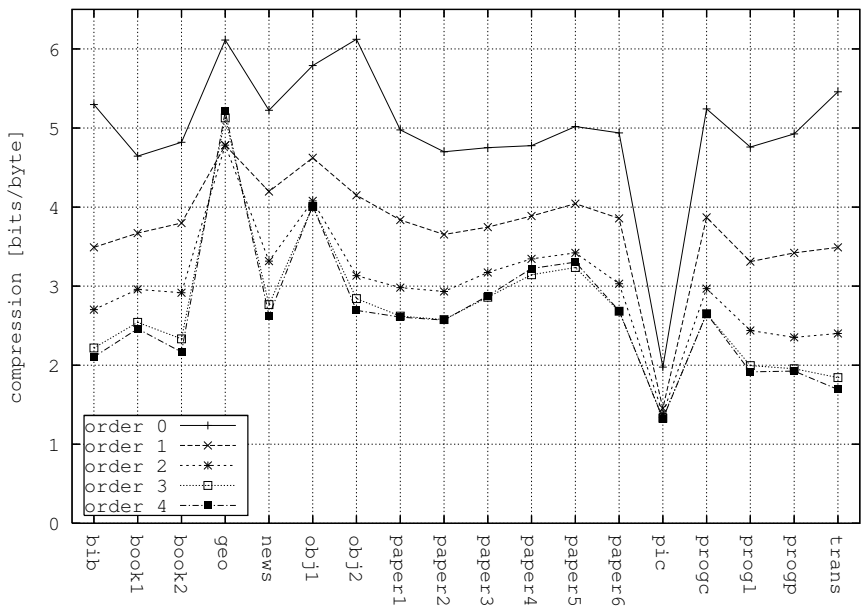


Figure 2.14: Compression results achieved by the own PPM implementation using the files of the Calgary corpus. The measurements are in accordance with the study in [36].

can cause frequent transmission of escape symbols. The given results are comparable to the evaluation in [36].

Figure 2.15 illustrates the performance of the arithmetic coder from section 2.3 compared to the results of the PPM implementation using a model order 0. The measurements give different results because the PPM implementation uses a scaling procedure. Generally, scaling results into better compression performance because the saturation of the statistical model is prevented. For the files *geo* and *pic*, however, the compression is worse. Both files contain data that is either very difficult or very easy to compress (as mentioned on page 20). The reason for the worse ratios may thus be that the scaling influences the probability of the frequent and not the rare symbols. If the statistical model is very unbalanced, that is, there are only very frequent and very rare symbols in the model, the scaling procedure can result into a more inaccurate statistical model.

2.4.6 Summary

PPM consists of an arithmetic coder and a statistical context model. Similarly than with arithmetic coding, each symbol is coded separately with the difference that the context of the symbol is taken into account. Thus a much better compression than for the order-0 model in the previous section is achieved. The compression can even be improved with specific update exclusion or renormalization techniques.

An important detail is that similarly as with the the order-0 model of the previous section, the context model works adaptively. That means that the statistics are gathered throughout the coding process. Similarly than the encoder, the decoder updates its model with each decoded symbol. Thus the model evolves equally at the encoder and the decoder.

In the second part of the section the own implementation of PPM in C++ is described. The class *model* allows for creation of a context model that includes functions to maintain the data tree and to compute the statistics. The data tree is realized through a hash function that

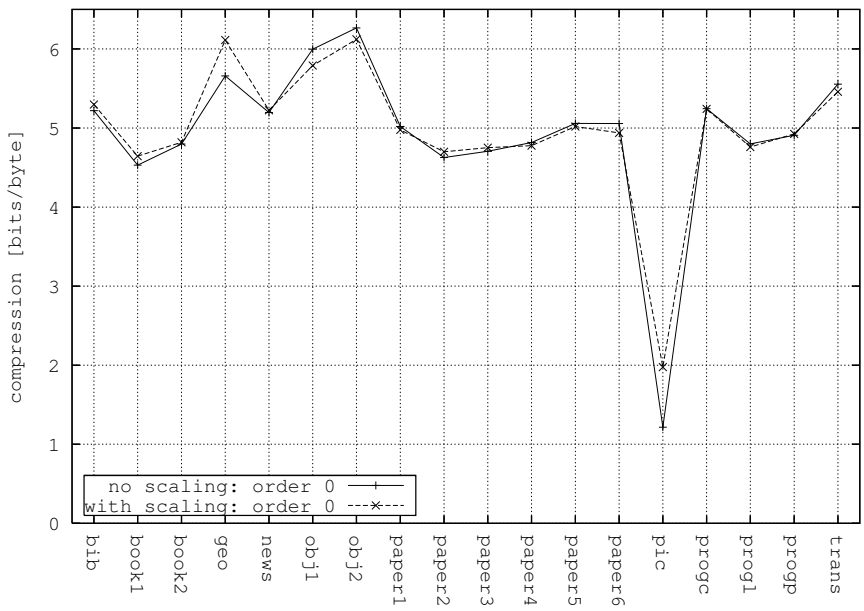


Figure 2.15: Comparison of the performance of the arithmetic coder with the order 0 model and the PPM implementation with order 0 and scaling procedure. The scaling procedure divides all symbol counts by two if a limit of 255 is reached by any symbol.

maps the strings to a list item of an array, where the statistical data of a node is stored. (Some features of the hash function are analyzed in the next section.) Collisions are resolved by linked list items. Such an implementation is much faster than a regular data tree.

The compression is verified with the files of the Calgary corpus for all model orders lower or equal than 4. Even if the implementation allows for higher model orders, the compression is only improved marginally for orders higher than 4.

The idea of this work (part text compression) is to design a low-complexity scheme for short messages. The method of PPM shall be taken as a starting point. The principle of PPM requires large amounts of statistical data to be stored. Furthermore, a set of functions is needed to access and maintain the data. This became even more transparent with the own implementation. The task is now to simplify the method and the own program while the compression performance should be the same. To allow for this, a deeper understanding of the statistical evolution throughout the coding process is necessary. In the next section the PPM-system is extended by functions in order to analyze what is going on in the model throughout the coding process.

2.5 Analysis of the Statistical Evolution

2.5.1 Introduction

In the previous two sections arithmetic coding and a statistical context model were detailed to form the text compression method *prediction by partial matching* (PPM). This method does not fulfill the low memory requirements and features for short messages as postulated in the introduction. The main question for modifying the method in this context is given as follows: Is the data structure a good model for the upcoming statistics? This question poses a set of sub-questions, and each of these questions is connected with a functional software extension

of the PPM implementation. Till now, the own PPM coder does not allow for an analysis of the statistical data that is gathered throughout the coding procedure. In the following a list of the software features is given that were added to the own PPM implementation to conduct the measurements in this section:

1. An option to write the content of the statistical model to a file in such a format that it can be analyzed by a high-level language like Matlab/Octave
2. The option to parametrize the internal model by the user (and not through the source code) so that the pool size can be easily varied: This includes the maximum number of keys, list- and collision items, and the hash table size.
3. An option to preload the data structure using a file with the statistical data
4. A function to flush/reset the internal model
5. A switch for *static context modeling*: In this mode the statistics are not updated throughout the compression.
6. An option to set the maximum count before rescaling starts, as in the past this value was fixed to 255.

The features can be controlled through command-line options of the encoder or decoder. A detailed description of the source code extensions and a manual for their usage is given in [42].

In the next subsections an evaluation is performed in order to gather insights on the compression routine and the computational requirements, which do especially concern the RAM memory. In the previous sections only the *Canterbury* text files were employed as the intention was to verify the own software. As the following evaluations shall now reveal insights for the development of a novel scheme, additional text files are included. The complete list of the selected English text files is given as follows:

- Files *alice29, asyoulik,* and *plrabn12* from the *Canterbury* corpus; the Canterbury corpus was developed in 1997 as an improved version of the Calgary corpus and the selection of files is explained in [43]. The files are given in table 5.3 on page 147 in the appendix.
- Files *hrom110* and *hrom220* from the *project Gutenberg* [44]
- All the text files from the *Calgary* corpus [30] listed in tables 5.1 and 5.2 on page 146 in the appendix
- The files *bible* and *world192* from the *large corpus*, available at `http://corpus.canterbury.ac.nz/descriptions`, see table 5.4 on page 147 in the appendix

Figure 2.16 shows the compression performance for all files including the non-text files. All the files are employed for the measurements in section 2.5.2. In the later sections the non-text files are excluded.

The evaluation is structured as follows. Section 2.5.2 reflects the effect of different maximum counts that cause the statistical model to be flushed. In section 2.5.3 the compression performance for adaptive context modeling is analyzed for reduced memory settings. In section 2.5.4 the performance of the hash key is verified in that it is checked if the statistical

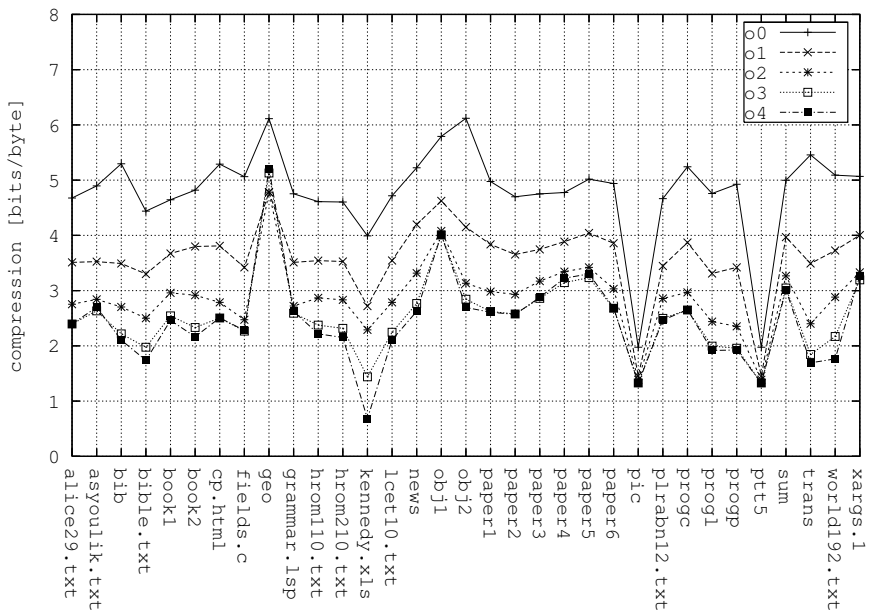
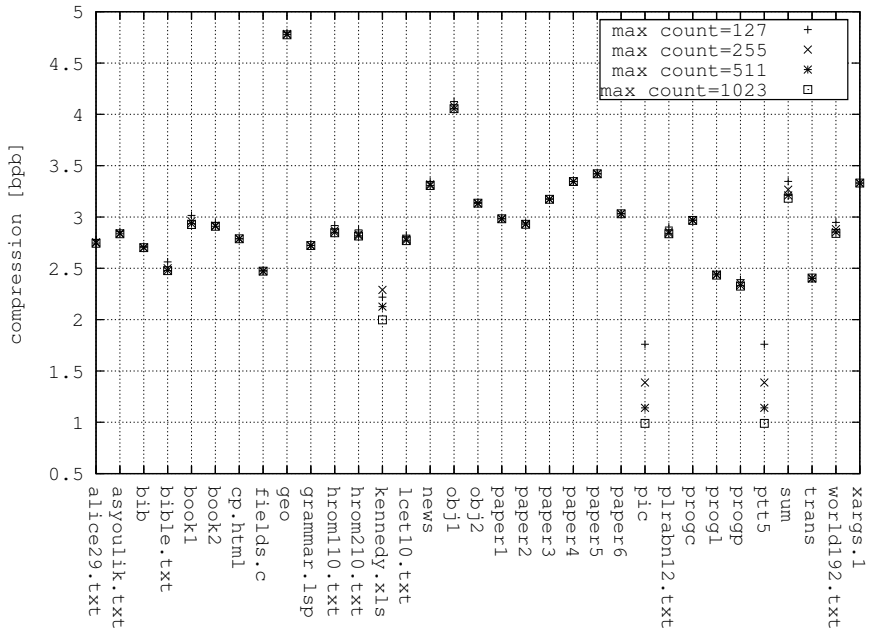


Figure 2.16: Overall compression results when there are no memory constraints for the files of the Canterbury, the Calgary, the Large corpus, and the files hrom110/220 from the project Gutenberg. For the Calgary and the Canterbury corpus the non-text files are included in this plot to allow for a comparison with figure 2.17.

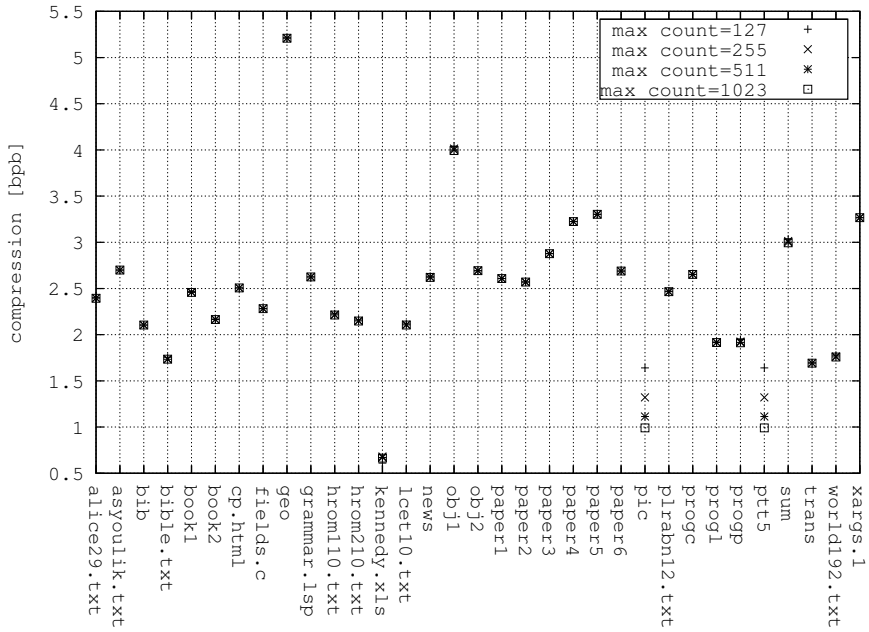
information is equally distributed over the data structure. The type of statistical information, i.e., the context order of a stored string/key, is illustrated in section 2.5.5. In section 2.5.6 the total number and length of context nodes within the tree is measured for all training files. In section 2.5.7 the statistical evolution over time is illustrated for context nodes of different lengths. The measurements in section 2.5.8 serve to gather insights on *static* context modeling, where a model is preloaded before the compression starts and is not updated throughout the compression. In the last section the measurement series is summarized and reflected.

2.5.2 Effect of Rescaling

By default, the own PPM compressor uses a maximum symbol count of 255 and divides all counts by two if any count is to be exceeded. A count thus requires one byte and exists for each list item. In the previous subsection the implementation was extended to allow for maximum counts/rescaling factors defined by the user (in this work the maximum counts are called *scaling* or *rescaling* factors and do not refer to the division factor, which always equals 2). Figure 2.17 shows the effect of the different rescaling factors $r = [127, 255, 511, 1023]$ on the compression performance. The figure illustrates that the maximum count has only little effect on the compression. An effect is visible for the non-text files kennedy.xls, pic, and ppt5 in that the compression is improved by a larger scaling factor. This is due to the statistical features of these files, specifically, the large discrepancy of symbol occurrences. As this discrepancy is not typical for text files, a larger variable size for the counts is not considered in this work.



a) order 2



b) order 4

Figure 2.17: Effect of different maximum count variables on the compression for order 2 in figure a) and order 4 in figure b). The maximum statistical counts are given as 127, 255, 511, and 1023. Enlarging the count results in small improvements for some of the files. For order 4, the improvement is only visible for the file pic and ptt5. The size of the count variable only has little effect on the compression and is thus not further considered in this work.

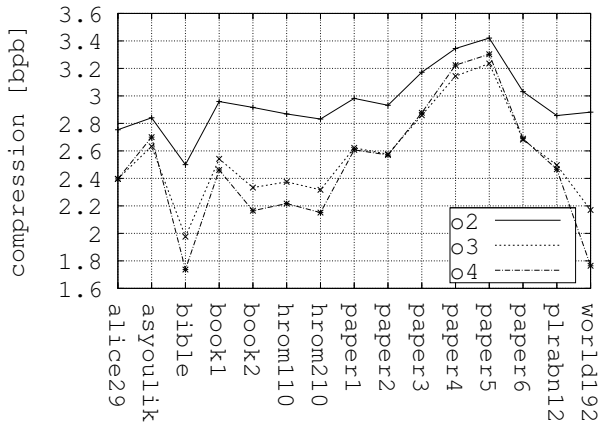


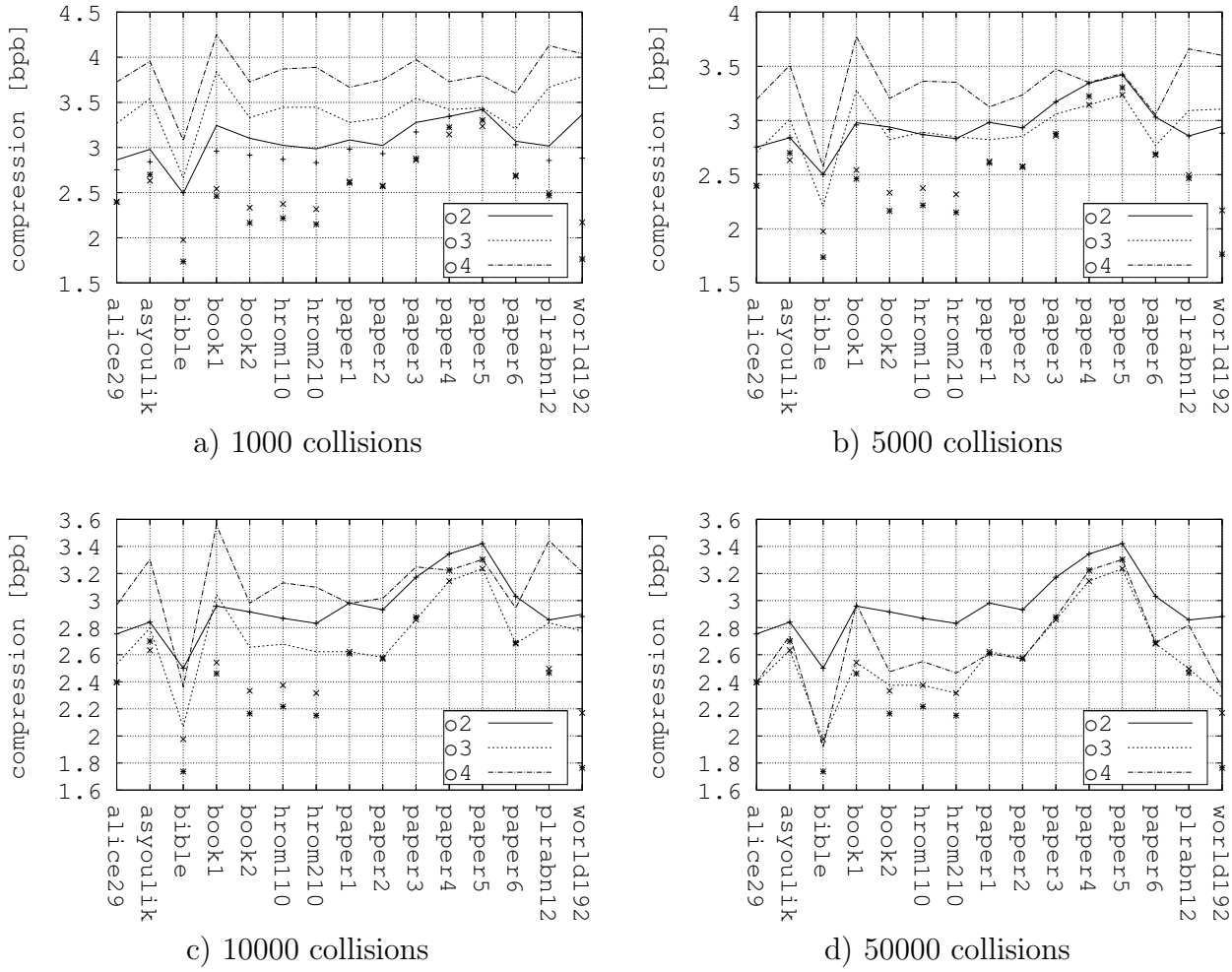
Figure 2.18: Adaptive compression performance with no memory constraints. This figure is almost the same than figure 2.16 with the difference that non-text files and orders 0,1 are excluded. The plot serves as a reference for the measurements in section 2.5.3, where the memory is reduced for text-files and the loss in compression is to be analyzed.

2.5.3 Adaptive Context Modeling with Memory Constraints

In this section the compression results for adaptive context modeling using reduced statistical context models are given. The reduction concerns a limited number of possible collisions while the hash table size is varied. As detailed in section 2.4, the statistical data is mapped through a hash function onto a hash table with a fixed table size. As the table size is much smaller than the possible space of keys, one hash table entry can be valid for a set of keys. This set is resolved by collision items.

Figure 2.18 gives a compression performance plot with no memory constraints for all files for the orders 2-4 similarly than figure 2.16 excluding the non-text files. The plot is given as a reference for the following plots with memory constraints.

Figures 2.19 and 2.20 depict the compression performance with limited memory for the text files from the selected corpora with the hash table sizes 16384 and 131072 (the plots for table sizes 32768 and 65536 are given in the appendix in figure 5.2 on page 149). Note that the table size has to be a power of 2 to uniformly fill the hash table. The maximum number of collisions is varied as $c = [1000, 5000, 10000, 50000]$. When the maximum number of collisions is attained the complete model is flushed. The plots can be interpreted as follows.



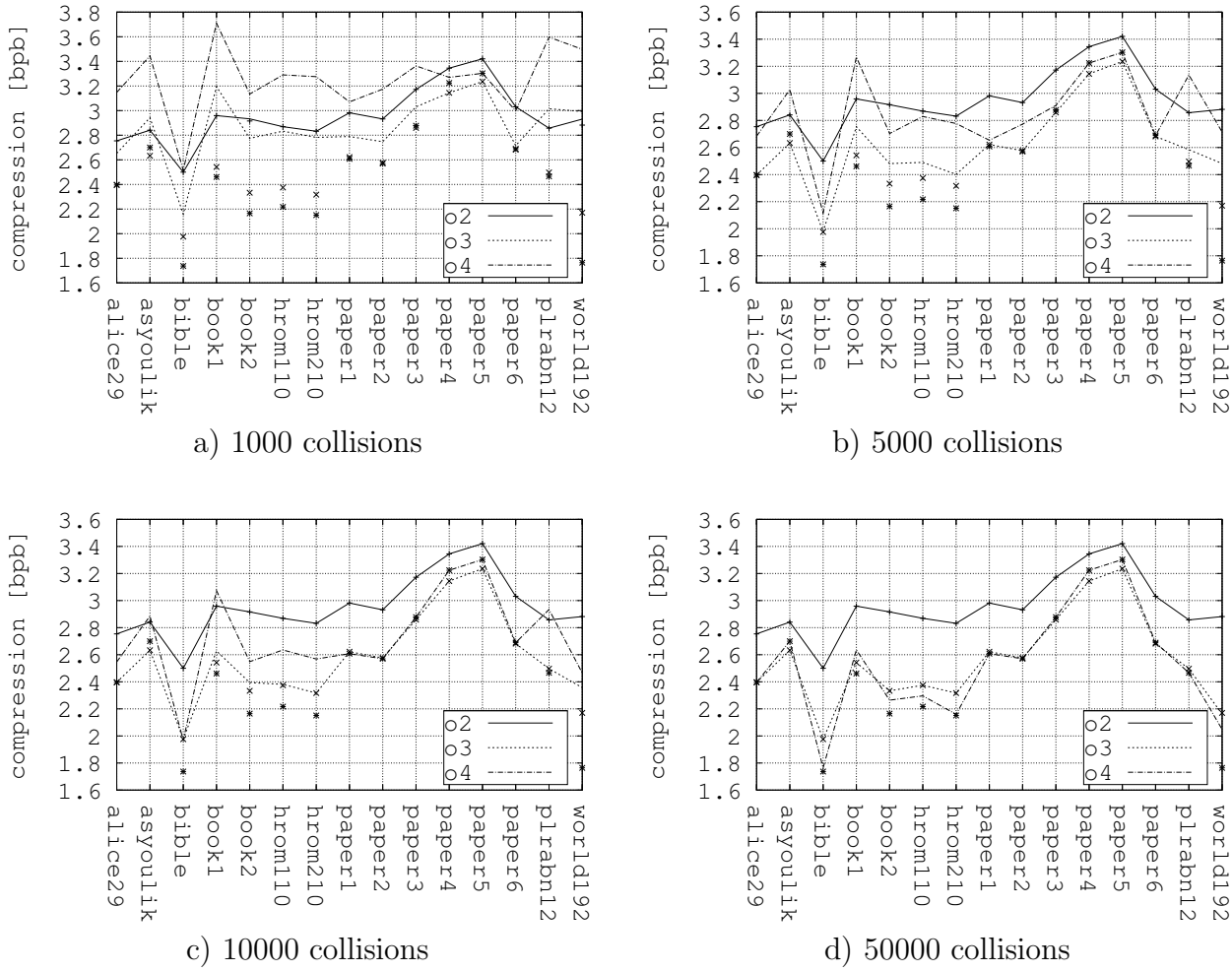
a) 1000 collisions

b) 5000 collisions

c) 10000 collisions

d) 50000 collisions

Figure 2.19: Adaptive compression performance for a hash table size of **16384 elements**. Figures a)-d) give the performance for the number of maximum collisions given as 1000, 5000, 10000, and 50000. For order 2 in figures b)-d) the compression is reasonably well. For order 3 figure d) illustrates that the model is sufficient. For order 4 none of the models is applicable. The data points are given for comparison to the case of unlimited memory, as given in figure 2.18.



a) 1000 collisions b) 5000 collisions

c) 10000 collisions d) 50000 collisions

Figure 2.20: Adaptive compression performance for a hash table size of **131072 elements**. Figures a)-d) give the performance for the number of maximum collisions given as 1000, 5000, 10000, and 50000. As expected, all the models are applicable to order 2. For order 3 10000 collisions or more should be allocated. Order 4 does not make sense for the constrained models, as the additional amount of memory needed is not at the rate of compression improvement.

Hash table size of 16384 elements:

a) **1000 collisions** The performance of order 2 is approximatively 0.2 bpb worse than without memory constraints. Order 3 and 4 do not give performance improvements at all. The model is exhausted for these orders.

b) **5000 collisions** For order 2 the performance is similar than without constraints. Thus the number of collisions is sufficient for this order. For order 3 the performance is improved up to 0.2 bpb for nine of the files, however, compared to the performance without constraints the compression is up to 0.7 bpb lower. For order 4 the model is exhausted.

c) **10000 collisions** For order 2 the model works fine (similarly than for case b)). For order 3 there is an improvement visible compared to case b), as it gives (with the exception of file book1) an improvement over order 2 in the range of 0.025..0.4 bpb. For some of the files the performance is already similar to the performance without constraints. For order 4 the model is still exhausted, however, for some files the performance is at least a little bit better (up to 0.1 bpb) than for order 2.

d) **50000 collisions** For order 2 the model works similarly than in b) and c). For order 3 the performance is very similar to the case without constraints (up to 0.05 bpb worse). Thus the given model size for order 2 and order 3 may be sufficient. For order 4 there is still no improvement visible compared to order 3. Thus the model size is too small for this order.

Hash table size of 131072 elements: For order 2 the model works fine for a)-d). For orders 3 and 4 the results are given as follows.

a) **1000 collisions** For order 3 the model performs up to approximatively 0.6 bpb worse than without constraints. Order 4 does perform worse.

b) **5000 collisions** For nine files the model with order 3 gives the same compression rates than without memory constraints. Order 4 performs worse than order 3 and is thus still not applicable to this model.

c) **10000 collisions** For most of the files order 3 works very similar than for the unconstrained model. A difference of approximatively 0.05, 0.06, and 0.2 bpb is visible for the files book1, book2, and world92. Order 4 still performs worse than order 3 and is not yet applicable.

d) **50000 collisions** For order 3 the model works fine. For most of the files order 4 gives similar or better compression than order 3. However, for the files asyoulik, book1, paper4, and paper5 the performance is worse. This is in accordance to the case without memory constraints, where order 4 does not give improvements for the files paper4 and paper5 compared to order 3. For the file world192 order 4 gives approximatively 0.3 bpb lower compression than for the unconstrained model. Due to the little gain that order 4 gives over order 3 and the significant difference to the unconstrained model, order 4 can still not be recommended for this model.

The question to answer by these measurement concerns the choice of the hash table size and the maximum number of collisions, i.e., the definition of the memory requirements. A constrained model with 16384 hash table elements already works very similar for order 2 than with the unconstrained model. For order 3 and table size 16384 at least 50000 collisions have

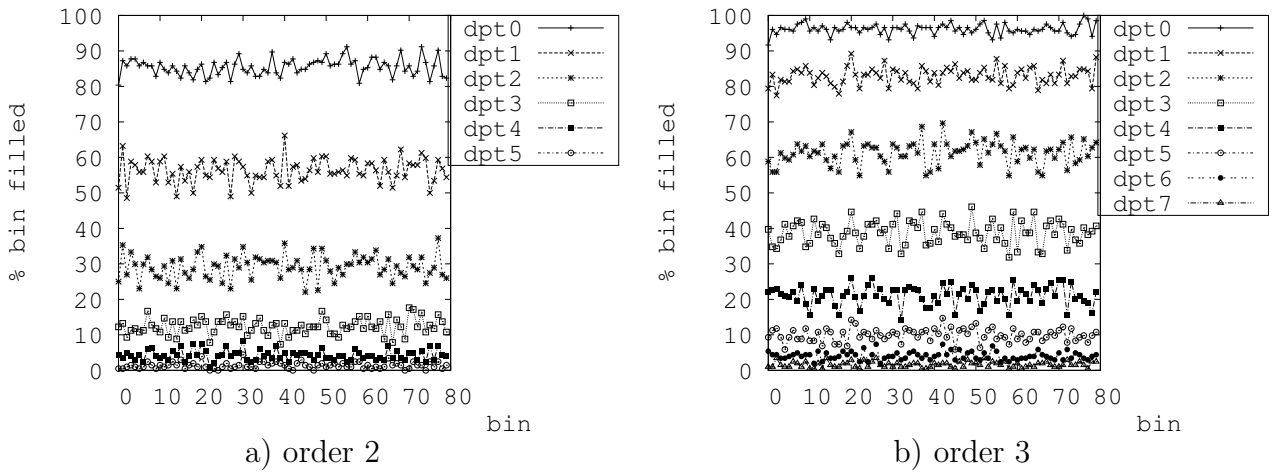


Figure 2.21: Fullness of the data structure for the file world192 with table size **16384** for order 2 (figure a)) and order 3 (figure b)). The array of table elements is summarized by 80 bins. For each bin a percentage is given to represent the amount of elements that are allocated for a collision. The depth of a collision represents the location within the linked list that is used to resolve the collisions. For order 3 the bins are much more filled due to the larger amount of statistical information. The amount of collisions decreases with more depth.

to be allowed. Alternatively the model with 131072 elements and 5000 maximum collisions can be selected. For order 4 all the models do not perform reasonably. This indicates that order 4 is not suitable for a low-complexity model. Regarding the small improvement that order 4 gives over order 3 (see figure 2.18), order 4 does not make sense when considering the additional amount of memory that is necessary to maintain this order.

2.5.4 Fullness of the Data Structure

In this section the performance of the hash key is evaluated in that it is inspected if the strings in the model are uniformly distributed over the array of hash table entries. To illustrate the distribution, the hash table entries are grouped into 80 bins. The number of 80 bins is selected for illustrative reasons. If the hash table size is for instance 16384, one bin contains $16384/80 \approx 205$ elements. For each bin it is checked how many elements of the bin are filled with data (result is given in %), while each computed result refers to a given *depth*. The depth denotes here the position of the element in the linked list, where depth 0 actually relates to the first element in the list.

Figures 2.21 and 2.22 illustrate how the internal data structure is filled for the model *world92*. (A series of measurements with the other given files, different table sizes, and orders was conducted but not included here as the two given figures summarize the results.) The percentage indicator is calculated as follows. First, the hash table elements are grouped in $NoBins = 80$ bins, each of them containing $TableSize/NoBins$ elements. This is done for each depth, where the maximum depth is the last one containing elements. Then, each bin is checked for the number of empty elements. Using this number, the percentage index is computed as

$$index(bin, depth)[\%] = \frac{FilledElements(bin, depth)}{TableSize/NoBins} \cdot 100, \tag{2.11}$$

where *FilledElements* gives the number of filled elements with a given depth for one of the 80

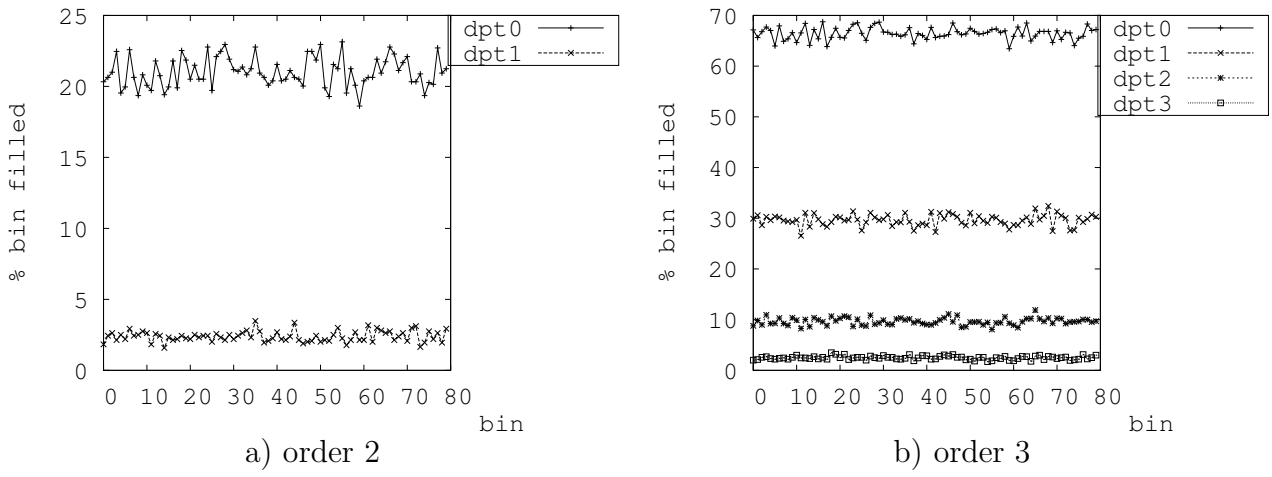


Figure 2.22: Fullness of the data structure for the file world192 with table size **131072** for order 2 (figure a)) and order 3 (figure b)). As the table is much larger now than in figure 2.21, the bins are less filled and the depth of the collisions only goes up to 3, which means that 3 collisions have to be resolved.

bins. Note that a depth of 0 relates to a table entry and not to a collision and that a *deep* collision refers to one of the last elements in a linked list. A good hash key tries to avoid collisions. Figure 2.23 illustrates the 80 bins for the depths $0 \ldots 7$.

The figures give information on the performance of the hash key in that it is shown whether the keys are uniformly distributed or the hash function has an error (with these plots two programming errors where found, and thus the compression performance could be improved). Another option for illustration would be the usage of three-dimensional plots, as it is done in figure 2.24 for the access frequency for each bin. The given plot indicates that there is a programming error in the hash key. The numerical values of the numbers on the surface plot, however, cannot clearly be read. Therefore only 2-dimensional plots for the percentage index are considered here.

The two figures 2.21 and 2.22 show that the employed hash function *One-at-a-time hash* from Bob Jenkins [41] performs well. The hash key equally fills the array of 80 bins. The deep bins are filled to a lower percentage, indicating that there exist less entries in the deep linked lists. This means that the hash key first tries to fill the elements at the front positions before the collisions force the method to allocate elements at a back-most list position. However, the hash key does not perform ideally, as a certain list position (depth) is not filled completely before the next level of positions is required.

2.5.5 Collisions Modeled by the Deep Data Structure

In the previous section the collisions were illustrated in their distribution over the hash table entries for one file. In this section, for each training file the number of collisions categorized by their depth is measured. The depth again refers to the position of the collision item in the linked list. A context node with a depth of 0 is the first entry in the list and is not a collision. Depth 1 reflects the first collision in the linked list, depth 2 the second one, and so on. If a node is very deep in the linked list, many operations are necessary to access its statistics, thus making the compression algorithm slow. In the given figures 2.25 and 2.26, the height of the bars reflects the total number of collisions for each model, which is calculated by the sum of all context nodes of one model, excluding the depth=0 nodes. The task of the hash key is to

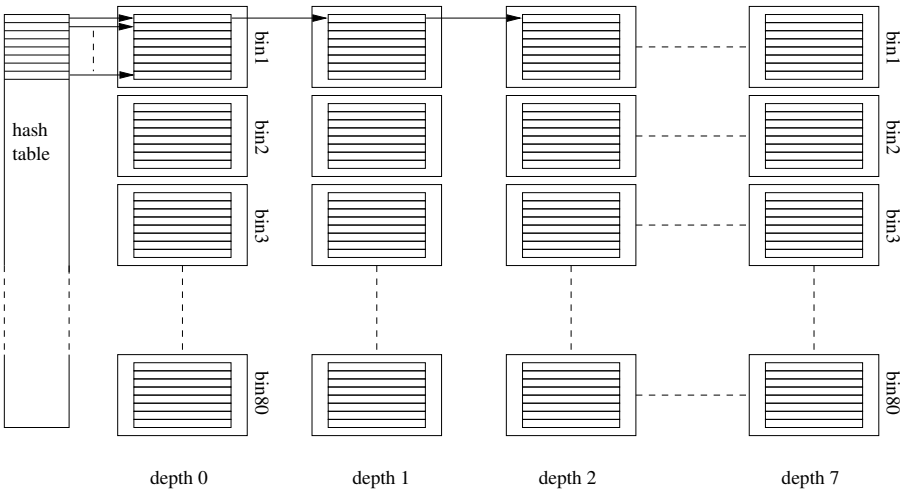


Figure 2.23: Principle of the measurements in section 2.5.4 for the fullness of the data structure. For each position in the linked list (denoted by the *depth*) there exists a number of 80 bins. For each bin the percentage of (with statistical data) filled items is measured.

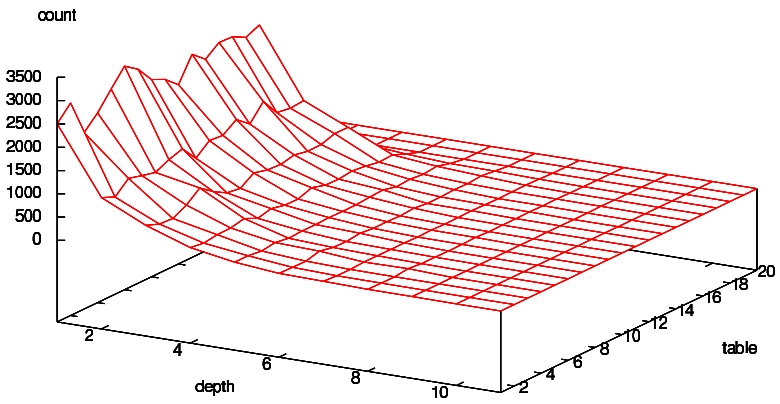


Figure 2.24: Example of a 3-dimensional illustration of the data structure loaded with data from file *alice29* with table size $100 \cdot 10^3$, where the hash table elements are grouped to 20 different bins. The depth gives the position in the linked list that is attached to each table element for resolving collisions. The *count* gives the sum for one bin of the single access frequencies of each node. The deeper bin should be more rarely accessed. The figure shows a hash error for table bins 17,18,19, and 20. This is due to the wrong choice of the hash table size, which has to be a power of 2.

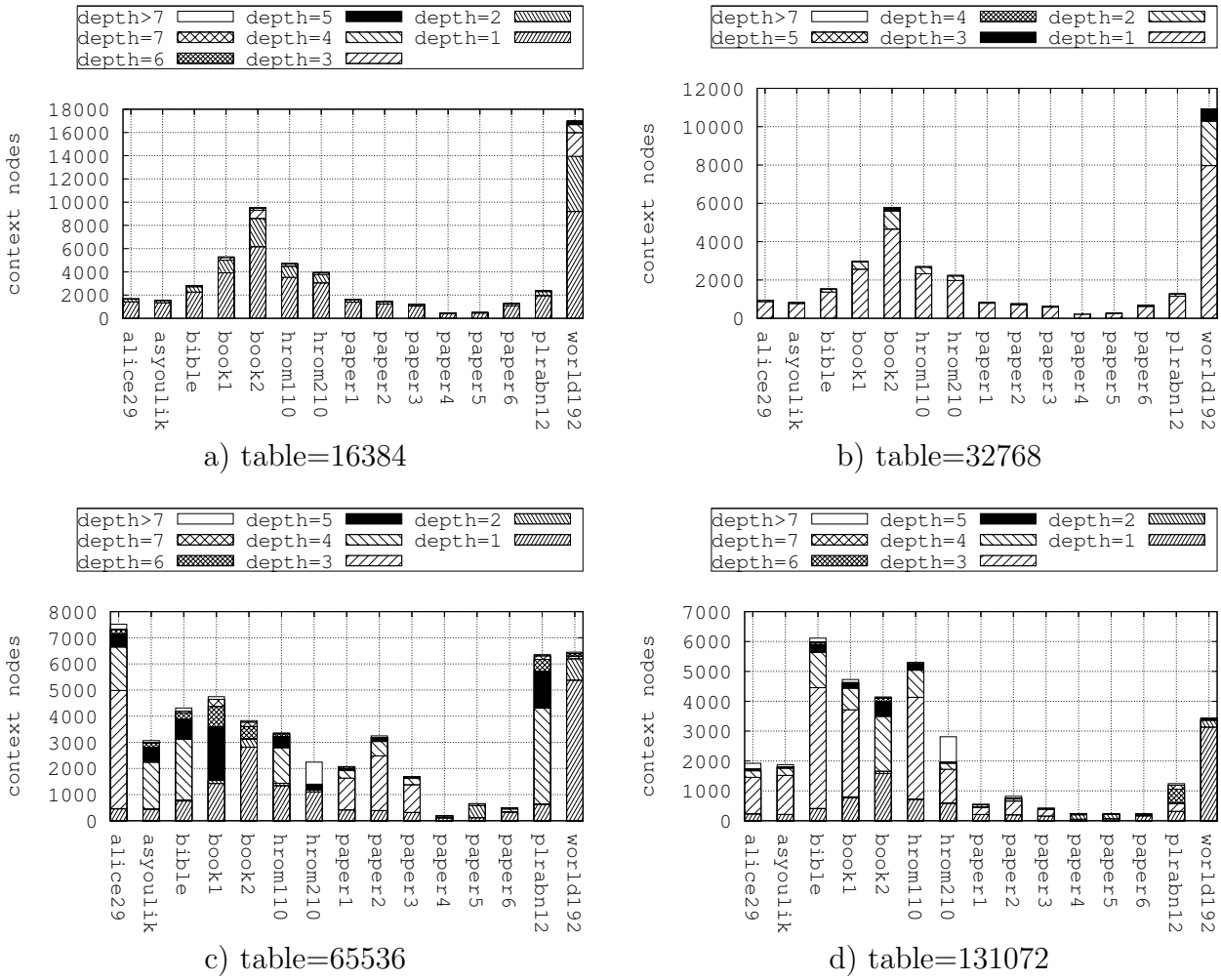


Figure 2.25: Number of collisions categorized by their depth within the data structure for model **order 2** for the table sizes 16384, 32768, 65536, and 131072 (figures a)-d)). The depth=0 collisions are not included as they actually do not reflect a collision but the first entry in the hash table.

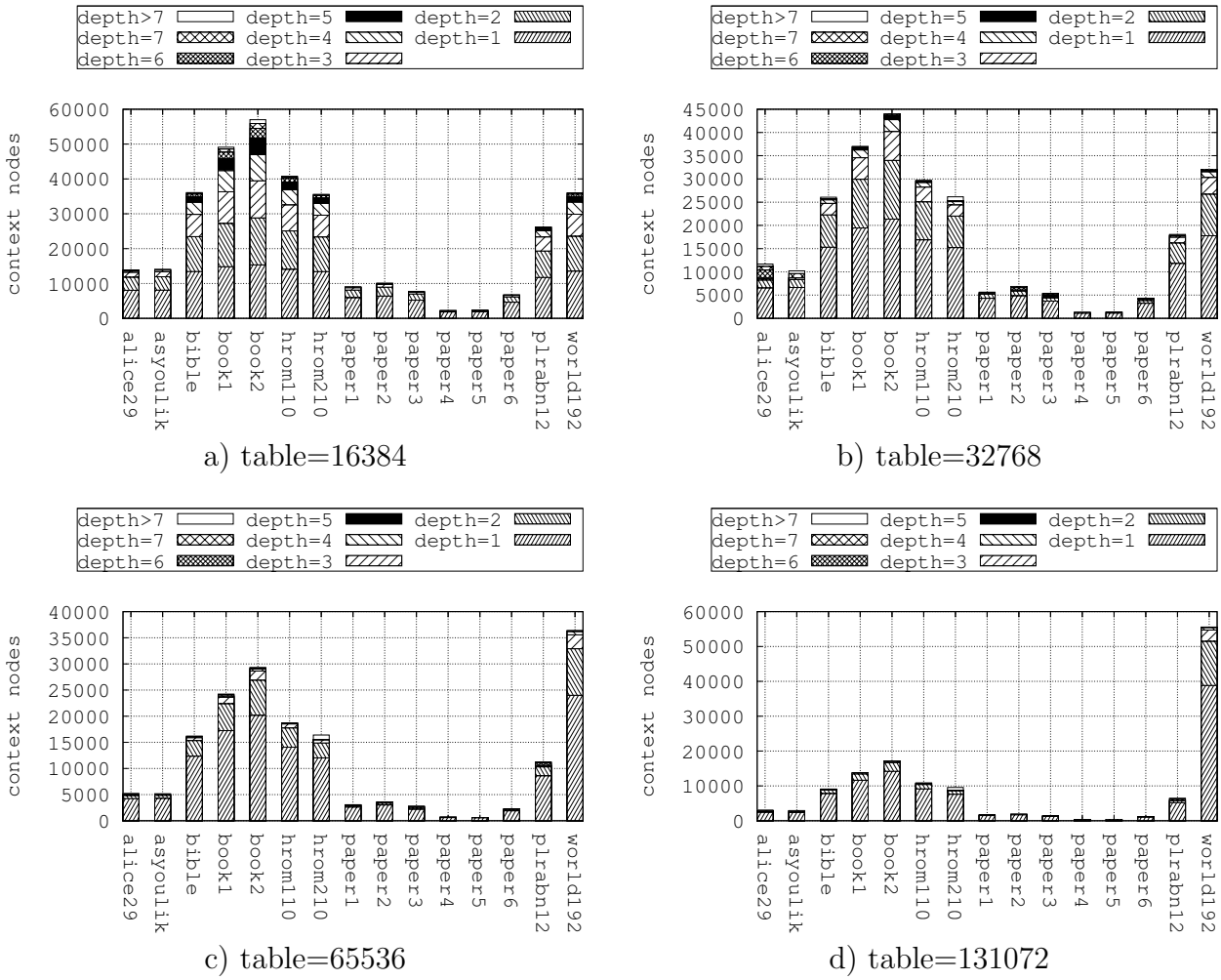


Figure 2.26: Number of collisions categorized by their depth within the data structure for model **order 3** for the table sizes 16384, 32768, 65536, and 131072 (figures a)-d)). A larger table size generally reduces the total number of required collisions, even if the observation for single files is contrary.

distribute the context nodes uniformly over the whole array of hash table elements. In ideal case, there are few very deep collisions. The larger the table size, the smaller the amount of collisions should be. Due to the larger amount of statistical data, a higher order generally requires much more context nodes and results in more collisions.

Figures 2.25 and 2.26 give the number of categorized collisions for order 2 and order 3, respectively. (The plots for order 4 are given in the appendix in figure 5.3 on page 150). Both figures were conducted for the table sizes 16384, 32768, 65536, and 131072 (figures a)-d)). Figure 2.25 b) shows improvements over the table size given in figure a) in that there are less collisions over all training files. For the table sizes 65536 (figure c)) and 131072 (figure d)), however, the number of collisions increases compared to table size 16384 (figure a)) for many of the given training files. The trend that is visible in figures a) and b) is not proceeded in figures c) and d), indicating that a larger hash table size does not necessarily reduce the number of collisions for a lower model order. The required number of collisions is given as 17000, 11000, 7500, and 6000 for the given hash table sizes.

Figure 2.26 shows a more constant trend for the total number of collisions when the hash table size is enlarged. This trend is though not visible for the file *world192*. The required number of collisions is given as approximatively 56000, 45000, 36000, and 55000 for the given hash table sizes in figures a)-d).

From the conducted measurements the conclusion can be drawn that a larger table size generally reduces the total number of required collisions, even if for single files the observation is contrary. For order 2 the total number of collisions is reduced by 35, 56, and 65 % when the larger table sizes 32768, 65536, and 131072 are employed. For order 3 the number of collisions is respectively reduced by 20, 38, and 2 %. The low reduction of 2 % is due to the file *world192*, for which the hash key has difficulties in performance. Excluding the file *world192* the reduction would be 68 %.

It is not possible at this moment to deduce information on the model's quality for the compression performance. The measurements just allow for computation of the required collision items if all the statistical data should be retained. If 10 bytes are allocated for an item, $32768 \cdot 10 + 6000 \cdot 10 = 379$ kBytes would be necessary for the *book2* model using order 2 with table size 32768. Regarding the results it can be concluded that the models exceed the maximum memory requirements that were discussed to be obligate for the low-complexity method to be developed. The measurements are in accordance to the typical statements on PPM, where the models are said to take several MBytes of RAM.

For this memory calculation 10 bytes were assumed to be allocated per table and collision item. As this a very rough assumption, in the next section the collisions are categorized by the length of their key.

2.5.6 Context Nodes Results

In this section the number of context nodes within the data tree categorized by their character length is analyzed. Figure 2.27 depicts the number of context nodes for orders 2 (figure a)) and 3 (figure b)) when there are no memory constraints for the complete array of text files.

For order 2 the number of context nodes varies between 4000 and 30000 nodes for the files *paper4* and the file *world192*. Similarly than in the previous section the file *world192* requires most of the context nodes because it has a very large file length. For order 2 and order 3 it can be observed that the longer context nodes take a larger proportion of the total amount of nodes. This is connected with the larger space of possible keys for higher orders, e.g., a key of

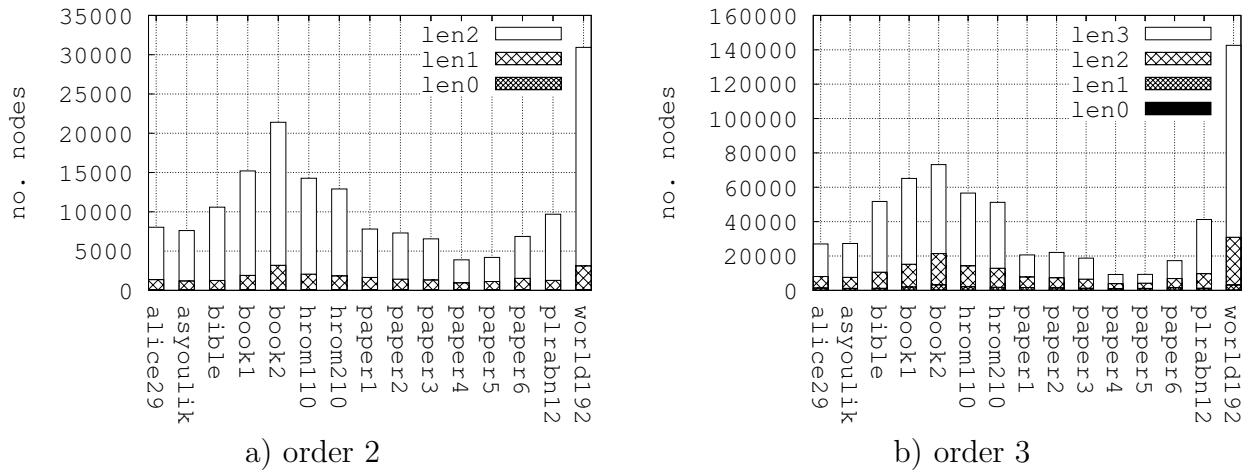


Figure 2.27: Number of context nodes categorized by their length (len2, len1, len0) within the data tree without memory constraints for order 2 (figure a)) and order 3 (figure b)). len0 requires one byte and len3 requires 4 bytes. The figures allow for computation of the required memory size.

length 0 has 256 characters and a key of length 1 has 256^2 possible characters. The amount of statistical data within a text file is not necessarily connected with the file length (as one may assume from the observation of the file *world192*), e.g., the file *book2* with the file length 587 kBytes has 21000 nodes for order 2, whereas the file *book1* with the length 751 kBytes requires only 15000 nodes.

Comparing the amount of context nodes for order 2 and 3, order 3 requires between two (file *paper5*) and five (file *world192*) times more of context nodes. The trend for the required total number of context nodes is similar for both orders with two maxima at the files *book2* and *world192*.

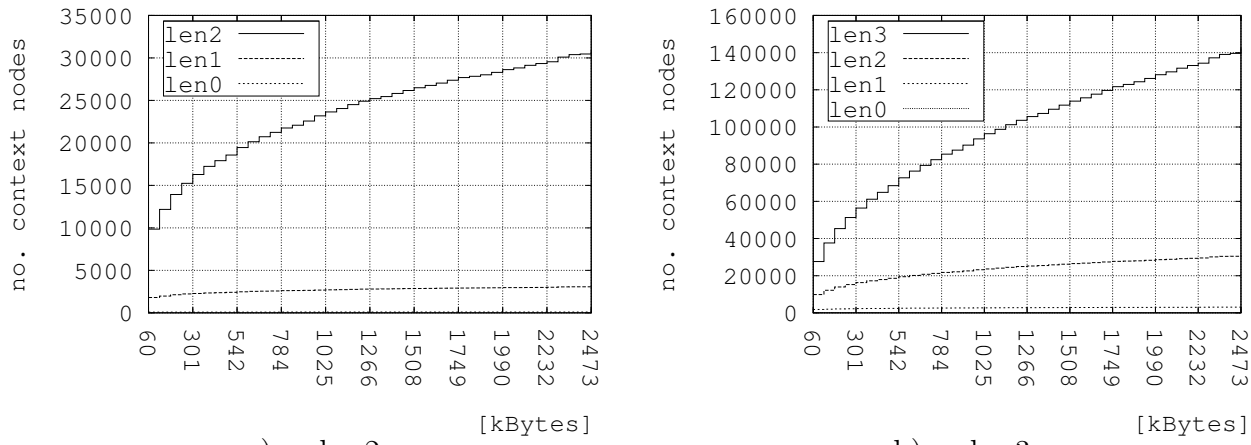
Together with the results of section 2.5.5, it is now possible to estimate the exact amount of memory required to store the statistical data for one file. A context node requires a collision item (allocating two pointers), a list item (including a key pointer, a count variable of at least one byte, and a bitmask of 32 bytes), and the required number of characters for different key lengths. The number of bytes for a context node is thus given as follows:

description	[bytes]	total [bytes]
collision item: 2 pointer	$2 \cdot 4$	8
list item: key pointer, count, bitmask	4+1+32	37
key		length+1
		46 +length

For the file *book2* with table size 16384 and order 2 the calculation is given as follows:

description	[bytes]	total [bytes]
hash table	$16384 \cdot 8$ (collision item)	131072
collisions, see figure 2.25 a)	$9500 \cdot 8$	76000
list items, see figure 2.27 a)	$22500 \cdot 37$	832500
keys, see figure 2.27 a)	19900 len2 and 2600 len1=$19900 \cdot 3 + 2600 \cdot 2$	64900
		1.05 MByte

Thus 1.05 MBytes are necessary for the order 2 *book2* file.



a) order 2 b) order 3

Figure 2.28: Statistical evolution for the file world92 with order 2 (figure a)) and order 3 (figure b)). Statistical data is gathered throughout the entire file, however, the amount of information gathered for the first half of the file is 83 % for order 2 and 71 % for order 3.

Especially the hash table with its pointers each of them consuming four bytes and the bitmask in the list items are very memory intensive. However, the given data structure allows for a relatively fast retrieval of the statistics - in contrast to a data tree, which has to be traversed for most of the nodes to be retrieved. In section 2.5.5, it has been illustrated that the linked lists for resolving the collisions are of limited length. For order 3 and a table size of 32768 most of the collisions were within a depth of 3 to 4, which requires at maximum 4 operations to retrieve an element of such a list.

To design a low-complexity data model using a hash key, the complexity of the given data structure has to be reduced. A possible approach may be to map all the important context nodes to the first entry of the hash table. This may be at the cost of compression performance. Another issue for the low-complexity model is that short messages from 50 to 200 bytes should be compressed. This is not yet realized as the compression generally starts after a larger amount of data has been processed (approximatively 1000 bytes). For the first bytes the compression performs poorly, as the internal model is established throughout the compression. To solve this problem statistical data has to be preloaded.

In the next section the statistical evolution over time for the text files will be analyzed. This shall answer the question if already a subpart of a data file delivers the needed statistics or if information is gathered throughout the compression.

2.5.7 Statistical Evolution over Time

Figure 2.28 depicts the increasing number of context nodes gathered during the compression for all text data files for order 2 (figure a)) and 3 (figure b)). The nodes are categorized by their length. The curves reflect the cumulative total count of nodes: Each curve of context length N is calculated by the sum of the nodes with context length N and all nodes with context length $< N$. Thus, the total number of nodes with context length N is given by the difference of the curve with $len = N$ and $len = N - 1$. If the curves are not rising, the data to be compressed gives known symbols with known context. The given figure shows the evolution for the file *world92*. For the file *bible* the evolution is given for orders 2 and 3 in the appendix in figure 5.4 on page 150.

	a29	asy	bib	bk1	bk2	hr1	hr2	pr1	pr2	pr3	pr4	pr5	pr6	pl1	wld
alice29	2.58	3.92	4.28	3.67	4.19	3.49	3.51	4.46	4.05	4.35	4.54	4.68	4.55	3.73	4.10
asyoulik	4.19	2.62	4.19	3.87	4.10	4.36	4.43	4.60	4.11	4.43	4.56	4.75	4.58	4.31	4.99
bible	3.33	3.16	2.66	3.12	3.43	3.31	3.34	3.81	3.51	3.56	3.74	4.07	3.80	3.08	3.90
book1	3.71	3.62	3.84	2.91	3.69	3.8	3.82	4.07	3.70	3.81	4.04	4.33	4.08	3.71	4.42
book2	4.68	4.57	4.74	4.13	2.85	4.19	4.26	3.62	3.73	3.69	3.69	4.11	3.86	4.43	4.69
hrom110	3.54	3.93	3.97	3.68	3.68	2.84	2.91	3.93	3.79	3.74	3.94	4.12	3.97	3.58	3.62
hrom210	3.54	3.91	3.93	3.65	3.67	2.91	2.82	3.91	3.77	3.72	3.93	4.11	3.97	3.54	3.59
paper1	5.24	5.07	5.30	4.66	3.41	4.59	4.67	2.48	3.84	3.91	4.35	4.24	3.92	4.86	5.05
paper2	4.05	3.95	4.15	3.60	3.28	3.84	3.86	3.41	2.61	3.20	3.67	3.92	3.76	3.97	4.43
paper3	4.38	4.25	4.42	3.83	3.33	3.93	3.95	3.50	3.27	2.70	3.77	3.99	3.86	4.14	4.33
paper4	4.36	4.32	4.43	3.83	3.31	3.86	3.89	3.55	3.52	3.45	2.47	3.79	3.60	4.16	4.31
paper5	5.11	4.97	5.15	4.60	3.78	4.59	4.66	3.88	4.08	4.04	4.18	2.35	3.74	4.84	5.02
paper6	5.35	5.21	5.44	4.90	3.68	4.78	4.90	3.86	4.16	4.31	4.49	4.21	2.43	4.99	5.30
plrabn12	3.64	3.84	3.79	3.68	3.98	3.50	3.51	4.23	3.97	3.99	4.29	4.54	4.33	2.79	3.87
world192	4.94	5.57	5.95	5.01	4.82	4.02	4.00	4.96	5.11	5.24	5.38	5.28	5.20	4.74	3.04
avg	4.18	4.20	4.42	3.95	**3.68**	**3.87**	3.91	3.89	**3.82**	3.88	4.07	4.17	3.98	4.06	4.32

Table 2.5: Static context modeling for **order 2**. For each training file (first horizontal row) the compression rate for each of the files (listed in the first vertical column) is measured. The statistics are not updated throughout the compression. The files book2, paper2 and hrom110 give the best performance ranging from 3.68 to 3.87 bpb.

In general, the curves for all files rise during the compression. The nodes with less context characters rise more slowly and sometimes tend to be constant, as their smaller context space is already filled. The measurements were conducted to answer the question whether it makes sense to employ the complete data file for construction of a static model to be preloaded. The word pool for a large text file was expected to be exhausted or to only rise very slowly when a large number of words have already been processed. However, as figure 2.28 illustrates, the vocabulary keeps rising till the end of the files. The percentage of gathered information is larger for the first half of the file, i.e., 83 % for order 2 and 71 % for order 3. The figure also verifies that the model is not flushed at any time as the allocated memory is sufficient.

As a result, it is assumed in the following that the complete text files deliver useful statistics and a further analysis of the statistical context development within the data files is not performed. In the next section the statistical information of one training file is used to compress one of the other files. This shall give insights about the usage of a preloaded model for compression of short messages.

2.5.8 Static Context Modeling

In this section measurements for *static* context modeling are conducted for all the given text files. There are two main differences between adaptive and static context modeling:

1. In static context modeling, the statistical data of a training file is preloaded before the compression starts. This assures that the compression can start from the first bytes of a short text sequence.

2. The statistics of the context model are not updated throughout the compression. This first may allow for a low-complexity compression scheme. Second, the update of the statistics might not give significant compression improvements, as only a short message

	a29	asy	bib	bk1	bk2	hr1	hr2	pr1	pr2	pr3	pr4	pr5	pr6	pl1	wld
alice29	1.94	3.78	4.14	3.34	4.03	3.31	3.34	4.43	3.92	4.28	4.61	4.72	4.54	3.48	3.96
asyoulik	4.12	2.09	3.95	3.64	4.04	4.20	4.32	4.62	4.09	4.47	4.68	4.83	4.65	4.08	4.88
bible	3.28	2.96	2.03	2.85	3.38	3.09	3.18	3.86	3.49	3.59	3.86	4.16	3.92	2.80	3.75
book1	3.56	3.47	3.64	2.30	3.50	3.60	3.65	4.01	3.58	3.76	4.10	4.39	4.09	3.48	4.24
book2	4.59	4.45	4.67	3.77	2.00	3.93	4.02	3.41	3.51	3.50	3.95	4.05	3.73	4.24	4.40
hrom110	3.42	3.79	3.79	3.33	3.44	2.14	2.36	3.82	3.64	3.64	3.96	4.14	3.93	3.28	3.26
hrom210	3.40	3.76	3.72	3.29	3.45	2.32	2.10	3.81	3.63	3.63	3.95	4.14	3.94	3.23	3.22
paper1	5.12	4.96	5.24	4.37	2.95	4.35	4.43	1.69	3.60	3.70	4.25	4.16	3.77	4.69	4.75
paper2	3.95	3.83	4.04	3.26	2.82	3.57	3.60	3.16	1.90	2.92	3.62	3.88	3.63	3.79	4.09
paper3	4.31	4.12	4.34	3.45	2.87	3.61	3.67	3.25	2.95	1.92	3.65	3.93	3.71	3.94	4.01
paper4	4.3	4.21	4.34	3.51	2.79	3.56	3.59	3.24	3.26	3.14	1.64	3.68	3.4	3.94	3.91
paper5	5.02	4.87	5.14	4.31	3.37	4.37	4.46	3.58	3.86	3.84	4.12	1.53	3.52	4.67	4.75
paper6	5.28	5.09	5.35	4.62	3.30	4.54	4.67	3.59	3.88	4.14	4.37	4.13	1.61	4.85	5.01
plrabn12	3.59	3.65	3.57	3.44	3.95	3.30	3.34	4.27	3.97	4.02	4.39	4.64	4.42	2.24	3.78
world192	5.02	5.57	5.99	4.95	4.78	3.92	3.94	4.97	5.09	5.19	5.36	5.29	5.20	4.71	2.06
avg	4.07	4.05	4.27	**3.63**	**3.38**	**3.59**	3.65	3.72	**3.63**	3.72	4.04	4.12	3.87	3.83	4.01

Table 2.6: Static context modeling for **order 3**. The training file book2 now gives an average compression of 3.38 bpb, which is an improvement of 0.3 bpb compared to order 2. The files hrom110, book1 and paper2 follow in compression performances from 3.59 to 3.63 bpb. For these files, order 3 significantly improves the compression performance in the range of 0.2 to 0.3 bpb.

	a29	asy	bib	bk1	bk2	hr1	hr2	pr1	pr2	pr3	pr4	pr5	pr6	pl1	wld
alice29	1.62	3.89	4.16	3.25	4.03	3.32	3.35	4.54	4.01	4.40	4.74	4.82	4.65	3.56	3.98
asyoulik	4.22	1.74	3.91	3.62	4.12	4.23	4.36	4.74	4.23	4.60	4.79	4.94	4.74	4.11	4.89
bible	3.39	3.01	1.67	2.8	3.42	3.07	3.19	4.00	3.65	3.73	4.00	4.31	4.05	2.75	3.74
book1	3.62	3.56	3.61	1.94	3.48	3.58	3.63	4.14	3.68	3.87	4.23	4.51	4.19	3.50	4.21
book2	4.65	4.55	4.71	3.69	1.58	3.86	3.96	3.45	3.53	3.55	4.04	4.14	3.79	4.26	4.28
hrom110	3.49	3.87	3.75	3.23	3.39	1.73	2.14	3.9	3.70	3.71	4.08	4.25	4.00	3.26	3.13
hrom210	3.47	3.84	3.66	3.20	3.40	2.09	1.70	3.90	3.69	3.71	4.07	4.25	4.01	3.21	3.09
paper1	5.17	5.04	5.26	4.28	2.82	4.30	4.38	1.32	3.61	3.75	4.34	4.24	3.81	4.68	4.61
paper2	4.00	3.94	4.06	3.16	2.69	3.51	3.53	3.21	1.53	2.96	3.73	3.99	3.69	3.82	3.91
paper3	4.35	4.19	4.31	3.31	2.73	3.52	3.57	3.29	2.96	1.49	3.75	4.04	3.77	3.94	3.85
paper4	4.36	4.28	4.39	3.41	2.65	3.52	3.54	3.25	3.26	3.17	1.29	3.79	3.43	3.94	3.79
paper5	5.08	4.99	5.19	4.21	3.27	4.34	4.43	3.60	3.88	3.92	4.22	1.23	3.57	4.69	4.62
paper6	5.33	5.15	5.37	4.53	3.19	4.49	4.62	3.61	3.90	4.19	4.45	4.21	1.27	4.86	4.86
plrabn12	3.73	3.77	3.56	3.44	4.02	3.34	3.39	4.41	4.12	4.16	4.52	4.75	4.53	1.93	3.86
world192	5.06	5.69	5.98	4.91	4.76	3.89	3.91	5.03	5.14	5.24	5.41	5.33	5.25	4.82	1.46
avg	4.11	4.10	4.24	**3.54**	**3.31**	**3.52**	3.58	3.76	3.66	3.77	4.12	4.19	3.92	3.83	3.89

Table 2.7: Static context modeling for **order 4**. Similarly than for order 2 and order 3, book2 gives the best compression performance of 3.31 bpb, which is only a small improvement of 0.07 bpb compared to order 3. The small improvements are in accordance with the measurements for adaptive context modeling in section 2.5.3, where order 4 only gave small improvements over order 3.

is to be compressed, which gives very little additional statistical information regarding the large amount of training data that is preloaded.

The measurements shall give insights for designing a low-complexity compression scheme that preloads a statistical model before the compression of a short text sequence starts. As a primary step the complete text files instead of short text sequences are compressed to find a suitable training file among the given files. The measurements shall also reveal the possible compression rates with static context modeling.

The figures 2.5, 2.6, and 2.7 give the compression results for static modeling for the orders 2,3, and 4, respectively. In the vertical columns, the compression performance using the appendant training file listed in the first row, is given for all the files listed in the first vertical column. In the last row the average compression is calculated for a given training file. For all orders (orders 2, 3, 4), the training file *book2* gives the best compression that is measured as 3.68, 3.38, and 3.31 bpb, respectively. As order 4 gives only minor improvements similarly than in section 2.5.3 for the adaptive measurements, it is skipped in the following considerations for the low-complexity scheme.

The measurements in this section give a rough estimate of the possible compression rate of 3.38 bpb for the low-complexity order 3 scheme to be developed. The value of **3.38** is considered as a requirement for the compression of short messages. The open question is if this requirement can be fulfilled as the here employed models do require too much memory and a model simplification may result in a lower compression gain.

2.5.9 Summary of the Evaluation

The idea of the evaluation in this section was to get insights for the development of a system for low-complexity short message compression.

The results in section 2.5.2 indicate that a statistical count variable with the size of one byte is sufficient. In the implementation an integer variable (four bytes) was used for the symbol count to allow for these measurements. The low-complexity scheme should use only one byte for each of the symbol counts.

The measurements in section 2.5.3 give the compression results for adaptive context modeling with limited memory. The memory is flushed when the maximum number of collisions is attained. The measurements indicate that the compression is scalable using different memory sizes for the implemented statistical model. The parameters concerned the hash table size and the maximum number of collisions. The scalability feature should be retained in the low-complexity model.

In section 2.5.4 it was verified that the employed hash key performs reasonably. The hash key distributed the required keys equally over the whole array of hash table elements for all measured positions (depth) in the lists for resolving collisions. Even if the hash does not perform ideally, it is also employed for the low-complexity scheme.

The sections 2.5.5 and 2.5.6 allow for a precise computation of the required memory. It is shown that the typical requirements of several MBytes memory for PPM are also given with the here introduced data structure. For the low-complexity statistical model, this structure has to be simplified in such a way that no lists of collisions are necessary any more. Then the pointers in the collision items would not be required any more and the statistical data could be stored in each of the hash table entries. Another large amount of memory is consumed by the bitmasks each of them allocating 32 bytes to signal the existing context nodes. (The

information about the context nodes is required to compute a symbol's probability.) In this case a computing technique should be developed so that the bitmasks can be discarded.

In section 2.5.7 it is decided that a complete training file (and not a subpart of a file) shall be preloaded into the model to allow for short message compression. The final method shall use a static context model where the statistics are not updated throughout the compression. The adaptive technique requires more complexity and may not give significant improvements as the statistics of the preloaded model can not be influenced by a short text sequence.

In section 2.5.8 compression results are given for static context modeling for each of the training files. The measurements indicate that order 4 does not make sense for a static low-memory model, as the compression rate is not improved significantly. The rate 3.38 bpb is considered to be a rough value for the final compression rate to be achieved by the method to be developed.

In the next section these results shall be employed to develop a novel compression scheme for short messages with very low memory requirements.

2.6 A Novel Context and Data Model for Short Message Compression

2.6.1 Introduction

In this section a novel low-complexity compression scheme for short messages is detailed. (The scheme was originally published in [45, 46] and resulted into the international and American patent applications in [47, 48].) The novel scheme is designed to fulfill the requirements given in the introduction in section 1 while the insights from the evaluation in section 2.5 shall be exploited. The techniques of the novel scheme including the resulting features are given as follows:

a)	Use a function that estimates the context of a given key. To realize this the complete possible context of a symbol (255 symbols) has to be traversed.	⇒	The memory requirements are reduced, as there is no need to store a bitmask any more. Functions to maintain the bitmask can be left as well.
b)	Usage of a hash table with the key function verified in section 2.5. The collisions of the data model are discarded with the cost of lower compression performance. From the original list of collisions, the collision with the maximum statistical count is retained. The size of the hash table allows for a trade-off between compression and memory requirements.	⇒	Reduction of memory requirements and complexity, as the statistical data can be stored in the hash table elements themselves. The compression is scalable.
c)	Use a smart approach so that instead of the strings for a key only a code with the length of one byte has to be stored. This code is employed to verify that the correct string in the data model was found.	⇒	Tremendous reduction of memory requirements, as for each context node only two bytes - one for the statistical count, and another one for the code, has to be stored. As it will be demonstrated, a data model of 32 kBytes already cuts a short message in half.

d) The hash table is preloaded with the statistical data of a text file. Throughout the compression the model is not updated, as a short data sequence is statistically not able to meaningfully update a large data model. (Updating the model for a series of messages would furthermore require the sender to keep track of different updates and to assign them to the correspondent receiver.) ⇒ The compression already starts for very short sequences from 50 bytes. As the model is not updated, the complexity of the implementation is reduced. The complete compression scheme can be realized with a few pages of source code.

In the next sections these techniques are explained in more detail. In subsection 2.6.2.1 the employed context model is detailed. The main difference to the context model in PPM is that the existing context is explored on the fly (technique a)). In subsection 2.6.2.2 the novel data model is detailed (features b) and c)). Instead of storing the characters of the keys in the data model - as it was done in previous works, only a code is stored in the hash table. In subsection 2.6.2.3 the methodology for building a low-complexity data model is explained together with the developed software. For the low-complexity scheme a separate implementation in C was realized. However, for building the low-complexity model the primary and more complex implementation is employed. In subsection 2.6.2.4 a technique for the compression of the data model is introduced. This may allow a device to maintain several different data models. In section 2.6.3 the performance results for the novel scheme are given, and in the last section the findings are summarized.

2.6.2 Short Message Compression Scheme

2.6.2.1 The Context Model

In this section the difference between a standard PPM context model and the context model employed for the low-complexity coder is explained. Specifically, the low-complexity coder employs a novel data model that requires a different concept for the calculation of the statistics. A data model is a part of a context model, as illustrated in figure 2.29.

In the encoding process, the context model generally estimates the left, the right, and the total count. These counts are estimated by traversing the probability line of the context of a given symbol. To estimate the left probability, all symbols that are before the symbol to be coded have to be traversed. The context model adds the single counts of the symbols to calculate the left probability. The right probability can then be calculated if a total count is available. In the reference implementation the right count is estimated on the fly to spare the memory for the total count.

Concerning the decoding procedure the decoder generally searches the position of the symbol to be decoded on a given probability line. It finds the symbol by comparing the cumulative count with the number decoded by the arithmetic decoder. When the cumulative count exceeds the given number, the symbol is decoded while also the left and the right count is found. Now the total count has to be computed to allow the arithmetic coder to decode the next number. This is performed similarly as with the encoder.

The encoder and decoder both need to traverse the context of a symbol - that are all the symbols that are located on the same probability line of the given symbol. (The symbols on a probability line all have the same preceding symbols, i.e., they have the same context.)

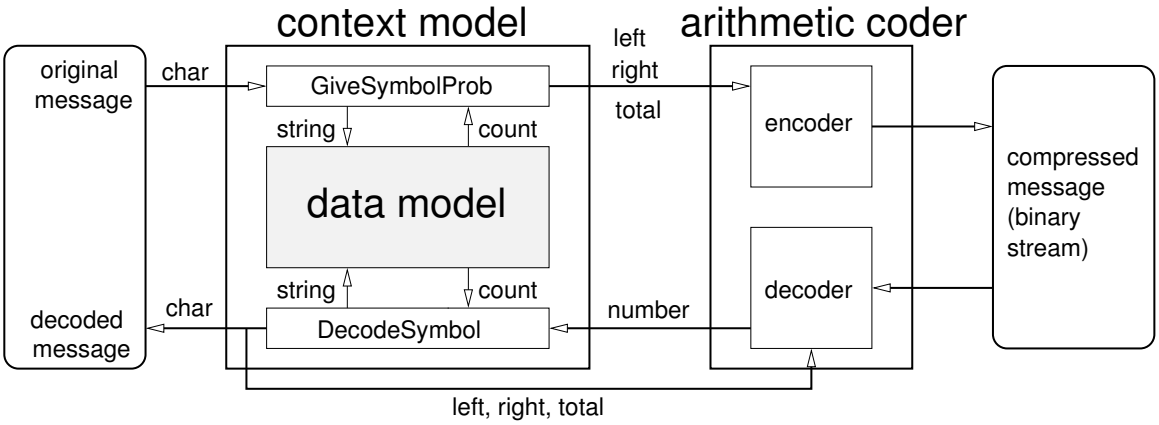


Figure 2.29: General coding procedure with a context model and a data model. A context model calculates the symbol probabilities for the arithmetic coder. The data model is part of the context model and delivers a statistical count for every string it is queried. The low-complexity coder uses a data model where the words themselves are not stored. Instead, a parity byte assures that collisions are detected.

Generally a technique is employed to signal the encoder/decoder, which symbols in the context are present. In the primary implementation, the bitmask was introduced for this purpose.

The novel low-complexity coder instead uses a function, which traverses all possible symbols in the context including the symbols that are not on the probability line. The memory requirements are hereby reduced by the cost of computational time. However, as the low-complexity coder is designed for short text sequences, the coding procedure is finished by a reasonable time. (As it will be shown in a later section, coding and decoding times are in the range of a few seconds on a mobile phone.)

The data model is part of a context model and delivers a statistical count for each given string - which represents a symbol with its context. As the context model uses a function for estimating the context of a symbol, the data model is heavily employed. When a queried symbol with a given context is not on the probability line, the data model has to signal this. In the next section the technique of the data model is explained.

2.6.2.2 A novel Data Model

The data model builds the core of the low-complexity coder. Its function is to provide a statistical count (one byte) for each string it is queried. The model shall be realized with a hash table where collisions are not resolved.

As each hash table element refers to a specific key or string, the general way to realize such a model would be to append the appropriate string to the hash table element. If the hash function maps a key to a hash table element, the appended string could be compared with the key. In case the key equals the stored string, the statistical count which is stored together with the string, could be returned. If the key does not equal the stored string, a zero count could be returned to signal that the key is not in the model. This kind of realization seems to make sense, as a data model generally needs to store the keys for which it supplies the appropriate statistical information.

In contrast to this approach, the low-complexity data model uses a different technique. The main idea of the novel model is that the coding process does only need the statistical count and not the words itself. Therefore a model is designed where the words themselves are not stored but only the statistical count. A collision detection technique is integrated by using a

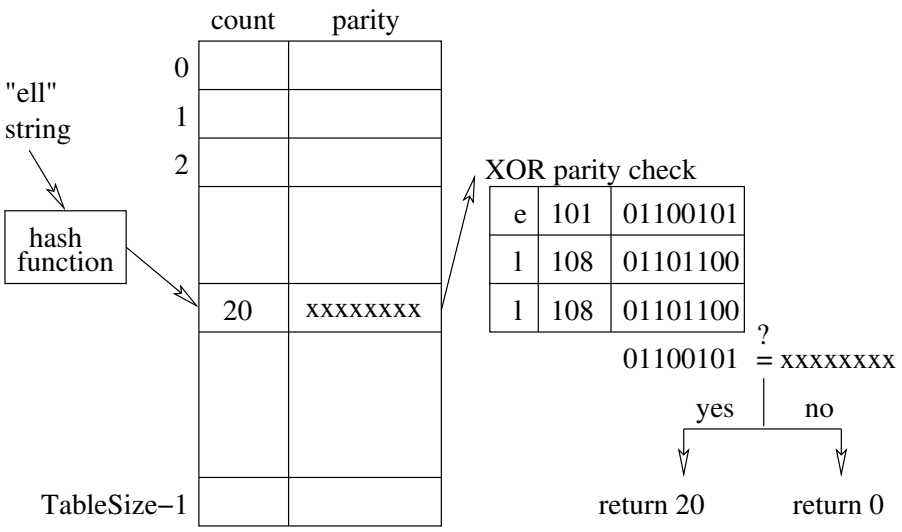


Figure 2.30: The novel data model. For each queried string a statistical count is returned. If the hash function maps a key to a wrong hash table entry, a collision is detected and a zero count is returned to signal that the string is not in the model. Each of the hash table entries stores a statistical count and a parity byte. The parity byte is compared with the parity of the key that is given to the model. If both parities are the same, the statistical count is returned. Otherwise a zero count signals that the key is not in the model.

parity check that requires one byte for each word of arbitrary length.

Figure 2.30 illustrates the parts of the employed data model. The model consists of a hash function, a hash table, and a parity check function. Each hash table entry contains two bytes, a statistical count and a parity check byte. The model works as follows. The hash function maps the input key to an entry of the hash table. Then the parity of the input key is calculated by an exclusive OR (XOR) operation and compared with the parity of the appropriate hash table key. If both parities are the same, the statistical count of the table entry is returned. If the both parities differ, the model returns a zero count thus signaling that the key is not present. The parity check is a collision detection technique. The functionality of the data model is almost the same as with a more complex model using a standard hashing technique, where the collisions are not resolved. A very small loss in compression can result from wrong parity checks, but decoding of the message would still be correct.

The parity check is essential as during the encoding/decoding process a large amount of strings is queried for the on the fly estimation of the symbol statistics. The compression technique even works without this parity check, however, the compression performance would be significantly deteriorated.

The hash table size is a variable parameter to be set by the user. It can control the amount of keys that are not in the model. A large hash table size will result in few collisions and allows for a similar behavior than a general data model with no memory constraints. In the next section it is explained how the statistical data for the model is gathered and loaded into the data structure.

2.6.2.3 Building the Data Model

In the previous section the functionality of the low-complexity data model was described. In this section the methodology to fill the data model with statistical data is given.

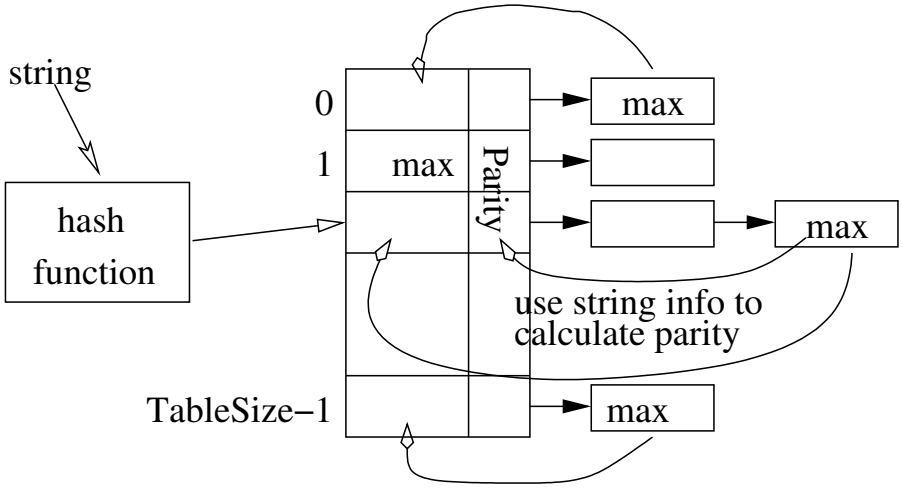


Figure 2.31: Principle of deleting collisions. The primary, more complex data structure gathers the statistical data. When the compression is finished, the lists of collisions are inspected for the elements with the maximum counts. These elements are retained and stored together with the computed parity bytes in a file to be preloaded into the low-complexity data model before the compression of short messages starts.

The data model of the low-complexity coder is different from the hash table with collision resolving techniques in the primary PPM reference implementation. Thus there exists an own implementation of the novel coder. (To allow the coder to be applied to mobile devices and platforms, it is implemented in C and not in C++ as the PPM reference implementation). To gather the statistical data for the novel coder, however, the more complex reference implementation has to be employed, see [42] for a manual. This has to be done as for building the low-complexity model, first the complete data of a training file has to be collected, from which all collisions expect the statistically most important ones have to be deleted. Then the data is stored in a file to be loaded by the low-complexity coder to allow for static context modeling and compression of short text.

The method for deleting the collisions is illustrated in figure 2.31. First the training file is compressed using the PPM reference implementation, where the compression is just done to gather the statistics and not to compress the file. When the compression is done, the linked lists are inspected for the collision with the maximum count. This collision is copied to the first entry of the hash table. Then an XOR parity check is generated for the string data of this element. Finally, all the elements of the hash table whose count do not equal zero are written into a file, where for each element a statistical count and a parity byte is stored. In the next section a compression technique for this kind of data is introduced.

2.6.2.4 Compression of the Data Model

The low-complexity data model is intended to be located on a mobile device, e.g., a cell phone or a wireless sensor. As the flash memory of such a device is limited, it makes sense to design a compression technique for the statistical data. Such a technique may also allow a mobile device to handle a set of data models, e.g., for different types of sensor data or to support different languages. In this case, the encoder could signal the decoder which model to use.

A simple yet effective method for compression of the statistical data can be achieved due to the fact that even with a good hash, many of the elements of the hash table are empty and not employed for the statistical model. An empty element is defined by a zero count. If an

file	RAM [kB]	size [byte]
o2tb-32	32	30278
o2tb-64	64	47606
o2tb128	128	63060
o2tb256	256	73260

a) order 2

file	RAM [kB]	size [byte]
o3tb-32	32	32762
o3tb-64	64	64752
o3tb128	128	117106
o3tb256	256	176490

b) order 3

Figure 2.32: Size of the compressed low-complexity data models from the training file book2 for orders 2 and 3. The RAM-size gives the original size of the model. A model of 32 kByte has 16384 hash table elements each of them storing a count and a parity byte. The proposed compression technique can compress the larger low-complexity models more effectively. With such a technique, a mobile device can maintain a set of different data models on its flash memory.

element is empty, there is no need for storage of a parity. To store the counts and the parities the hash table is traversed from its first to the last element. For each element, the count and the parity is sequentially stored. In case of a zero count, a zero byte and a byte for the total number of zero elements in a stream is written on disk. Thereby, large streams of zero elements are stored by just two bytes. Typically, there exist many streams of zero elements in the data model. The maximum stream size that can be denoted is 256 empty elements.

Figure 2.32 gives the size of different compressed data models in bytes for the orders 2 in table a) and 3 in table b). For the measurements, models with the hash table lengths 16384, 32768, 65536, and 131072 elements are employed. As each element requires two bytes - the symbol count and the parity, the statistical models allocate 32, 64, 128, and 256 kBytes of RAM memory for each respective table length. The models are constructed from the text file *book2* from the Calgary corpus. The *book2* training file was selected as in section 2.5 it gave the best compression results among all other text files.

2.6.3 Results

2.6.3.1 Statistical Features

This section gives some results indicating the ability of the proposed low-complexity model to retain the statistical data for different memory settings. The required memory for the models varies from 32 to 256 kBytes (denoted as tb-32, tb-64, tb-128, and tb-256). As a training file the *book2* text is selected. (The results for all the other text files concerning the categorized number of context nodes were very similar and thus are given in the appendix in figure 5.5 on page 151 for order 2 and in figure 5.6 on page 152 for order 3.) Figure 2.33 a) compares the number of context nodes of the limited models with the model without memory constraints, which is typical for PPM. The unbounded model is denoted as *tbxxx*. The context lengths *len0* to *len3* refer to the order of the key of a context node (for instance, len1 refers to a key of 2 bytes and the model order 1). The models o2tb-32 and o2tb-64 preserve 56 and 70 % of the original model space, which is given as 21400 nodes. The models o2tb128 and o2tb256 tend to preserve the complete space of occurring context nodes, and therefore are expected to result into good compression performance. For order 3, the space of possible context nodes is given as 73191. The models o3tb-32 and o3tb-64 preserve just 22 and 41 % of the original model space. The models o3tb128 and o3tb256 preserve 60% and 77%, respectively. For order 2 the

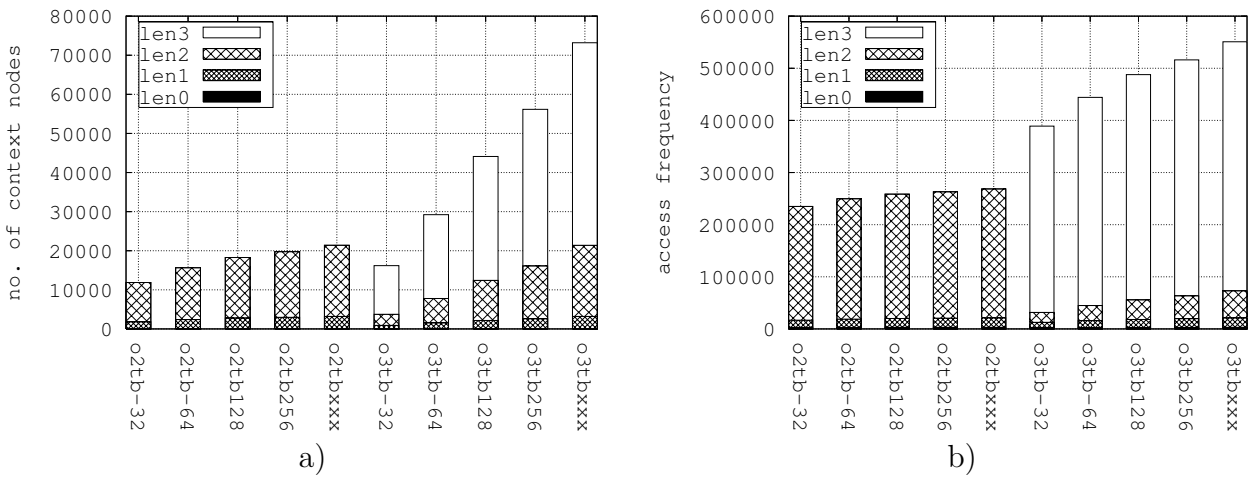


Figure 2.33: Statistical features of the low-complexity data models compared to the unbounded model (denoted as o2tbxxx/o3tbxxx) for the book2 training file. Figure a) classifies the retained nodes by their context length. The figure illustrates the scalability feature of the proposed technique. For the order 2 models the differences to the unbounded model are much smaller than for the order 3 model. Figure b) illustrates the sum of the statistical counts for the context nodes categorized by their length. For order 2 the total count stays almost constant, indicating that even the smaller models may achieve a similar compression than the unbounded model.

percentage of retained nodes is higher than for order 3. Anyway, for both orders the scalability feature is visible.

A low-complexity data model consists of a subspace of the original data model, which should contain the context nodes that are of statistical importance. An indicator to assess the statistical quality of the retained nodes may be the frequency each node is accessed - that is, the sum of the node counts, which is illustrated in figure 2.33 b). In contrast to the order 3 models, the order 2 models give a relatively constant access frequency over the different table sizes. This indicates that even the smaller order 2 models will give good compression results, as the statistically more weighted data is preserved. This is not the case for the order 3 models, however, for this order the rising trend of the bars is smaller than in figure a). This similarly indicates that also for order three the model tries to keep the important data.

2.6.3.2 Compression of Text with Static Modeling

In this section the compression performance for the introduced low-complexity data model for long text files is given. The analysis shall reveal how the compression is deteriorated compared to the usage of the original and complete data of the given training file. The files *paper2* and *book2* were selected to supply the statistical data for the compression of the complete set of text files. Results for the other files providing the statistical data are given for order 2 in the appendix in figures 5.7 and 5.8 on page 153 ff., and for order 3 in figures 5.9 and 5.10 on page 155 ff.. The model sizes were varied from 32 to 256 kByte (denoted as tb32, tb64, tb128, and tb256). The unlimited model file is denoted as PPMc. The measurements for each curve were conducted by preloading the relevant model and then compressing each of the text files while the loaded statistics were not updated (static modeling).

Figure 2.34 illustrates the performance of the low-complexity models with training data from the file *paper2* for order 2 (figure a)) and order 3 (figure b)). The figure shows that the

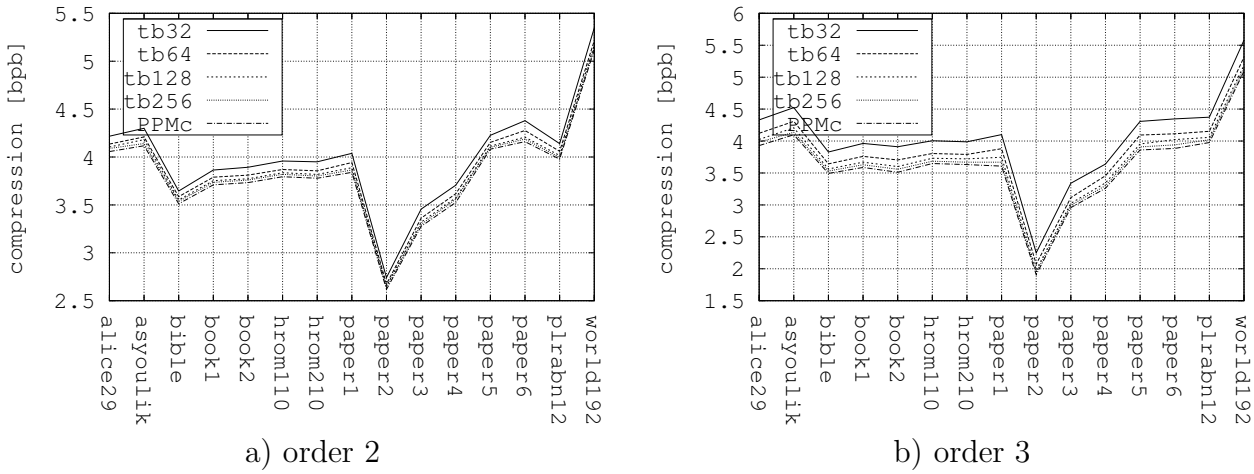


Figure 2.34: Results for compression of long text files using the low-complexity models with training data from the file **paper2**. The models were not updated throughout the compression. Figure a) and b) were conducted for orders 2 and 3, respectively. The size of the models varies from 32 to 256 kByte. The larger models perform very similar than the model without memory constraints (denoted as PPMc) with the complete data. The measurements indicate the potential of the low-complexity models to compress short messages.

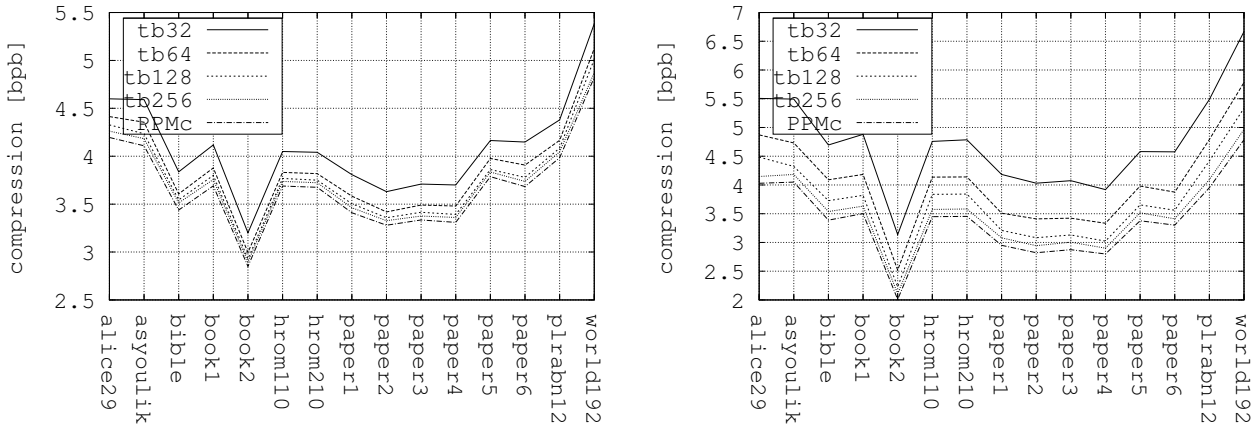
performance can be very similar than with the original data model, especially for the larger models. The largest improvement is achieved by switching from the 32 kByte to the 64 kByte model, i.e., 0.13 bpb and 0.25 bpb for the order 2 and 3 model, respectively.

Figure 2.35 illustrates the compression performance for the static low-complexity models with training data from the file *book2*. The smaller models perform worse than the smaller models with training data from *paper2* (figure 2.34). The improvement from doubling the model size from 32 kByte to 64 kByte is much larger, i.e., 0.24 bpb and 0.5 bpb for the order 2 and 3 models, respectively. The average compression for the largest model (indicated as PPMc), however, is better than with the *paper2* models, see section 2.5.8. A conclusion can be that it is advisable to choose models with less context nodes if only little memory is available. In the next section the *book2* model is employed for compression of short text.

2.6.3.3 Compression of Short Text

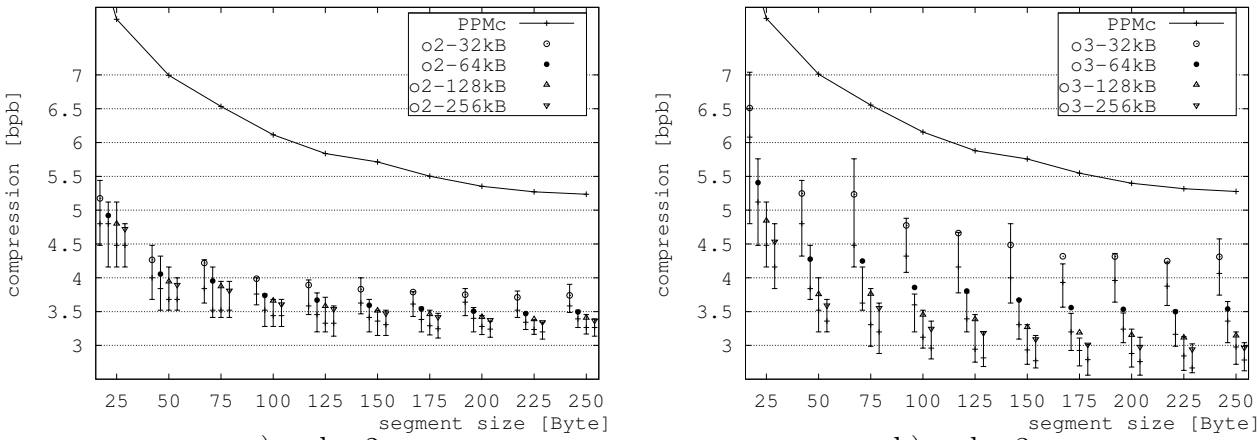
In the previous section the compression performance for the proposed low-complexity data models was evaluated for long text files. In this section the compression performance for these models is evaluated using short text messages. Figure 2.36 gives statistical descriptors for the compression rates:

- The measurements were conducted for the model orders 2 (figure a) and 3 (figure b) using the statistical models with 32, 64, 128, and 256 kBytes.
- The file *book2* was used to train the statistical models. (The model is not updated while the short messages are compressed.)
- Short messages from the file *paper4*, here denoted as *segments*, were compressed. The size of the segments (*segment size*) ranges from 25 to 250 bytes. For illustrative reasons, the results of the four different models for a fixed segment size are slightly displaced. Each set of the four measurements relates to the same segment size.



a) order 2 b) order 3

Figure 2.35: Results for compression of long text files using the low-complexity models with training data from the file **book2**. Similarly than in figure 2.34, the largest improvement in performance results from switching from the 32 kByte to the 64 kByte model. The improvement from doubling the model size is much larger than when using data from the file paper2. This results from the larger number of context nodes (see figure 2.27) in book2. The compression with unlimited models is best for book2, however, if only little memory is available, the paper2 model can give better results.



a) order 2 b) order 3

Figure 2.36: Compression performance with the novel low-complexity coder for short messages for order 2 (figure a)) and order 3 (figure b)). The models require 32, 64, 128, and 256 kByte of memory. PPMc without a preloaded model and the 32 kByte order 3 model fail for short messages. The other models indicate the ability of the own coder to provide a scalable trade-off between compression efficiency, memory and computational requirements. The results are achieved by preloading a model into a specific data structure that retains the statistically important context nodes. The data structure only maintains two bytes for each node and not the string data itself.

- For each given segment size and model, N=30 measurements were conducted to measure the compression. For each of the 30 measurements, the mean (illustrated by the $\circ, \bullet, \triangle$, and the $\bigtriangledown$ symbols), the first quartile (=25th percentile, which cuts of the lowest 25 % of the data), the second quartile (=50th percentile or median, which cuts the data set in half), and the third quartile (=75th percentile, which cuts of the highest 25 % of the data) are computed. The quartiles are given as bars on a vertical line.
- For comparison the compression of the regular PPMc encoder is included.

Figure a) shows that for model order 2 and messages longer than 100 bytes,

- the **o2-32** kByte model gives mean compression rates between **4** and **3.7** bpb,
- the **o2-64** kByte model gives rates between **3.7** and **3.5** bpb,
- the **o2-128** kByte model performs between **3.7** and **3.4** bpb,
- and the **o2-256** kByte model achieves between **3.6** and **3.4** bpb.

For **order 3** (figure b)), the 32 kByte model is exhausted and does not work reasonably. For messages longer than 100 bytes,

- the **o3-64** kByte order 3 model gives mean compression rates between **3.9** and **3.5** bpb,
- the **o3-128** kByte model gives rates between **3.5** and **3.1** bpb,
- and the **o3-256** kByte model achieves the best rates between **3.2** and **3** bpb.

The usage of order 3 only makes sense for the larger models of 128 and 256 kBytes. The quartile bars were included into the plots of figure 2.36 to give some information on the skewness of the sample's distribution (each of the samples containing N=30 measurements). It is visible that the distribution of the measurements is generally right-skewed, that means that there are relatively few high compression values. That actually considers the median to be a better descriptor of the statistics. Regarding the median values of the compression, the performance of the described system is even better. However, to include the few high outlying values (in the fourth quartile) and because the mean is a conventional estimator in data compression evaluations, it is selected as well for the here given analysis.

The results show that especially for order 2, the novel model gives reasonable compression performance while preserving low computational requirements. The requirement of 3.3 bpb from section 2.5.8 is fulfilled with the order 3 256 kByte model (o3-256 kByte). Usage of order 3 models generally requires at least 128 kByte. It might be advisable to use shorter training files (e.g., the model *paper2*) that result into less context nodes, when only little memory is available to model a higher order.

2.6.4 Summary

In this section a scheme for low-complexity compression of short messages was introduced. The novel scheme simplifies the memory intensive components of the context modeling technique prediction by partial matching (PPM) and preloads statistical data to allow for compression from the first few bytes.

A main contribution is the novel statistical model, which allows the application on memory limited platforms. The model uses a trick to allow the tremendous memory savings: It does

not store the words of the codebook themselves, as it was done with previously introduced data models. Instead it just stores a one-byte code for any word, which can be of arbitrary length. Furthermore a compression technique for the models is detailed, which allows to keep several models for different types of data in the flash memory of the device.

Even if the measurements were only conducted for order 2 and 3, the scheme holds true for any model order and source code refinements are not necessary. The data model requires a special search function - which is a component of the context model, to explore the existing nodes on the probability line. Thereby the left, the right, and the total count are estimated, which are needed by the arithmetic coder for coding a symbol.

The results show that the novel scheme is able to compress short messages from 50 bytes. (The adaptive PPM technique and today's standard PC lossless compression techniques, e.g., the Lempel-Ziv LZ77 algorithm, do not compress short text.) The compression rate is scalable by the choice of the hash table size. For an order of 2, a data model of 32 kBytes already cuts the message in half. For a compression rate of 3.2 bpb an order 3 model with 256 kBytes was necessary. This actually fulfills the requirement of 3.38 bpb from section 2.5.8 for the scheme to be developed.

In the study in [14], compression ratios ranging from 2.95 to 3.1 for compression of the file *hrom210* using training data from *book1* is achieved. The better compression performance is obtained with the cost of higher conceptual complexity and memory requirements of 512 kBytes. In contrast to this approach, the here proposed scheme is designed as a scalable trade-off between complexity and compression performance.

Applications of the scheme may be in sensor networks for exchange of sensor and signaling data (e.g., routing), in space communications, or in the cellular world. In the next section the introduced scheme is applied to a cell-phone software that allows users to reduce their costs and battery consumption for short message services.

2.7 SMSzipper: Application in Cellular Network

The introduced low-complexity compression scheme was applied to a cell phone by porting the original C code to Java. The developed software allows the user to send compressed short messages. Thus it is possible to send two or three messages by one message. Moreover the software does not only allow the user to save costs but also to reduce the battery consumption, as it will be demonstrated by the measurements in this section. More details on the developed software are given in [49].

Figure 2.37 illustrates the required energy on the Nokia 6630 phone for encoding, sending, decoding, and receiving a message of $203, 291$, and 302 bytes, which is marked as *enc, snd, dec,* and *rcv*, respectively. The messages are listed in the appendix in figure 5.5 on page 148.

The measurements were conducted with the phone connected to the Agilent 34401A multimeter measuring the current from the Nokia battery with 3.7 V. Own scripts were employed to measure the energy consumption for each action (compression, sending, receiving, and decompression) separately. The display lightning was switched off throughout the measurements.

The total energy for sending a compressed message (marked as *cmpr*) is compared with the total energy for sending an uncompressed message (marked as *uncmpr*). The energy for sending or receiving a message is denoted as *transceiver* energy and the energy for encoding or decoding a message as *operational* energy. The two numbers above the bins indicate the battery savings for the sending phone (lower number) and the receiving phone (upper number) in %.

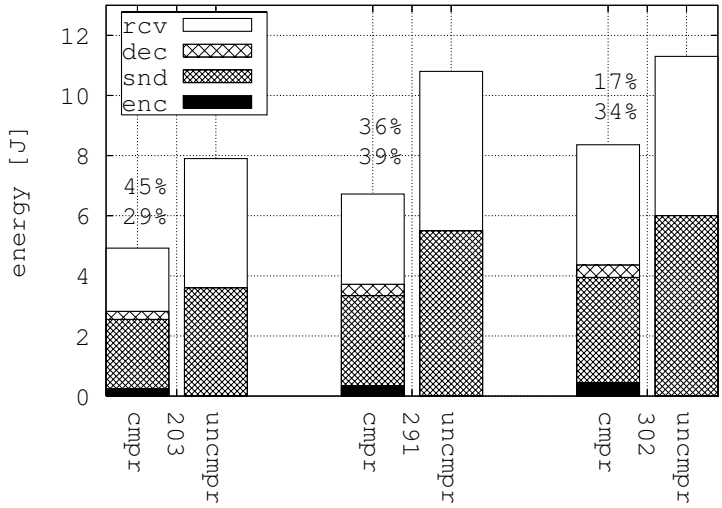


Figure 2.37: Operational energy (marked as *enc,dec* for encoding/decoding a short message) and transceiver energy (marked as *snd,rcv* for sending/receiving a message) needed to convey a message of different lengths with the own compression scheme (marked as *cmpr*) and without it (marked as *uncompr*). The three employed messages have 203, 291, and 302 characters and are given in the appendix. The two numbers on top of the bars give the battery savings in % for the sender (upper number) and the receiver battery. Compression not only pays off in user costs and effective medium usage, it also gives significant battery savings for the user's device.

As the operational energy is much smaller than the transceiver energy, the compression scheme on average pays off by 34% in battery savings for the receiver and by 33% for the sender.

Figure 2.38 shows the operational time on the Nokia 6630 for encoding and decoding a message with 203, 291, 302, and 454 characters using orders 2 and 3, see page 148 for the messages' content. When the messages have 203 bytes the encoding time is longer than the decoding time. When the messages are longer than 291 bytes the decoding time tends to be longer than the time for encoding. The reason why the encoding takes more time than the decoding for the smaller message is not clear, as the decoding procedure generally requires more computation. It might be a Java implementation issue concerning the load of the statistical model into the memory.

Order 3 takes more time than order 2, e.g. for the 454 byte message, it takes 1.13 seconds more than order 2 for decoding, and 0.83 seconds more than order 2 for encoding. The reason for this is that a higher order generally requires more computations to estimate a symbol probability, because there exist more contexts that have to be checked in the larger statistical model.

2.8 Conclusion Text Compression

In this section the work for the chapter on text compression is shortly summarized while the contributions are reviewed. For the development of a novel text compression scheme a set of steps were taken, each of them including own implementation work. The technique of PPM was selected as a starting point, as it appeared to deliver promising results. It is based on arithmetic coding, a standard technique for binary coding of symbols regarding their entropy. The first step was thus to review this technique and to provide an own implementation. For the standard

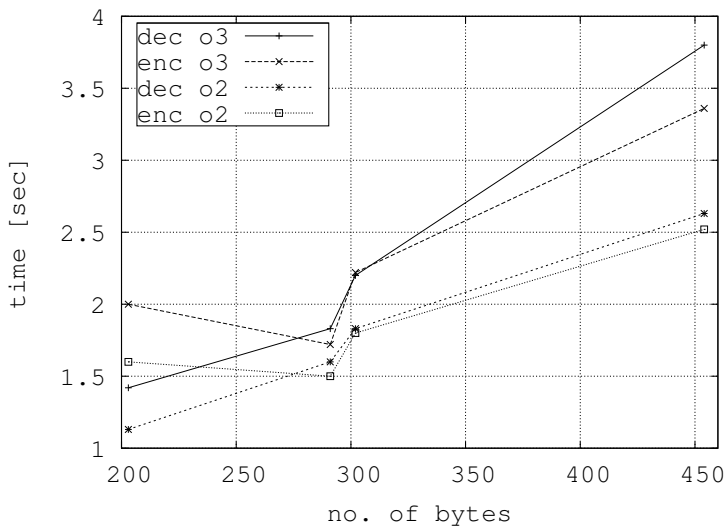


Figure 2.38: Time needed for encoding (marked as *enc*) and decoding (marked as *dec*) messages with the lengths of 203, 291, 302, and 454 characters for order 2 (marked as *o2*) and order 3 (marked as *o3*) on the Nokia 6630. A typical message of 302 bytes will be cut in half in approximatively 1.8 seconds. Thus only one message instead of two messages will be sent, which results into cost savings. The user receiving the message will probably not perceive the decoding time of 1.83 seconds, as the software starts the decompression immediately when the message is received.

PPM technique, an own concept of a data structure was designed and then implemented in C++. When the PPM system was verified to work correctly, the software was extended to allow for an analysis of the statistics. The results of the analysis helped to design the novel low-complexity scheme, which was implemented in C. The final compression scheme fulfilled the list of requirements, including low memory, compression of short messages, and low complexity. Short messages were already cut in half for a model of 32 kByte. (The most recent related work in [14] required 0.5 MByte at least.)

Future work can concern the construction of a codebook that is based on a database of short messages. This might furthermore reduce the memory requirements and improve the compression. A reduction in computing time can be achieved by the use of the *range coder* [28, 50] instead of the arithmetic coder. This may be especially interesting for the application in sensor networks, where computational resources are even more limited than in cellular devices.

To allow for the verification of novel compression techniques to be applied in sensor networks, an own sensor platform is designed in the next chapter. Future work that concerns the improvement of the text compression technique given in this chapter is currently carried out by student work [50].

The field of data compression can be classified in 1) lossless and 2) lossy techniques. This chapter concerned a lossless technique. To broaden the relevance of this thesis, in the next chapter a novel technique for compression of pictures is introduced, which may refer to the class of lossy compression.

Chapter 3

Picture Compression in Sensor Network

3.1 Chapter Overview

In this section the chapter on picture compression is shortly outlined. The chapter concerns the introduction of a novel system for wavelet picture compression with extremely low memory requirements. These requirements are typically defined by small wireless sensors, which may build a sensor network.

Wavelet picture compression is twofold - the first step is the wavelet transform of the picture samples, and the second step concerns the coding of the transformed samples. The first step does not result in a compression at all, however, the data space is prepared to allow for a data reduction technique, which is applied in the second step. This technique is generally an algorithm that exploits the appearance of sets of very small coefficients, which occur in known patterns.

The chapter thus gives two contributions - a novel computation scheme for the wavelet transform, which is called *fractional wavelet filter*, and a novel recursive algorithm for coding of wavelet coefficients - that is, the *wavelet image two-line* (Wi2l) coder. In section 3.2 related work on the fractional wavelet filter and the Wi2l coder is reviewed.

The novel algorithms require a real system to be verified. Section 3.3 introduces the for this purpose developed own sensor network platform in soft- and hardware. (A related version of this section appeared in [51].) The platform is designed to allow the application of novel signal processing techniques on the sensors themselves.

Section 3.4 reviews a filter method for the one-dimensional wavelet transform. The scheme can be applied in two dimensions thus realizing the wavelet picture transform. For the sensor node hardware, however, this method is not yet applicable due to the limited memory and the very slow support of floating-point numbers, which results from the 16 bit processor architecture.

In section 3.5 the novel fractional filter as a solution to overcome these limitations is detailed. (The filter was primarily introduced in [52].) The filter allows the computation of the picture wavelet transform on a camera sensor node with less than 1.5 kByte of RAM memory, while it uses fixed-point arithmetic. The filter is evaluated on the own sensor platform equipped with a camera and a multimedia card (MMC) for storage of the picture data.

Section 3.6 gives an introduction to current picture wavelet coding techniques. Two techniques are most established in this field - the *embedded zerotree wavelet* (EZW) and the *set*

partitioning in hierarchical trees (Spiht) algorithm, which is an improved version of EZW. Finally the *backward coding wavelet tree* (Bcwt) is reviewed, a recent algorithm that tremendously reduces the memory requirements of Spiht by reversing the coding order of the coefficients. As this recent technique appears to be promising, it is selected as a starting point for the own investigations. This implies - similarly as with the methodology in the first chapter for text compression, an own implementation of Bcwt, to be shortly touched in this section.

Section 3.7 details the novel recursive Wi2l coder for the picture compression in sensor networks. (The coder was primarily introduced in [53].) Similarly than Bcwt, the own algorithm encodes the picture backwards, however, it breaks the trees of wavelet coefficients to be processed in small blocks. As a result only two lines of a wavelet subband need to be kept in memory, which reduces the memory requirements by more than five times. Similarly than with the fractional filter, less than 1.5 kByte of memory are required. The novel coder is finally evaluated on the own platform using the MMC card as an input and output medium.

In the last section the picture compression chapter is summarized.

3.2 Related Work

3.2.1 Low-Complexity Wavelet Transform

One major difficulty in applying the discrete two-dimensional wavelet transform to a platform with scarce resources is the need for large memory. Implementations on a personal computer (PC) generally keep the whole source and/or destination picture in memory, where horizontal and vertical filters are applied separately. As this is generally not possible on a resource-limited platform, the recent literature addresses the memory-efficient implementation of the wavelet transform.

A very large part of the literature concerns the implementation of the wavelet transform on a field programmable gate array (FPGA), see for instance [54, 55, 56, 57]. The FPGA-platforms are generally designed for one specific purpose and are not an appropriate candidate for a sensor node that has to perform many tasks concerning communication and analysis of the surrounding area, see [4, 51] for details.

This work is different from the literature on FPGAs in that it considers the 9/7 picture wavelet transform for a microcontroller with very little RAM. Such a solution can easily be integrated in current sensor network platforms and offers much flexibility. Costs and programming efforts are limited as the extension only concerns a standard multimedia card, a camera module, and a software module for the transform.

The most related work to this thesis is given in [58], where a line-based version of the wavelet transform is given. The authors describe a system of buffers where only a small subset of the coefficients has to be stored thus tremendously reducing the memory requirements compared to the traditional approach. A very efficient implementation of the line-based transform using the lifting scheme and improved communication between the buffers is detailed in [59], where the authors use a PC-based C++ implementation for demonstration.

In the context of our requirements, however, it is not applicable to a sensor node with extremely little RAM as it uses in ideal case 26 kByte RAM for a six-level transform of an 512x512 picture, whereas the approach given in this thesis would only require roughly 5 kByte using the floating-point scheme. Any paper where the 9/7 picture wavelet transform is implemented on a low-cost 16 bit microcontroller was not found throughout the literature research.

3.2.2 Low-Complexity Coding of Wavelet Coefficients

There exists a large amount of literature in the field of low-memory wavelet-based image coding. In the following work is reviewed that intends to reduce the memory requirements of Spiht, which is a standard technique for image coding that gives very competitive compression while it is conceptually simple. The work is categorized into two classes, the first one reducing the requirements for storage of intermediate data, and the second one concerning the maintenance of the picture data itself. Examples of the first class are given in [60, 61, 62, 63, 64, 65, 66], where the list of insignificant pixels (LIP), the list of significant pixels (LSP), and the list of insignificant sets (LIS) are reduced or excluded. As these lists are essential in Spiht and can require several MBytes of RAM, the reported memory savings are given in respect of the lists.

In the second class of low-memory coders the memory requirements of the transformed picture itself are addressed. Whereas the first class assumes free access on each pixel, this class only considers a subset of all pixels to be accessible by the coding algorithm. The memory needed for this subset is added to the RAM requirements. This class is discussed in the following to be more relevant for this thesis, as only a small buffer of coefficients is available for direct coding activities.

The coders in [67, 68] both are designed for FPGA platforms and require 15 kByte of RAM for a picture dimension of 128x128, and 0.5 MByte of RAM for a picture dimension of 512x512. The work in [69] reports that there are at minimum eight memory lines needed by proposing a novel tree structure, which would require 256x8x2 bytes RAM for a picture dimension of 256x256 using 16 bits per coefficient. Results are given using a Matlab simulation.

While these works require specific hardware or were not yet verified using low-level implementations, the memory requirements are still too high. The backward coding wavelet tree (Bcwt) algorithm in [70, 6] is here considered to give the most potential for a low-complexity wavelet coder and thus its idea of backwards coding is resumed in this thesis. In Bcwt the encoder starts at the lowest level and the coded coefficients can here be discarded as a map keeps track of the most significant levels of a tree. It is actually a backward version of Spiht with same compression rates with the cost that the progressive feature is distorted.

If w denotes the picture width in pixels, Bcwt requires $42 \cdot w$ coefficients (16 bit) and $3 \cdot w$ quantization levels (8 bit) for the map, resulting in more than 20 kBytes of RAM for a 256x256 picture. This is due to the fact that coding one Bcwt unit requires data from several lines from different wavelet levels. A Bcwt unit codes a block of 16 coefficients while data of the parent coefficients is required. A unit generally also needs tree level data, which is stored in a list that concerns a complete subband.

This thesis introduces the novel backward Wi2l coder that in contrast to Bcwt operates on two lines of a subband and only stores intermediate results concerning these lines in a small maximum quantization level (MQL) buffer. Thus the memory requirements are approximatively 1.5 kByte for the reference picture dimension of 256x256 pixels. To our best knowledge such a low-requirement system has not yet been proposed in the literature.

3.3 Building an own Sensor Network Platform for Signal Processing

3.3.1 Introduction

Designing a wireless sensor ought to be simple: Take a microcontroller, a sensing circuitry, a transceiver chip and put them together with the appropriate software - these may be the basic steps. If network communication protocols are included in the software architecture, a low-cost and very flexible sensor network is established, which can lead to many creative applications, including personal assistance, medical care, or surveillance. This may be one of the reasons why sensor networks have aroused so much interest in the communication society, and more recently also in the data and signal processing community.

There exist ready-to-use sensor network solutions like the family of *Mica modes* from Berkeley [71], for which the operating system *TinyOS* [72] is available. Such solutions allow for immediate start of sophisticated research investigations beyond of basic hardware and software problems that have already been solved by others. However, due to the plan of this thesis - the design of a novel picture compression algorithm, the available solutions did not appear to be useful, because the specific requirements of an integrated camera and a flash memory to store the pictures were not fulfilled. For these reasons an own architecture with the following specifications was to be built:

- The sensor board needs a camera interface or even better an integrated camera to take the pictures to be compressed.
- A flash memory card, preferably a multimedia card (MMC) for storage of the picture data shall be part of the board.
- The sensor board shall be well-priced. This can be realized by designing the sensor as a construction kit, where the pieces can be put easily together.

Additional specifications for the platform which are typical for wireless sensor nodes are given as follows:

- The sensor shall have low energy requirements. This can be realized by a low-cost 16 bit microcontroller.
- A transceiver circuitry is necessary to allow for a communication between the sensors in the industrial, scientific, and medical (ISM) band.
- Serial, USB, or bluetooth interfaces are needed for a connection to a PC.

Another aspect that demands an own sensor network are the joint chances of this in education. Class work in wireless communication is often performed by the study of the underlying principles - i.e., network architectures and protocols. A large set of low cost sensors can easily be distributed to the students for conducting own experiments and even programming some basic algorithms, thus getting a more thorough understanding in communications. The sensor introduced in this section was welcomed by many students as a nice playground to develop novel applications. Furthermore, as small and mobile devices have already become part of our daily life, experience in embedded programming is a worthwhile educational investigation.

Sensor networks impose a broad field of disciplines including sensor network architecture, energy saving communication protocols, and security. A very thorough description of these is given in [4]. This section focuses on the signal processing aspects of a small sensor network and only scarcely considers the communication aspects - which include medium access protocols (MAC), time synchronization, localization, and routing, to name a few. Furthermore, in distributed signal processing, a smaller number of sensors can already lead to interesting applications. In such a scenario the communication aspects are less complex than in a very dense network.

The outline of this section is given as follows. Subsection 3.3.2 describes the main components of the sensor board. These are the microcontroller and the transceiver chip. The appropriate software for the system is given in subsection 3.3.3. The section only details the employed medium access protocol and a general data structure that may simplify the sensor's signal processing computations. In subsection 3.3.4 a conclusion is given.

3.3.2 Hardware

In this section the processor and the transceiver chip of the designed platform is described. A schematic of the hardware components and possible extensions is depicted in figure 3.1. A

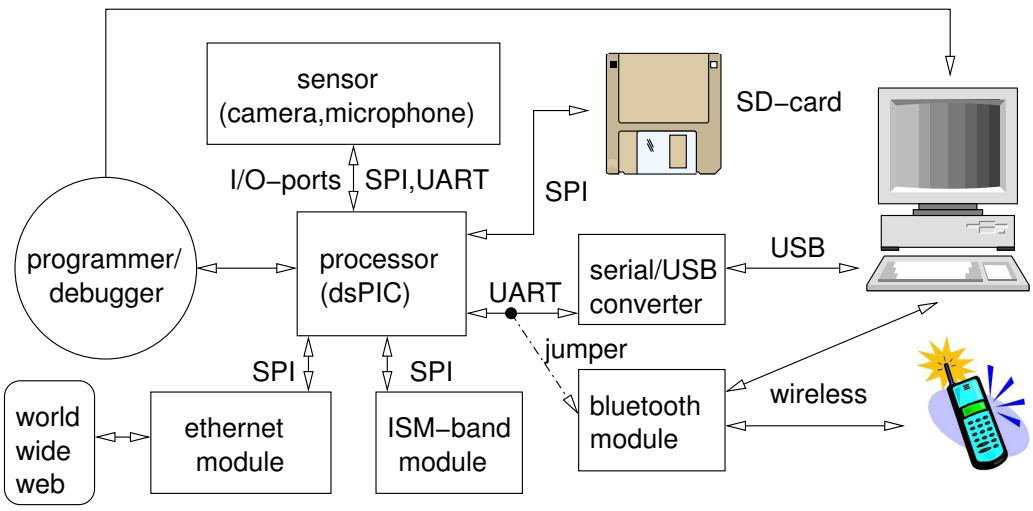


Figure 3.1: Schematic of the hardware components and possible extensions of the own platform. The main parts are the processor, the serial/USB- converter, and the ISM-band module. The processor was programmed in C using a commercial in-circuit debugger. Later a camera and a multimedia card were included to allow for the verification of a novel picture compression algorithm.

prototype of the first sensor board is illustrated in figure 3.2. Possible extension modules can concern audio, picture, and global positioning system (GPS) data, memory card, and Ethernet, to name a few examples. A memory card extension may allow for building a data logging system or be useful when the collected data can not be processed in time. An Ethernet module may be helpful for remote access allowing for distant surveillance (software for an Internet protocol stack is freely available). In a later section of this thesis the extensions camera and memory card are employed to verify an own picture compression algorithm.

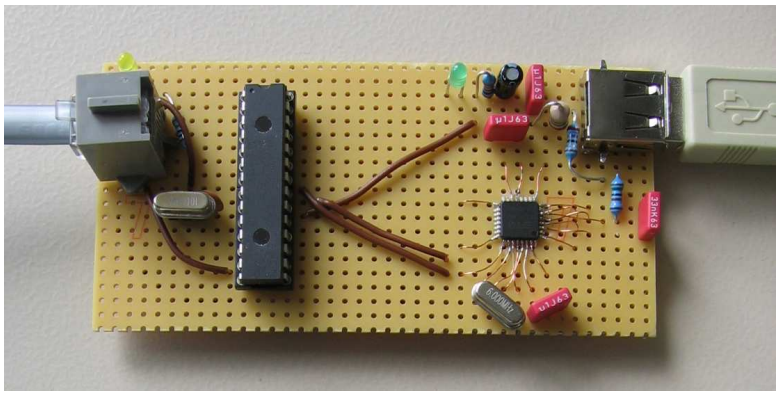


Figure 3.2: Primary prototype of the own sensor board. The board consists of the signal controller *dsPIC* from Microchip and an USB- RS232 converter. The board contains a port for connecting a programmer and an USB cable. A simple own terminal program allowed to send commands to the processor to test the computation of a matrix product.

3.3.2.1 Processor

In this section the choice of the processing unit for the sensor board is motivated. There exist two main classes of architectures, the *microcontrollers* and the *digital signal processors* (DSP). The microcontrollers fulfill most of the requirements despite the extensive math abilities. The DSPs are explicitly optimized for high-level math - more specifically, vector and matrix operations - excluding most of the other required features. These features especially concern the *embedded design*, which refers to a highly integrated processor that does not need external components to start the operation. A microcontroller is therefore the typical choice for a wireless sensor [4].

In this thesis, the *dsPIC* controller from Microchip is selected as a processor. It is a 16 bit processor with a math coprocessor that combines the individual benefits of microcontrollers and DSPs. The reasons for this choice are given as follows:

- The *dsPIC* is a low-cost 16 bit processor, which is typical for a wireless sensor node [4].
- It offers hardware support to multiply two fixed-point 16 bit numbers in that intermediate 32 bit results can be stored, which is needed for the fixed-point wavelet transform of a picture with 16 bit samples.
- It uses a refined version of the GNU C-Compiler. Prototype algorithms developed on a PC can easily be ported to the sensor hardware.

3.3.2.2 Transceiver

For transmission of data packets, a wireless sensor needs a radio frequency transceiver circuitry. Due to the here required embedded design it makes sense to select a transceiver chip that almost needs an antenna but not many more components to start operation. These transceivers are available for different commercial standards, which exist for wireless local area networks (WLAN) like IEEE 802.11 and wireless personal area networks (WPAN) like IEEE 802.15.4. Two examples for WPANs are Bluetooth specified in IEEE 802.15.1 and Zigbee as an extension of IEEE 802.15.4, both operating in the industrial, scientific, and medical band (ISM). The standards include physical layer and medium access protocol (MAC) specifications.

One reason why IEEE 802.11 and bluetooth are considered as less suitable for wireless sensor networks are the higher energy consumption, which is caused by the high bitrates of IEEE 802.11 and in case of Bluetooth by the master node activity to poll the slaves [4]. Even if there exist more suitable standards, the here detailed flexibility requirement requests to use none of them:

- Using a standard does not necessarily allow for the design of own and completely novel protocols.
- It might give a too complex environment for testing novel signal processing algorithms and for learning the basics in wireless communications in the classroom.
- The costs for a transceiver with a commercial standard are higher.

In this thesis, the single chip radio transceiver nRF905 from Nordic is employed. Its features are given as follows:

- 433/868/915 MHz ISM band
- channel resolution of 100/200 kHz
- *Gaussian frequency shift keying* (GFSK) modulation
- maximum data rate of 50 kiloBits (kBits) per second
- internal Manchester coding
- serial peripheral interface (SPI)
- hardware cyclic redundancy check (CRC)
- Carrier detect function
- Manchester coding of the packets
- packet payload from 1-32 bytes and address size from 1-4 bytes

Similarly as the selected processor unit, the nRF905 is an exemplary choice. As packet size and transmit time can be set by the user it is possible to design own MAC protocols with the restrictions given by the GFSK modulation, which for instance does not allow for *code division multiple access* (CDMA) schemes. However, a schedule-based protocol like *time division multiple access* (TDMA) or a contention-based protocol like carrier sense multiple access (CSMA) are possible. A simple variant of CSMA to be employed in this section is presented in section 3.3.3. The exploration of more complex MAC protocols is left to the interested reader. TDMA requires precise time synchronization and imposes more signaling overhead. A *frequency division multiple access* (FDMA) protocol might be employed as well as the transceiver supports multiple channels. *Space division multiple access* (SDMA) is not regarded as a candidate for sensor networks in [4] because of the required sophisticated arrays of antennas and signal processing techniques. However, for the specific sensor network described in this section it may be an interesting field of research.

3.3.3 Software Design

The software modules of the designed sensor are illustrated in figure 3.3. Of major importance are the MAC module that is implemented in *trans.c* and the *stack* module, both to be described in this section.

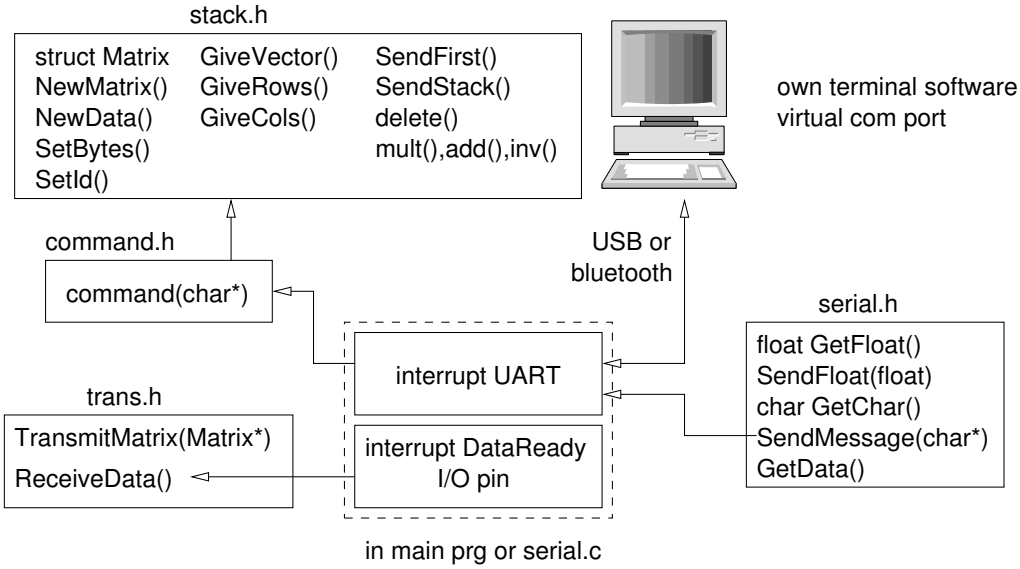


Figure 3.3: Software design of the own sensor. It consists of a command wrapper for user interaction, a serial interface module with help functions to allow data exchange between the user and the sensor, a module for wireless transmission of data via the ISM-band, and a stack module for data processing.

3.3.3.1 Medium Access Protocol

The medium access control (MAC) module is one of the basic components to establish communication. In case all sensors share the same channel, a protocol like ALOHA or CSMA may be considered. (For streaming applications a TDMA protocol may be more suitable.) In pure ALOHA each node can transmit whenever it has data. Thus, a channel utilization up to 18 percent can be achieved [73]. Slotted ALOHA can give a throughput of 37 percent [73], however, synchronization between the nodes is necessary. Carrier sense protocols where the node first listens to the channel can give better performance. In *1-persistent* CSMA a packet is sent immediately when an idle channel is detected. In *non persistent* CSMA the nodes wait for a random time when the channel is in use and then sense the channel again. This prevents a collision from two nodes that wait until the end of a transmission of a third node before they both immediately transmit.

A very simple variant of a non persistent CSMA protocol is employed here, see figure 3.4. It is similar to the protocol presented in [74] expect that the ready-to-send (RTS)/ clear-to-send (CTS) handshake is only done once for the complete array of packets. Thus signaling overhead is reduced as the packets have a relatively small payload. The basis for the own implementation was given by the supervised work in [75], where a basic communication between two sensors was addressed.

The packet layout is given in figure 3.5 for a data packet and in figure 3.6 for the signaling packets. There are fixed fields for the preamble, the receiver's address (ADDR) with four bytes

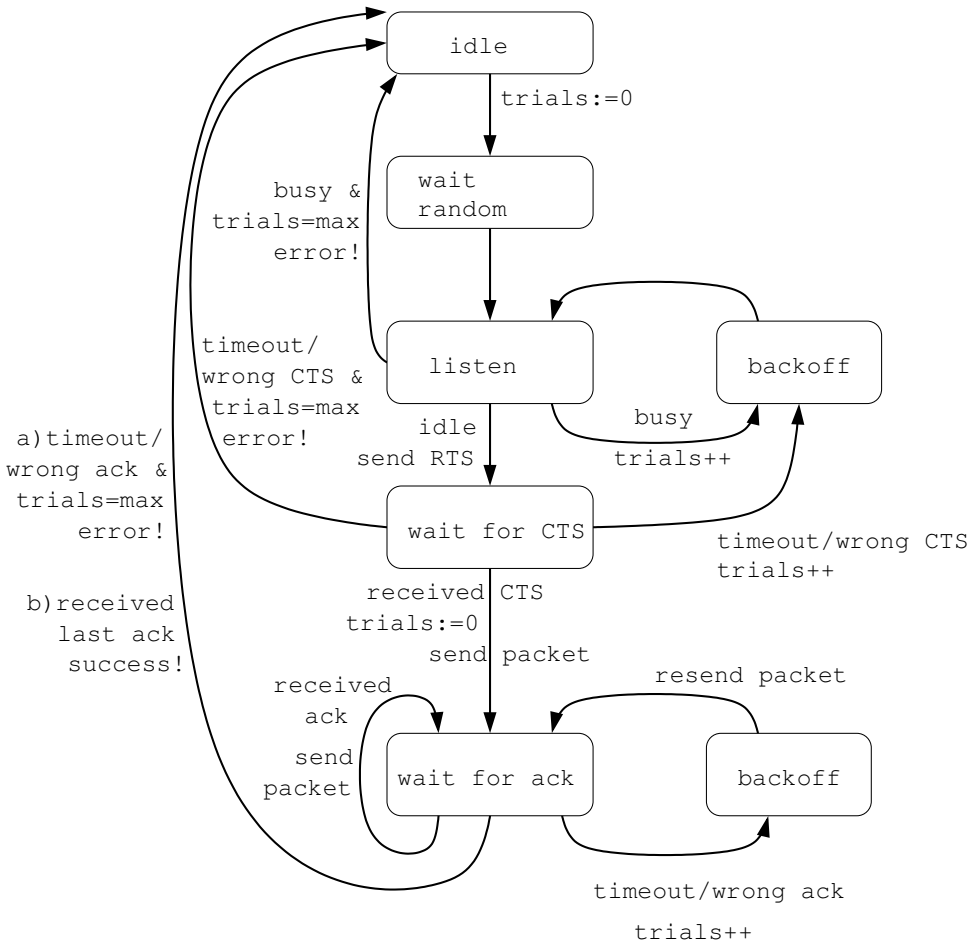


Figure 3.4: CSMA protocol as it is employed on the own sensor. It allows for a reliable transmission where each packet gets an acknowledge. RTS/CTS - handshake is only done once when the connection is established, and not for each packet.

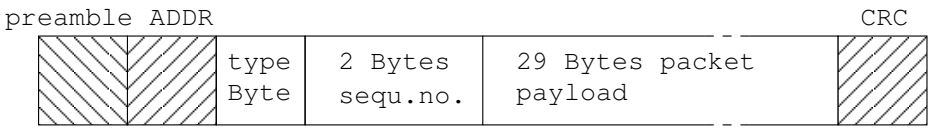


Figure 3.5: Fields of a data packet. The hatched fields are allocated by the transceiver to perform an address- and a CRC check. A type byte is used to specify the data format and a sequence number to allow for file transfer. Each packet can contain 29 bytes of payload data.

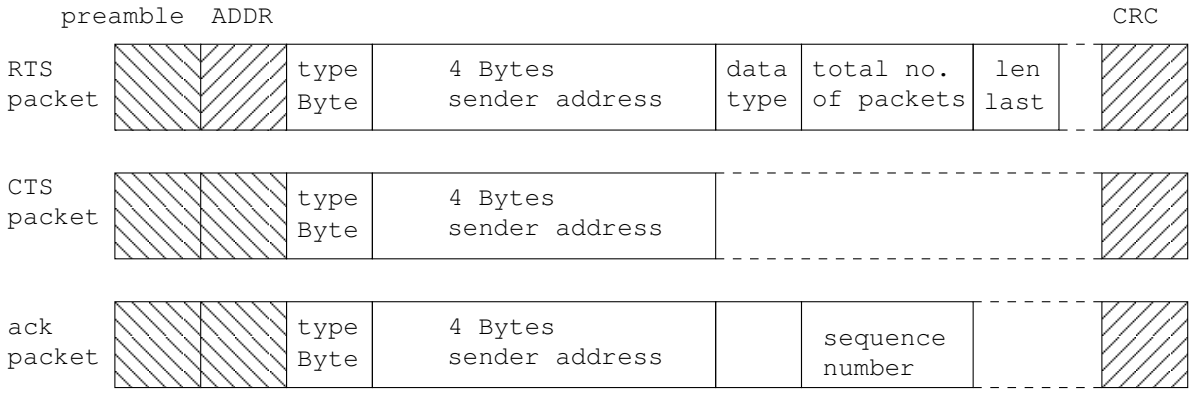


Figure 3.6: Fields of the three types of signaling packets employed by the medium access protocol. The ready-to-send (RTS) and clear-to-send (CTS) packets establish a connection. For each data packet an acknowledge (ACK) packet is sent.

at most, and the CRC checksum. The user payload can be 32 bytes at most. Each packet has a *type byte* to specify the type of packet, which can be data, RTS, CTS, or acknowledge (Ack). The RTS packet has a byte to specify the data type (such as float, integer, or character), an integer field for the total number of packets, and one byte to give the length in bytes for the last packet. This information is stored by the receiver when a connection is established and is not contained in the data packets themselves.

The achieved data rates using the own hardware with the described protocol are given for two distances in figure 3.7. In ideal case, a transmission rate of 12.5 kBits per second is achieved.

3.3.3.2 The Stack: An Embedded Calculator

This section shortly outlines the so-called *stack*, a system needed to verify data compression techniques on the sensor hardware. As illustrated in figure 3.3, the stack can be regarded as an operating system that realizes the sensor data management and transfer to be controlled by the user. It contains an embedded calculator related to the reverse polish notation system (RPN). In RPN the operands precede the operator. For example, (2+5)/3 is computed by entering 3, 2, and 5 into the stack and then calling an *add* and a *divide* operation. In this work the stack is twofold: It is first the data structure of a linked list, and second, it contains a set of functions that can be performed on the stack elements - e.g., filter operations or a wavelet transform, which are typical signal processing operations. A stack element can be a matrix with picture data. Data can be transferred from a PC to the sensor using a terminal program. Stack elements can also be stored to flash memory or vice versa.

3.3.4 Summary and Continued Work

In this section the sensor platform that is employed in this thesis to verify the picture compression algorithms is introduced. The platform is designed to especially fulfill the needs of signal processing techniques. It is realized by a mixed controller that is easy to handle yet including hardware support for filter operations. A standard transceiver chip with a CSMA software module lets the sensors exchange data with rates of 12.5 kBits per second. Although

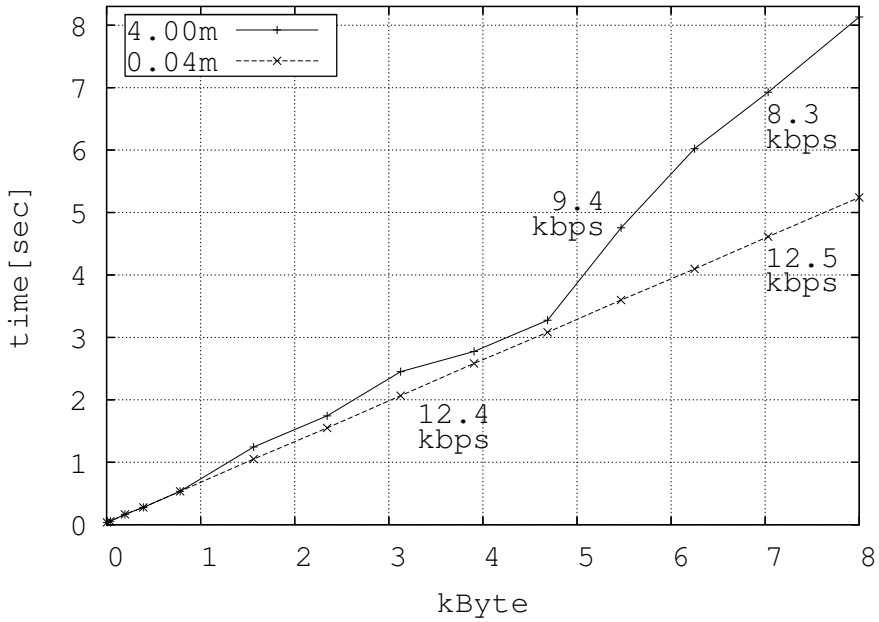


Figure 3.7: Time that is needed to transfer up to 8 kBytes achieved by the employed transceiver chip using a CSMA protocol. Measurements were conducted in a small office room for the distances four centimeters and four meters. For the larger distance, the sensors had no line of sight and were placed at different heights from ground.

this seems to be a fairly low rate, it may be sufficient for transmitting the result of data processing algorithms. The simple communication protocol fulfills the low energy constraints that are typical for sensor networks.

Due to space constraints, the next step in this work – that is, distributively computing a matrix product in the given wireless sensor network – is not detailed here, and the interested reader is referred to the book chapter in [51]. The findings of these considerations helped to take the next steps towards the wavelet transform.

The following sections concern the field of wavelet image compression on sensor node platforms. For the verification of wavelet algorithms, the system described in this section was extended and revised. Software to control a small camera module was included in [76]. An appropriate library for a multimedia card (MMC) was introduced in [5]. The platform was revised to be easier to assemble. For this purpose another version of the processor was selected, which only has 2 kBytes of RAM memory (in the selected processor family more RAM memory was connected with not suitable processor packages that did not allow to easily build the sensor). A picture of the revised hardware is given in the appendix in figure 5.13 on page 159. The next section gives an introduction on computing the wavelet transform for pictures.

3.4 Computing the Picture Wavelet Transform

3.4.1 Introduction

In this section the computation of the picture wavelet transform is described. The section does not describe the basics and foundations of the wavelet transform, as there exists a large amount

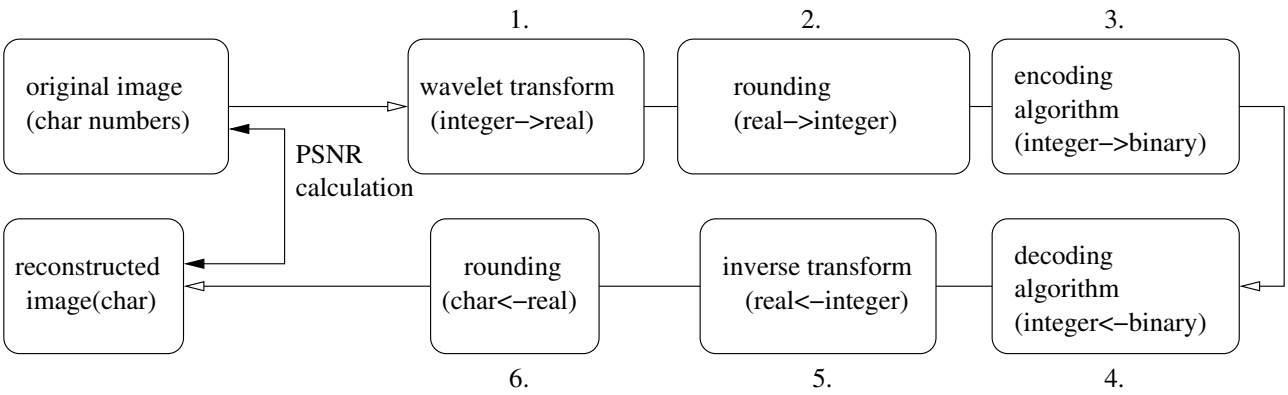


Figure 3.8: Rough outline of wavelet-based picture compression. In this thesis a wavelet transform with real numbers is applied, as it gives better compression. The rounding in step 2) does not result into a loss of picture quality. The data compression and the loss of quality results from step 3). In a later section the transform is computed with fixed-point arithmetic.

of literature on wavelet theory, see for instance [77, 78, 79]. The computational schemes of the wavelet transform, however, are rarely discussed, as many books assume the usage of wavelet toolboxes. This is probably due to a general methodology of the signal processing researcher, who uses a PC with a high-level language to discover the features of the transform and to apply it to solve a given problem, such as the detection of specific patterns in a data base [80]. In fact, this may be a reasonable approach as theoretical work on wavelets is available to a tremendous extend, while applications are only scarcely sown. However, in the context of this thesis the computational schemes to be described in this section are of importance, as they have to be refined to be applied to a very limited platform. Some more details including C code are given in [81].

The computation of the wavelet transform is part of the typical methodology for wavelet-based picture compression, as illustrated in figure 3.8. The original picture is given in the signed character data format (which refers to the CHAR data type in the C-language). Then the picture wavelet transform is applied. The rounded coefficients are encoded to a binary stream in step 3). This is actually the step where the compression is performed. The wavelet transform only prepares the data in such a way that an appropriate algorithm can exploit the correlations. Usually this step results into a loss of quality. In step 4) the picture is decoded. In step 5) the inverse wavelet transform is performed. The result is rounded and forms the reconstructed image. The peak signal to noise ratio (PSNR) can then be calculated to estimate the loss of quality as follows. Let j, k denote the single pixels of the original image $g[j, k]$ and the distorted image $f[j, k]$ with $N \cdot N$ pixels. The PSNR is given for eight bit per pixel (bpp) images in decibels [dB] as

$$PSNR = 10 \cdot log_{10} \left(\frac{255^2}{MSE} \right), \tag{3.1}$$

where the *mean square error* (MSE) is calculated as

$$MSE = \frac{1}{N^2} \sum_{\forall j,k} (f[j, k] - g[j, k])^2. \tag{3.2}$$

The logarithmic notation refers more meaningfully to the human perception. A PSNR of 40 dB or larger is nearly not distinguishable for human observers [82].

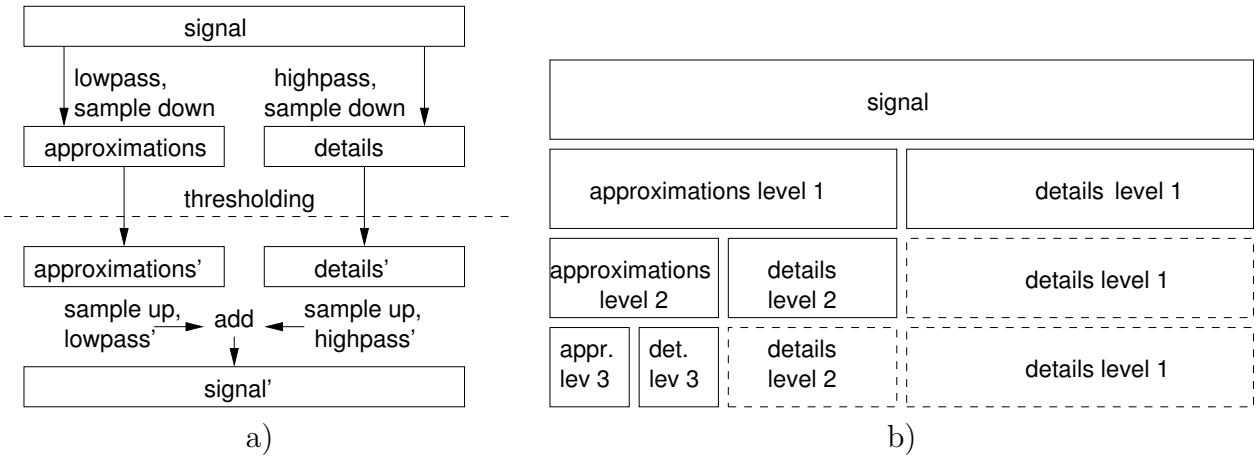


Figure 3.9: Figure a) shows a one-level wavelet transform, including the reconstruction. The thresholding operation deletes very small coefficients and is a general data compression technique. These small coefficients are expected to not contain meaningful information. Figure b) illustrates a 3-level wavelet transform. For each next level the approximations of the previous level build the input. The details are not further processed and just copied to the next levels.

This section is organized as follows. In subsection 3.4.2 the operation of convolution is applied to compute the one-dimensional wavelet transform. The coefficients of the *Daubechies* 9/7 wavelet are listed to be employed from then. To achieve a multi-level transform the *pyramidal algorithm* is applied. (The *lifting scheme* [83] for in-place calculation of the wavelet transform is not discussed here as it is not applied by the novel filter to be introduced in this thesis.) Subsection 3.4.3 finally details how the one-dimensional transform is applied to perform the two-dimensional picture wavelet transform.

3.4.2 Wavelet Transform (WT) for One Dimension

3.4.2.1 One-dim-WT using Convolution

Haar-Wavelet To simplify the readability, the *fast dyadic wavelet transform* is in this thesis simply denoted as wavelet transform. Other transforms are not applied in this work. A wavelet transform can be computed by filter operations, for which in this thesis symmetric convolution is used to extend the picture lines at their boundaries. For instance, a filter operation with the signal **s** and the filter **h** can be computed by sliding the filter over the signal:

$$\begin{array}{|cccccccc|} h_2 & h_1 & h_0 & \rightarrow & & & & \\ s_2 & s_1 & s_0 & s_1 & s_2 & s_3 & s_2 & s_1 \end{array} \tag{3.3}$$

A one-dimensional wavelet transform is typically performed by using two different filters - that are a lowpass and a highpass filter, as illustrated in figure 3.9 a). The lowpass and highpass refer to the scaling filter and the wavelet filter, respectively. The filtered signals are sampled down - that is, leaving out each second value and keeping the even indexed elements. Note that this is a difference to a usual filter operation. The so obtained values are the wavelet coefficients and are called *approximations* and *details*. The number of approximation and detail coefficients equals the signal dimension. To reconstruct the original signal the coefficients have to be sampled up - that is, inserting zeros between each second value. The vector

$$s = [1, 2, 3, 4]$$

would thus result into

$$s_{2up} = [1, 0, 2, 0, 3, 0, 4, 0].$$

Then the sampled up versions of the approximations and details are filtered by the flipped lowpass and highpass filters. Last, both filtered arrays of values are summed up. Such a wavelet transform can be performed multiple times to achieve a multi-level transform, as illustrated in figure 3.9 b). The scheme is called *pyramidal algorithm.* In this scheme, only the approximations are continued to be processed.

Now a *Haar-Wavelet* filter is applied on the example signal $s = [4, 9, 7, 3, 2, 0, 6, 5]$. The Haar-filter coefficients for the lowpass are given as [0.5 0.5], and for the highpass as [0.5 -0.5]. The detail coefficients for the first level can be computed by sliding the highpass filter over the signal (without using signal extension), resulting into

$$det = [-2, -2.5, 1, 2, 0.5, 1, -3, 0.5, 2.5].$$

With the sample down operation the details are given as

$$det_2 = [-2.5, 2, 1, 0.5].$$

All the other coefficients can be computed similarly. Actually, it makes more sense not to compute all the convolution values if each second value is discarded anyway. Thus, the filter coefficients should be slided by two sample steps instead of one step, as illustrated for the first two values:

$$\begin{array}{|cccccccc|} 0.5 & -0.5 & \rightarrow & & & & & \\ 4 & 9 & 7 & 3 & 2 & 0 & 6 & 5 \end{array} \qquad \begin{array}{|cccccccc|} & & 0.5 & -0.5 & \rightarrow & & & \\ 4 & 9 & 7 & 3 & 2 & 0 & 6 & 5 \end{array} \tag{3.4}$$

These convolution values are computed as

$$0.5 \cdot 4 - 0.5 \cdot 9 = -2.5 \quad \text{and} \quad 0.5 \cdot 7 - 0.5 \cdot 3 = 2.$$

The three-level wavelet transform for this signal is given as follows:

4	9	7	3	2	0	6	5	signal
6.5	5	1	5.5	-2.5	2	1	0.5	level 1
5.75	3.25	0.75	-2.25					level 2
4.5	1.25							level 3
4.5	1.25	0.75	-2.25	-2.5	2	1	0.5	result

Note that the result vector contains all the detail coefficients of the previous levels, which are not further processed. The result vector can be computed with linear algebra using a matrix by vector product, where the matrix contains the filter coefficients for the low- and the highpass [84]. The reconstruction can be computed using the matrix inverse. Note that generally the normalized filter coefficients are used instead of the previous ones that were employed due to ease of demonstration. The normalized coefficients allow for perfect reconstruction and are given as

$$lp = [\frac{1}{\sqrt{2}} \quad \frac{1}{\sqrt{2}}]$$

for the lowpass filter and as

$$hp = [\frac{1}{\sqrt{2}} \quad \frac{-1}{\sqrt{2}}]$$

for the highpass filter.

In the next subsection a more advanced wavelet will be discussed. This wavelet has more coefficients while the principles of the Haar-Wavelet similarly hold true.

j	analysis lowpass	analysis highpass	synthesis lowpass	synthesis highpass
-4	0.037828			0.037828
-3	-0.023849	0.064539	-0.064539	0.023849
-2	-0.110624	-0.040689	-0.040689	-0.110624
-1	0.377403	-0.418092	0.418092	-0.377403
0	0.852699	0.788486	0.788486	0.852699
1	0.377403	-0.418092	0.418092	-0.377403
2	-0.110624	-0.040689	-0.040689	-0.110624
3	-0.023849	0.064539	-0.064539	0.023849
4	0.037828			0.037828

Table 3.1: Filter coefficients of the Daubechies 9/7 wavelet. These coefficients are employed in this thesis as they give state-of-the-art image compression. They also are part of the JPEG2000 image compression standard. Computationally the 9/7 wavelet is not an ideal choice, as it requires real computations and a relatively large number of coefficients. A computational scheme that applies a 9/7 picture transform on a limited platform with less than 2 kByte of RAM would be a novelty.

Daubechies 9/7 Wavelet The *biorthogonal Daubechies* 9/7 wavelet (also called FBI- fingerprint wavelet or Cohen-Daubechies-Feveau wavelet) is used in many wavelet compression algorithms, including the *embedded zerotree wavelet* (EZW), the *set partitioning in hierarchical trees* (SPIHT) algorithm [79], and the JPEG2000 compression standard for lossy compression. The JPEG2000 book in [85] motivates its preferred usage over other wavelets for lossy picture compression by the results of quality evaluations, where it gives the best performance:

> *...this is referred as the "CDF 9/7" transform, since the low- and high-pass analysis filters have 9 and 7 taps respectively. This transform has been found to yield optimal or near optimal performance in image compression applications and has enjoyed widespread popularity in the image compression community. It is one of the two transforms which must be implemented by every compliant JPEG2000 decompressor, the other being that of Example 6.3.*

(In the book, the given Example 6.3 refers to the Spline 5/3 transform, which is employed in JPEG2000 for the lossless case.) For the same reasons, the 9/7 wavelet is also used in this thesis.

The 9/7 wavelet is given by its filter coefficients in table 3.1. A wavelet transform with these filter coefficients can be similarly computed than in case of the Haar-wavelet by low- and highpass filtering of an input signal. The input signal can be a single line of a picture with the dimension N. The two resulting signals are then sampled down - that is, each second value is discarded, and then form the *approximations* and *details*, which are two sets of $N/2$ wavelet coefficients. The approximations $L(i)$ can be computed with

$$L(i) = \sum_{j=-4}^{4} line_{pic}(i+j) \cdot Al(j), \qquad i = 0, 2, 4, \ldots, N-1 \tag{3.5}$$

and the details $H(i)$ with

$$H(i) = \sum_{j=-3}^{3} line_{pic}(i+j) \cdot Ah(j), \qquad i = 1, 3, 5, \ldots, N, \tag{3.6}$$

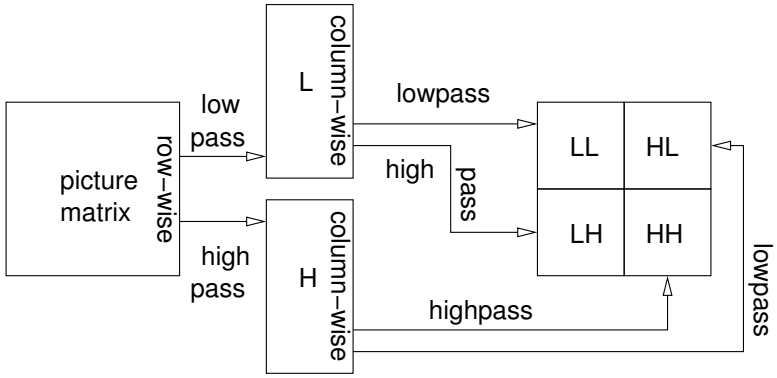


Figure 3.10: One-level picture wavelet transform. The wavelet-picture is partitioned into four regions that are called *subbands*. The picture is first filtered row-wisely resulting in the *L* and *H* matrix, which both have half the dimension of the original picture. Then these matrices are filtered column-wisely resulting into the four subbands. It is also possible to first filter column by column and then line by line. The *HL* subband for instance denotes that first the highpass and then the lowpass was applied. In the literature, the subband *HL* is sometimes denoted by *LH*. Therefore the position of these subbands in the destination matrix is sometimes interchanged.

where $Al(j)$ and $Ah(j)$ denote the filter coefficients given in table 3.1, respectively. Note that the sample down operation can be incorporated by only computing the required coefficients, thereby avoiding useless computation. At the signal boundaries, a symmetrical extension is performed to avoid border effects, e.g., a signal $s(i), i = 0 \ldots N-1$ is extended for the highpass filter as follows:
$s_{ext} = s(3)s(2)s(1)s(0)s(1)\ldots s(N-1)s(N-2)s(N-3)s(N-4)$
The filter is slided over the signal in such a way that the center of the filter is located upside the ith signal value s(i), e.g., for $i = 1$:

Ah(-3)	Ah(-2)	Ah(-1)	Ah(0)	Ah(1)	Ah(2)	Ah(3)
s(2)	s(1)	s(0)	s(1)	s(2)	s(3)	s(4)

For the lowpass filter the center of the filter slides over the even values $i = 0, 2, \ldots, N-1$, for the highpass it moves over the odd values $i = 1, 3, 5, \ldots N$. To achieve multiple levels this process can be repeated. In the next section this method is employed for computing the picture wavelet transform.

3.4.3 Two-Dimensional Wavelet Transform

A wavelet transform of a matrix, i.e. - a transform of a picture, can be achieved by first transforming all the rows of the matrix. For each row a result vector is computed that contains the approximation coefficients of the last level and the detail coefficients of the previous levels. Then each column of the so obtained matrix (which has the same dimension than the original matrix) is again wavelet-transformed.

In this work a different scheme is employed, resulting in a different order of the wavelet coefficients in the transformed matrix. A row or a column of a matrix is not transformed up to its last level, but only for one level. Figure 3.10 illustrates such a one-level wavelet transform. The picture matrix is first processed row-wisely - that is, for each row the approximations and details are computed by lowpass and highpass filtering. This results into two matrices L and H each of them with half of the original picture dimension. These matrices are similarly low- and highpass filtered to result in the four sub-matrices LL, LH, HL, HH, which are called *subbands*:

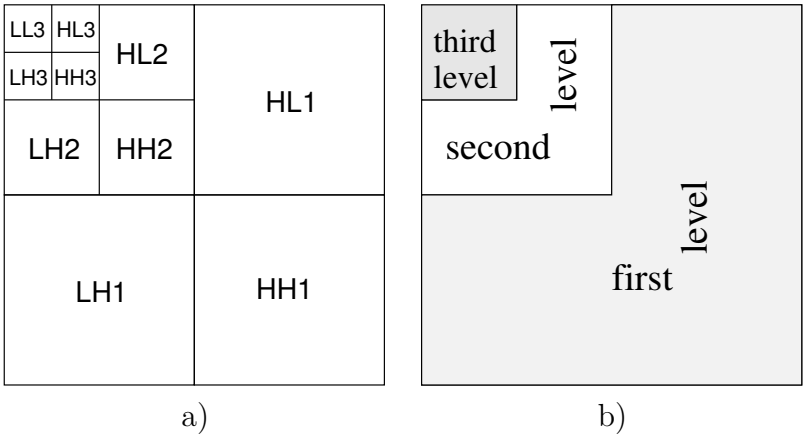


Figure 3.11: Three-level picture wavelet transform. Figure a) illustrates the subbands of each level. For computation of a level the LL-subband of the previous wavelet level is retrieved. In figure b) the areas of each level are hatched.

LL is the all-lowpass (coarse approximation image),

HL is the vertical,

LH is the horizontal, and

HH is the diagonal subband.

To achieve a second wavelet transform level only the LL-submatrix is processed, as illustrated in figure 3.11, and so on. The operations can be repeated on each of the LL subbands to get the higher level subbands. Note that LL represents a smaller and smoothed version of the original image. LH intensifies the horizontal elements, HL the vertical, and HH the diagonal elements. The reconstruction of the original image can be performed similarly. An example of a two-level wavelet transform is given in figure 3.12. In the next section a new technique is introduced that allows to compute such a transform with extremely little memory.

3.5 Fractional Wavelet Filter: A Novel Picture Wavelet Transform

3.5.1 Introduction

This section introduces the *fractional wavelet filter* as a technique to compute fractional values of each wavelet subband, thus allowing a low-cost camera sensor node with less than 2 kByte RAM to perform a multi-level 9/7 picture wavelet transform. The picture dimension can be 256x256 using fixed-point arithmetic and 128x128 using floating-point arithmetic. The technique is applied to the platform given in section 3.3, i.e., a typical 16 bit sensor node architecture with external flash memory. The flash memory allows to line-wisely read and write picture data.

The motivation for designing a wavelet scheme for typical sensor nodes was given in the introduction in section 1: As the bandwidth is very limited in wireless sensor networks, data compression techniques are inevitable when large amounts of data, e.g., pictures, have to be

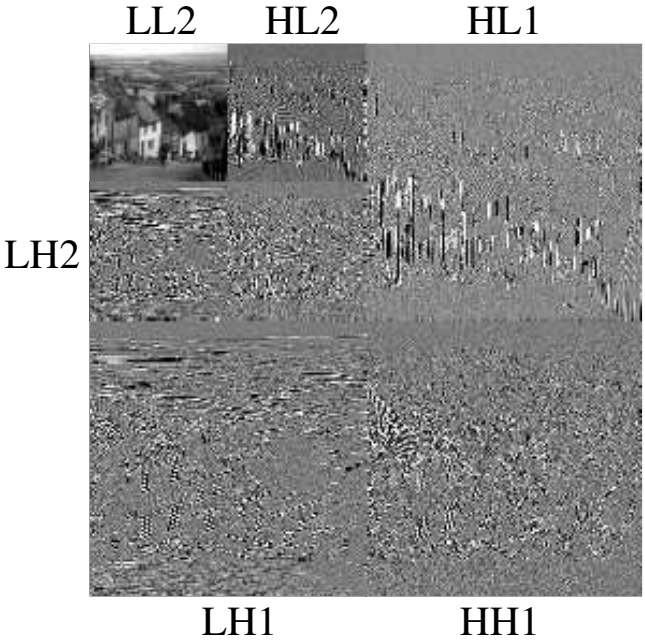


Figure 3.12: An example of a two-level picture wavelet transform. Each one-level transform results into four subbands denoted by LL, LH, HL, HH. The contrast of each subband was adjusted to fill the entire intensity range.

transferred. The wavelet transform is an established decorrelating tool that allows coding algorithms to achieve superior compression gains among current picture compression techniques. However, as the wavelet transform itself is a memory intensive procedure, wavelet based compression has hardly arrived on the low energy consuming microcontrollers that drive low cost sensor nodes. Thus the design of low-memory wavelet transform schemes is still a current research activity, as detailed in section 3.2.

The problem is addressed in this section by introducing a wavelet filter system - that is, the *fractional wavelet filter*, for the Daubechies 9/7 wavelet (also called the *FBI-Wavelet*), which reads and writes the picture data row-wisely and roughly allocates 1.2 kBytes for a fixed-point wavelet transform with the picture dimension N=256. The conceptually less complex 5/3 wavelet transform that works with less coefficients or the integer wavelet transform are not considered here, as these generally give lower compression gains.

The own sensor node where the fractional filter is verified uses the *Microchip dsPIC30F4013*, i.e., a 16 bit digital signal controller with 2 kByte RAM, the camera module C328-7640 with an universal asynchronous receiver transmitter (UART) interface (available at `http://www.comedia.com.hk`), and an external 64 MByte multimedia card (MMC) as a flash memory, connected to the controller through a serial peripheral interface (SPI) bus. As the cost for flash memory has fallen dramatically while size, power consumption, and access times were improved [86], flash memory like a multimedia card may be a standard component of future sensor nodes to store the gathered data and to allow for more complex applications while relieving the wireless network. Such a flash card and the camera module can be connected to any ordinary sensor node with UART and SPI ports. The data of the MMC-card is accessed through an own filesystem [5]. The system is designed to capture still images.

In the next subsection, the novel filter is detailed. In this thesis two forward transforms have been realized, one using floating-point numbers for high precision and another transform using

fixed-point numbers that needs less memory while being computationally more suitable for a 16 bit controller. In subsection 3.5.3, the two implementations for the fractional filter are evaluated on the own hardware concerning picture quality, flash memory access and computation times. In the last subsection the findings are reflected.

3.5.2 Fractional Wavelet Filter

In this subsection the fractional wavelet filter - that is, a computational scheme to compute the picture wavelet transform with very little RAM memory, is explained. The data on the MMC-card can only be accessed in blocks of 512 bytes, thus a sample-wise access as easily executed with RAM memory on PCs is here not feasible. Even if it is possible to access a smaller number of samples of a block, the read/write time would significantly slow down the algorithm, as the time to load a few samples is the same as for a complete block. The fractional filter has to take this restriction into account.

For the first level, the algorithm reads the picture samples line by line from the MMC-card while it writes the subbands line-wisely to a different destination on the MMC-card. Two single lines of the LL/HL subbands and two lines of the LH/HH subbands build a destination line. For the next level the LL subband contains the input data. Note that the input samples for the first level are of the type unsigned character (UCHAR) with 8 bits, whereas the input for the higher levels is either of type floating-point (FLOAT) with 32 bits when using the floating-point filter or signed integers (INT16) with 16 bits when using the fixed-point filter. The filter does not work in place and for each level a new destination matrix is allocated on the MMC-card. However, as the MMC-card has plenty of memory, it does not affect the sensor's resources. This also allows to reconstruct the picture from any level (and not necessarily from the highest level, as it would be necessary for the standard transform outlined in section 3.4). The scheme is illustrated in figure 3.13.

3.5.2.1 Floating-Point Filter

The floating-point wavelet filter computes the wavelet transform with a high precision, as it uses 32 bit floating-point variables for the wavelet and filter coefficients as well as for the intermediate operations. Thus the pictures can be reconstructed without loss of information. The wavelet filter uses three buffers of the dimension N, one buffer for the current input line and two buffers for two destination lines each of them building a row of the LL/HL subbands and a row of the LH/HH subbands. The filter computes the horizontal wavelet coefficients on the fly, while it computes a set of *fractions* (here the fractions are denoted as a part of a sum) of each subband destination line. Let $ll(i,j,k), lh(i,j,k), hl(i,j,k)$, and $hh(i,j,k)$ denote the fractional subband wavelet coefficients, where

- $i = 0, 2, 4, \ldots N/2 - 1$,
- $j = -4 \cdots + 4$ assign the current input line as $i \cdot 2 + j$, and
- $k = 0 \ldots N/2 - 1$ denotes the horizontal coefficient index.

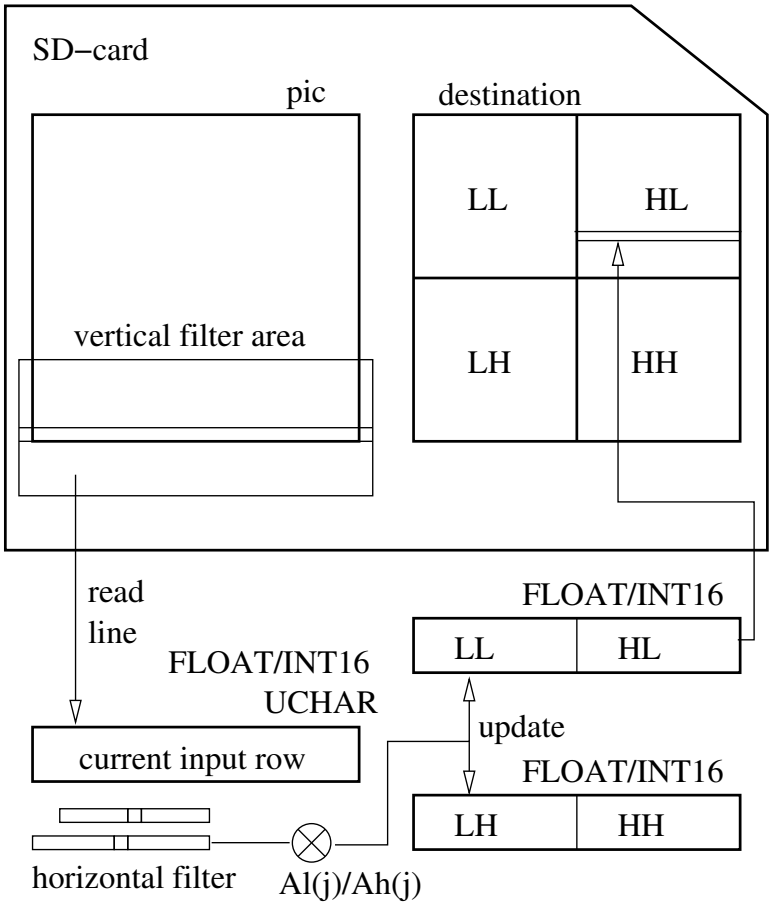


Figure 3.13: Proposed system for the picture wavelet transform. The horizontal wavelet coefficients are computed on the fly and employed to compute the fractional wavelet coefficients that update the subbands. For each wavelet level, a different destination object is written to the MMC-card. Thus the picture can be reconstructed from any level.

The fractional coefficients are defined as follows:

$$ll(i,j,k) = Al(j) \sum_{l=-4}^{4} pic(i \cdot N + k + l) \cdot Al(l) \tag{3.7}$$

$$lh(i,j,k) = Ah(j) \sum_{l=-4}^{4} pic(i \cdot N + k + l) \cdot Al(l) \tag{3.8}$$

$$hl(i,j,k) = Al(j) \sum_{l=-3}^{3} pic(i \cdot N + k + l) \cdot Ah(l) \tag{3.9}$$

$$hh(i,j,k) = Ah(j) \sum_{l=-3}^{3} pic(i \cdot N + k + l) \cdot Ah(l) \tag{3.10}$$

The final coefficients are computed by update operations and thus first have to be initialized: $LL(i,k) = LH(i,k) = HL(i,k) = HH(i,k) = 0 \mid_{\forall i,k}$. For $j = -4 \ldots 4$, the update operations

	floating-point		fixed-point		
N	level	bytes	level	bytes	Format
256		-	1	1280	Q10.5
128	1	1152	2	768	Q11.4
64	2	640	3	640	Q12.3
32	3	320	4	512	Q13.2
16	4	160	5	384	Q14.1

Table 3.2: RAM memory required for the novel fractional wavelet filter. The required bytes are given for each wavelet level using a 128x128 picture for floating-point and a 256x256 picture for fixed-point calculation. The data format for fixed-point calculation is given in the Texas Instruments notation.

are then given as follows:

$$LL(i,k) += ll(i,j,k) \qquad LH(i,k) += lh(i,j,k) \tag{3.11}$$

$$HL(i,k) += hl(i,j,k) \qquad HH(i,k) += hh(i,j,k) \tag{3.12}$$

The special requirement for the fractional filter is that j stays constant for updating all subband rows. The process of updating the destination lines is repeated until the final subband coefficients have been estimated. The pseudo-code of this procedure uses the horizontal filter functions *filtL* and *filtH*. The low- and highpass analysis filter coefficients from table 3.3 a) are accessed through *al(j)* and *ah(j)*, where j denotes the filter index. The floating-point code for the first level is given in the appendix on page 165.

For each vertical filter area, nine input lines have to be read. As the filter moves up by two lines (implicit vertical down sampling), the total number of lines to be read is given as $N/2 \cdot 9$, where N is the picture dimension.

When the filter input area moves up, the input row of the last block can be used for the current filter area, thus reducing the number of repetitive readings. For a picture with dimension N, the number of line readings would reduce to $N/2 \cdot 8$. For simplicity and for code readability, this is not implemented in the fractional filter.

For the higher levels the input data is not the original picture anymore but the previous LL-subband. Thus, the current input row has to be of the type float (32 bit). The number of bytes required for a multi-level wavelet transform of a picture with the dimension N_{pic} is given as

$$Bytes_{float} = \begin{cases} N_{pic}/level \cdot 9, & level = 1 \\ N_{pic}/level \cdot 12, & level > 1, \end{cases}$$

where *level* refers to the required wavelet level.Table 3.2 gives the required bytes for the floating-point implementation with an 128x128x8 input picture for different levels.

3.5.2.2 Fixed-Point Filter

As many other low-cost processors, the dsPIC controller does not give any hardware support for floating-point operations. If they are coded anyway, the compiler translates them to integer operations. Switching an algorithm from floating- to fixed-point can result into time and power savings with

- the cost of less precision,
- the need for a thorough number range analysis,

	analysis lowpass		analysis highpass	
i	real	Q15	real	Q15
0	0.852699	27941	0.788486	25837
±1	0.377403	12367	-0.418092	-13700
±2	-0.110624	-3625	-0.040689	-1333
±3	-0.023849	-781	0.064539	2115
±4	0.037828	1240	0	0

a) Analysis filter coefficients

	synthesis lowpass		synthesis highpass	
i	real	Q15	real	Q15
0	0.788486	25837	0.852699	27941
±1	0.418092	13700	-0.377403	-12367
±2	-0.040689	-1333	-0.110624	-3625
±3	-0.064539	-2115	0.023849	781
±4	0	0	0.037828	1240

b) Synthesis filter coefficients

Table 3.3: 9/7 a) analysis and b) synthesis wavelet filter coefficients in real and Q15 data format. The Q15 data format is a fixed-point representation of real numbers in the range of $[-1\ 0.999..]$ which requires 16 bits.

- and more programming work.

Thus using a fixed-point format for the fractional filter can help to reduce the computational requirements and to reduce the RAM memory needed for the destination subbands. Due to space constraints, the methodology for fixed-point arithmetic is only shortly reviewed here, and the interested reader is referred to the internal report in [87].

Fixed-point numbers are internally stored as an integer number (and the two- complement is used for the bit-wise representation). A binary N-bit fixed-point number, denoted by its integer value a, has $exp(a)$ fractional bits - that are the bits after the radix point, and $N - exp(a) - 1$ integer bits. Note that $exp(.)$ here refers to the exponent and not to the exponential function. A standard notation for such a format is the Texas Instruments Qm.n format, with $m = N - exp(a) - 1$ and $n = exp(a)$. The real number A to be interpreted by the user is then given as

$$A = a \cdot 2^{-exp(a)}. \tag{3.13}$$

The fractional fixed-point filter can then be realized by first transforming the real wavelet coefficients to the Q0.15 format, see table 3.3. The second requirement is that for all add and multiply operations the exponent has to be taken into account. For an add operation, the operands both must have the same exponent. A multiply operation will require a result exponent given by the sum of the input exponents. Both operations can be supported by an exponent change function that is realized by left or right bit-shift. Regarding these requirements, the fixed-point filter can be programmed similarly than the floating-point filter.

As the number range of the wavelet coefficients gets larger from level to level, the output data format has to be enlarged accordingly from level to level. The study in [54] on the required range reports the data formats in table 3.2 to be sufficient. (Note that there is a general difference when the usage of fixed-point numbers for wavelet transforms is discussed in the literature. Sometimes, the fixed-point numbers are only used for the representation of the wavelet coefficients, as it is done in [54]. In this work, when a fixed-point algorithm is discussed, this includes the coefficient representation as well as the internal calculations.) For the first wavelet level, the input data format is Q15.0, as the picture samples are integer numbers. Note that the L and H wavelet coefficients (computed by the functions *filtL and filtH*) are already computed in the data format of the next wavelet level (even if they may need a smaller range than the subband coefficients). The memory requirements for the fixed-point

filter are computed as

$$Bytes_{fixed} = \begin{cases} N_{pic}/level \cdot 5, & level = 1 \\ N_{pic}/level \cdot 6, & level > 1 \end{cases}$$

and are given in table 3.2 for the levels 1-5.

3.5.3 Performance Evaluation

3.5.3.1 Quality Measurements

The floating-point transform gives nearly perfectly reconstructed images (at least for the here employed test set). The fixed-point transform introduces errors through the filter computations.

For the quality evaluation the *GreySet1* test images from the *Waterloo Repertoire* (available at `http://links.uwaterloo.ca/bragzone.base.html`) were chosen. This set contains twelve 256x256x8 greyscale images in the graphics interchange format (GIF), which are depicted in the appendix on page 158. The images were converted to the portable network graphics (PNG) format using the *convert* command of the software suite *ImageMagick* (available at `http://www.imagemagick.org`). Finally, the data was converted to plain text containing unsigned char numbers using the software *Octave*.

The quality measurements evaluate the fractional fixed-point filter computing the peak signal-to-noise ratio (PSNR) to compare the original image with the reconstructed image, which is generated through a fixed-point wavelet transform of the original image followed by an inverse fixed-point wavelet transform. For this purpose a reference software for the transform has been implemented in C. (The reference software served as a basis for the final implementation of the fractional wavelet filter for the camera sensor.) As the reference implementation gives the same results than the final forward fractional filter for the camera sensor, it was employed for the here conducted PSNR measurements. The measurements include six wavelet levels for the given set of pictures. Even if for us there was no quality degradation visible, the picture reconstruction was not lossless. Figure 3.14 demonstrates that the PSNR values for the first level range from 60 to 87 dB, whereas the higher levels give much lower PSNR ranging from 47 to 57 dB. The reason for this difference may be that the input for the first level are integer numbers that are represented without loss of precision. For the levels 2-6 there is roughly a difference of 1-2 dB between the levels. The loss of precision affects the picture quality, but this may be of little interest when a lossy compression algorithm is employed, as for instance the bitplane coding technique SPIHT. In the lossy mode, SPIHT cuts off the least significant bits and thereby can give very high compression ratios.

3.5.3.2 Time Measurements

For the time measurements the forward fractional wavelet filter was employed on the camera sensor for a six-level forward wavelet transform. The 2 kBytes of the sensor's RAM did allow to transform a 128x128x8 picture with floating-point precision and a 256x256x8 picture with reduced fixed-point precision. Table 3.4 gives the times needed for the transform listed by read and write MMC-card access times and processor computing times. The floating-point transform takes 16.18 seconds and the fixed-point transform 12.574 seconds. The fixed-point time measurements reflect the mean value of 20 measurements, where the values are very constant expect of the write access times. A more detailed overview of the categorized times needed for each wavelet level is given in figure 3.15 for the fixed-point transform.

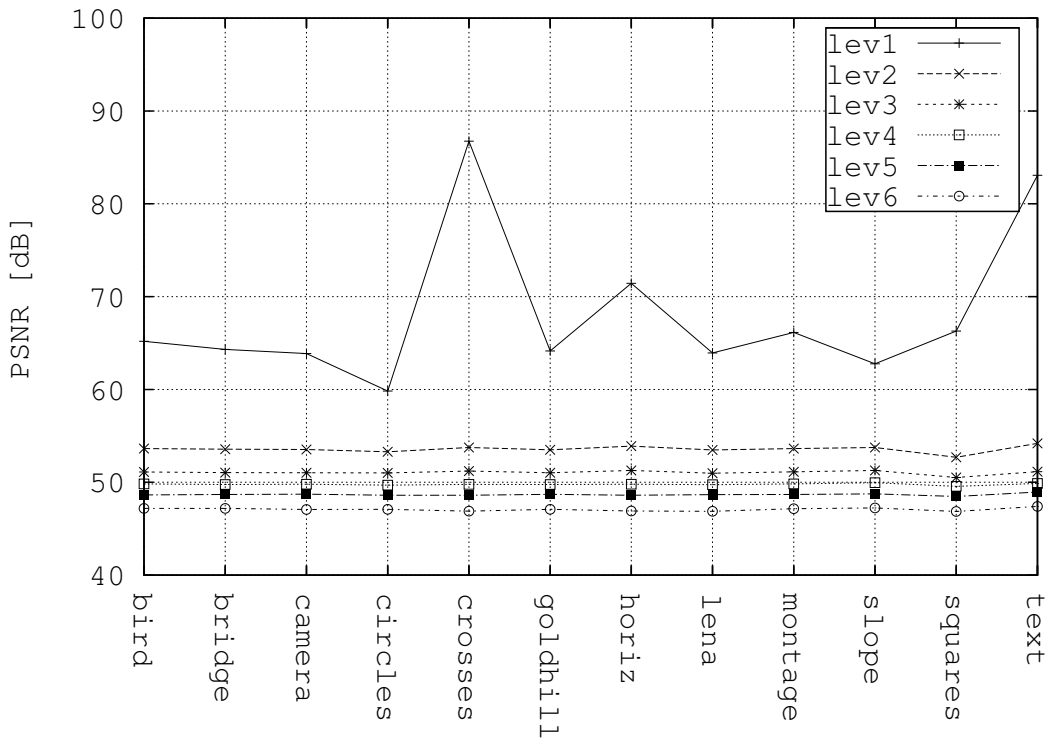


Figure 3.14: PSNR quality measurements for the fractional fixed-point wavelet filter. Even if there was no quality degradation visible, the transform is not lossless. The first level gives very high PSNR values as the transform input values are integers. The quality loss may not affect a lossy compression algorithm as least significant bits are cut anyway.

For each of the given floating-point time values only one measurement was conducted. The floating-point algorithm is very slow because the processor does not support the required high-precision arithmetic. Instead, the operations are realized through heavy compiler support. The floating-point computations are about seven times slower than the fixed-point computations. The computing times take the largest amount of the total time, especially for the floating-point filter. Unexpectedly, the flash memory is not the bottleneck of the fractional filter, even if there is large overhead in the read process, as the rows are read repetitively.

In most applications, the fixed-point algorithm will be preferable as the floating-point algorithm is much slower even for a four times smaller input picture while it needs the same amount of RAM.

	time [sec]			
	T_{read}	T_{write}	$T_{compute}$	T_{total}
float128	1.421	0.5425	14.22	16.18
fix256	2.8646	2.4854	7.2244	12.574

Table 3.4: MMC-card access and processor computing times for a six-level fractional wavelet filter. For the floating-point measurement an 128x128x8 picture and for the fixed-point measurement a 256x256x8 picture were transformed. The floating- point algorithm is very slow because the compiler translates the computations to 16 bit integer operations.

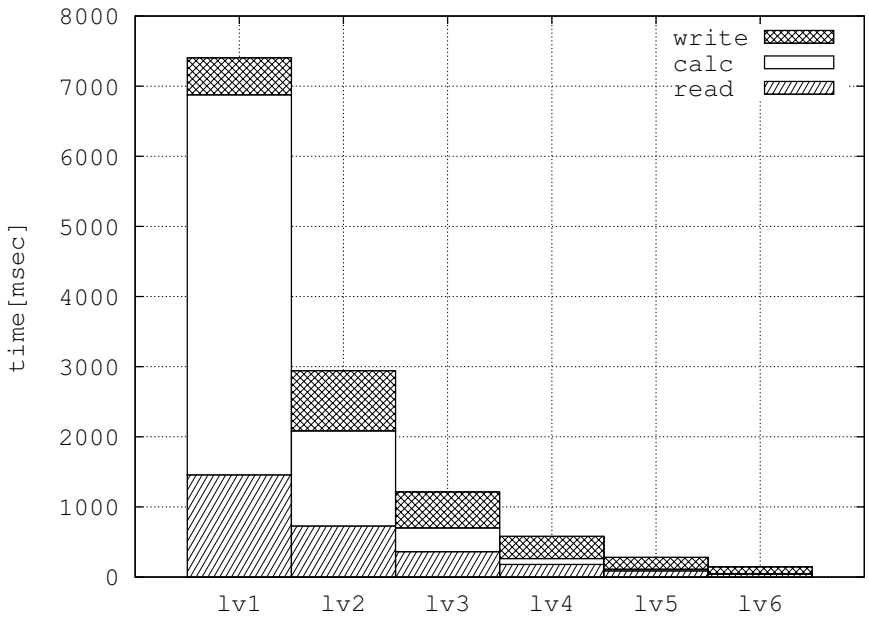


Figure 3.15: Mean values of 20 measurements of the times for performing a fixed-point wavelet transform for levels 1-6 with the introduced fractional wavelet filter. The measurements are conducted using the dsPIC4013 microcontroller set to 29 million instructions per second (MIPS). The times are categorized by read and write access from the multimedia card and by computing times. The largest proportion is taken by the computing times.

3.5.4 Summary

In this section the fractional wavelet filter technique was introduced as a computation scheme that allows the picture wavelet transform to be implemented with very little RAM requirements. The fractional wavelet is applied to a camera sensor node that uses a 16 bit low-cost signal controller with 2 kByte of RAM. The floating-point fractional wavelet transform takes an 128x128x8 picture as an input and needs 16 seconds for six wavelet levels, whereas the fixed-point filter only roughly needs 12 seconds while transforming a 256x256x8 picture.

Even if there is not any quality degradation visible in the pictures, the fixed-point reconstruction is not lossless. However, when a bitplane coding technique is applied the quality may not be affected in the lossy mode as the least significant bits of the coefficients are cut anyway. The fixed-point filter is a preferable choice as the wavelet based coding techniques especially give superior results with high compression ratios.

The fractional filter uses a MMC-card to read the image and to write the wavelet subbands. An external flash memory is a very realistic extension to a camera sensor node, as the data of several pictures has to be stored anyway, including the original and the transformed picture. It is barely feasible to immediately send out the picture to the sensor network, as there might be network congestion or internal ongoing operations with higher priority. The fractional filter operates line by line and thus allows the usage of a filesystem that only can access blocks of 512 bytes. Remarkably, as the time measurements show, the MMC-card is not the bottleneck of the transform but the filter computations.

Some improvements in computing time may result from incorporating the *lifting scheme* [83]. The scheme allows to compute the one- dimensional transform in place. In this case there is only one buffer for the input and output values, where the applied buffer must have the

appropriate size to contain the final wavelet coefficients.

The lifting scheme cannot be applied to the vertical filtering technique of the fractional filter and also not to the horizontal transform of the first level, as the input lines of this level have to use an integer buffer array to not exceed the memory. It can however be applied to the higher levels of the horizontal transform, as the input lines for these levels take variables with the larger data format anyway (and not just 8 bit unsigned char as for the first level input).

Another method to reduce the computational time and to offer an industrial solution may be to use assembly language filter operations.

The wavelet transform does not yet result into data compression. A coding technique for the wavelet coefficients is necessary to exploit the features given by the transform. Thus in the next section the general wavelet picture coding techniques are outlined before a novel coding algorithm for sensor nodes is introduced.

3.6 Coding of Wavelet Coefficients

3.6.1 Introduction

This section reviews different coding techniques for wavelet based picture compression. These techniques process the result of a picture wavelet transform, which is generally given as a matrix with the same size than the original picture. (In this work the word *picture* is sometimes used to refer to the wavelet transformed picture.) As the real wavelet coefficients are rounded, the data format is integer numbers.

The reviewed techniques build up in succession to introduce the recent technique of backward coding that is employed as a starting point for the investigations. Chapter 6.3 in [82] served as a reference for the prior techniques.

Subsection 3.6.2 explains the concept of *Bit-plane encoding*, which allows to progressively encode a picture. This concept is refined in subsection 3.6.3 to form the method *embedded zerotree wavelet* (EZW). EZW introduces the pattern of *zerotrees* within the picture, which are typical for the wavelet transform and can be summarized.

In subsection 3.6.4 the method *set partitioning in hierarchical trees* (Spiht) is outlined as an improved version of EZW. Spiht uses a state transition model to signal the decoder zerotree information when moving from one threshold level to the next. The bits that signal the states may be interleaved by other processing outputs. Spiht achieves superior compression results to JPEG and is a standard in wavelet based image coding. As reported in [85], Spiht is even competitive to the state-of-the-art JPEG2000 compression:

> *The reader should also observe that the objective compression performance of JPEG 2000 is marginally better than that of SPIHT with arithmetic coding and significantly better than the uncoded version of SPIHT. More significantly, the JPEG2000 coder is more robust to changes in image type. In particular, the performance gap between SPIHT and JPEG2000 is much larger for artificial image source than it is for natural imagery.*

Regarding its succinct concept, Spiht is preferable to JPEG2000 for applications on limited platforms.

In section 3.6.5 the recent concept of *backwards coding of wavelet trees* (Bcwt) is described. Bcwt codes the image starting from the lowest wavelet level. Thereby the typical lists of Spiht that can require several MBytes are not needed any more, while the compression is

the same as with Spiht. Bcwt is selected as a starting point for the own investigations. In subsection 3.6.6 performance details are given on the own reference implementation of Bcwt. This implementation serves to verify the concept of backward coding.

The coders Spiht and Bcwt both output a binary stream that can be compressed further with an arithmetic coder. Arithmetic coding on the sensor platform introduced in section 3.3 is discussed in [50] with an implementation of the *range coder* and not in this thesis.

3.6.2 Bit-Plane Encoding

In *bit-plane encoding* the coefficients are transmitted in a progressive way, thus allowing the image to improve over the time. This kind of progressive encoding forms the basis of the algorithms to be described in later sections. Bit-plane encoding is a kind of loop, where a set of coefficients is updated from run to run thus minimizing the quantization errors. In the following the principle of this loop is shortly outlined.

Let $w(m)$ denote the image coefficients and $w_q(m)$ a quantized version of the image. (The image is stored here in a vector.) Let the functions *significance pass* and *refinement pass* denote procedures that analyze the wavelet coefficients $w(m)$ to update the values $w_q(m)$ and to transmit binary coding symbols. Let T_0 define the largest coefficient threshold as

$$T_0 = 2^{\lfloor log_2(max[w(m)]) \rfloor}, \tag{3.14}$$

and T_{min} define the minimum quantization threshold that is the stop condition for the main loop. The operation $\lfloor(\ldots)\rfloor$ here rounds to the nearest integer lower or equal the input, e.g., $\lfloor 5.32 \rfloor = 5$. The main loop for bit-plane encoding is then given as follows:

1. Set threshold $T = T_0$
2. Init $w_q(m) = 0 \; \forall \; m$
3. while $T > T_{min}$ do
   (a) significance-pass $(T, w(m), w_q(m))$
   (b) refinement-pass $(T, w(m), w_q(m))$
   (c) $T = T/2$

The significance and the refinement pass are given as follows.

Significance pass The function *significance pass* analyzes all coefficients that have not yet been quantized. These coefficients still have their initial value of zero. For each value $w(m)$ it is checked if it equals or exceeds the current threshold T. If this is the case the sign is transmitted and the appropriate $w_q(m)$ is set to the current threshold:

$\forall \; w_q(m) == 0$

if $|w(m)| \geq T$
1. Transmit sign of $w(m)$
2. Set $w_q(m) = T$

else transmit zero sign

The sign of $w(m)$ can be coded as 10 for a positive value and as 01 for a negative one. A zero symbol can be coded as 00. The decoder can detect the end of the significance pass as it knows the total number of symbols.

Refinement pass In the refinement pass, the values $w_q(m)$ are refined that have a higher or equal threshold than $2T$. Clearly, the values that already have been set to a threshold in the previous loops, are now updated using a finer T. Note that the values of the previous significance pass are not addressed here (these refer to the current loop and not to the previous one). Also, the refinement pass is skipped for $T = T_0$, as this threshold refers to the first loop. All values $w_q(m)$ are scanned as follows:

$\forall w_q(m) \geq 2T$ do

if $|w(m)|$ is in interval $[w_q(m), w_q(m) + T)$ transmit bit 0

else if $|w(m)|$ is in interval $[w_q(m) + T, w_q(m) + 2T)$

1. transmit bit 1
2. update $w_q(m) = w_q(m) + T$

The decoder can detect the end of the refinement pass as it knows the size of the array of values $w(m)$ that have to be addressed in this pass.

An example for the picture row $w(m) = [53, 10, 26, 20, 63]$ is given as follows:

```
octave:1> pic=[53,10,26,20,63]
pic =

  53  10  26  20  63

octave:2> EncodeW(pic,3)
T_0 = 32
w_q =

  32   0   0   0  32

w_q =

  48   0  16  16  48

w_q =

  48   8  24  16  56

ans =

  1  0  0  0  0  0  0  0  1  0  0  0  1  0  1  0  1  1  1  0  0  1  0  1
```

To be more precise this sequence is explained as follows:

$$\underbrace{1000000010}_{first\ sign.pass}\ \overbrace{001010}^{second\ sign.pass}\ \underbrace{11}_{first\ ref.pass}\ \overbrace{10}^{third\ sign.pass}\ \underbrace{0101}_{second\ ref.pass}$$

In the example the significance pass first quantizes the values 53 and 63 with $T_0 = 32$. Then T_0 is set to 16. In the next loop the values 26 and 20 are set, and the two quantized values of 32 are refined both to 48. In the last loop the value 10 is first quantized to 8. The two values 16 and 48 are refined to 24 and 56. The example is illustrated in figure 3.16.

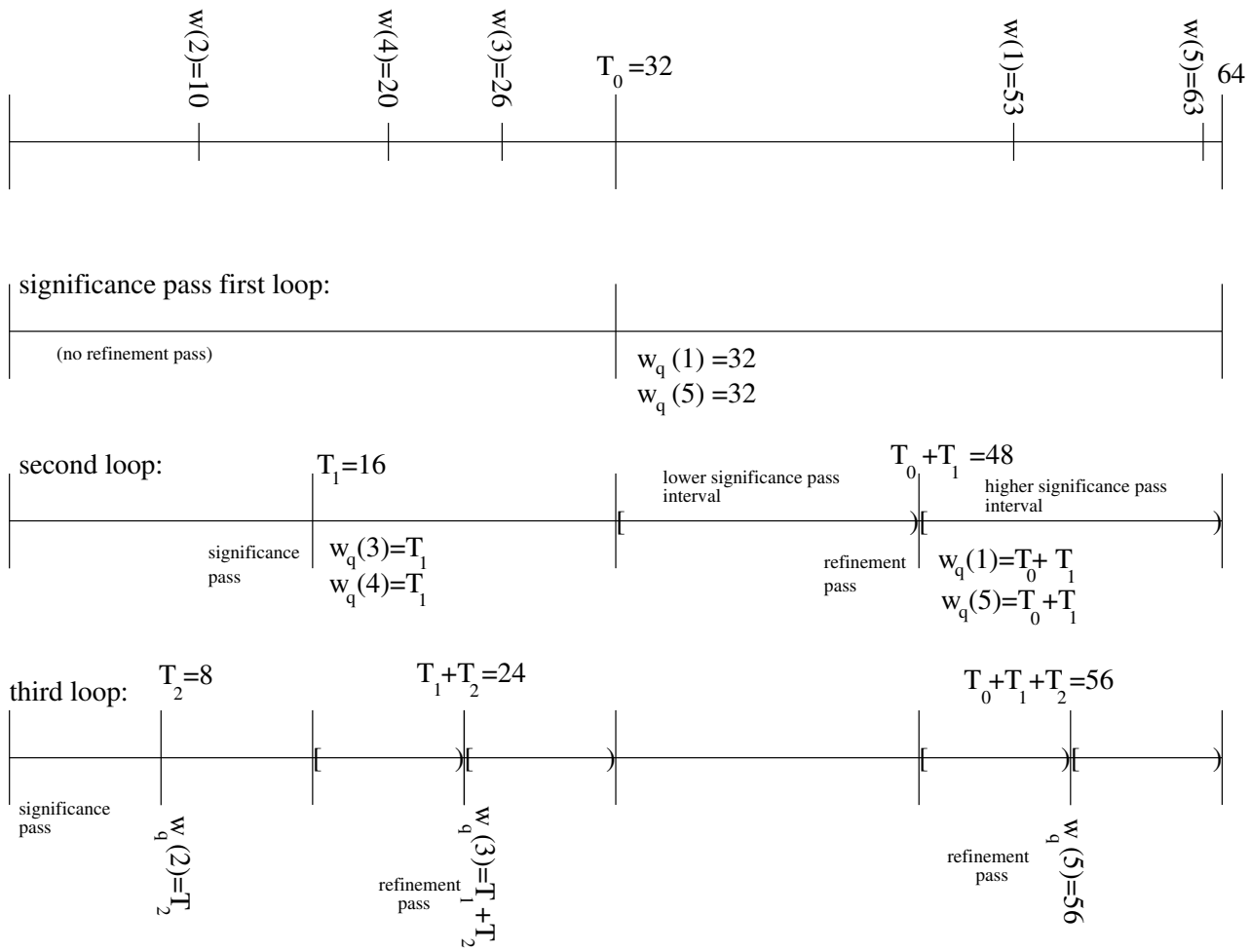


Figure 3.16: Example of a bitplane encoding procedure for the pixel sequence $w(m) = [53, 10, 26, 20, 63]$ using three loops. In the first loop the values $w(1) = 53$ and $w(5) = 63$ are quantized to $T_0 = 32$. In the second loop the values $w(3) = 26$ and $w(4) = 20$ are quantized to $T_1 = 16$ in the significance pass. The two quantized values $w_q(1) = 32$ and $w_q(5) = 32$ are both refined to the value $T_0 + T_1 = 48$. In the third loop $w(2) = 10$ is quantized to $w_q(2) = T_3 = 8$. The quantized values $w_q(3) = 16$ and $w_q(5) = 48$ are refined to the values $16 + 8 = 24$ and $48 + 8 = 56$, respectively. With each of the loops the quantized version of the picture becomes more precise.

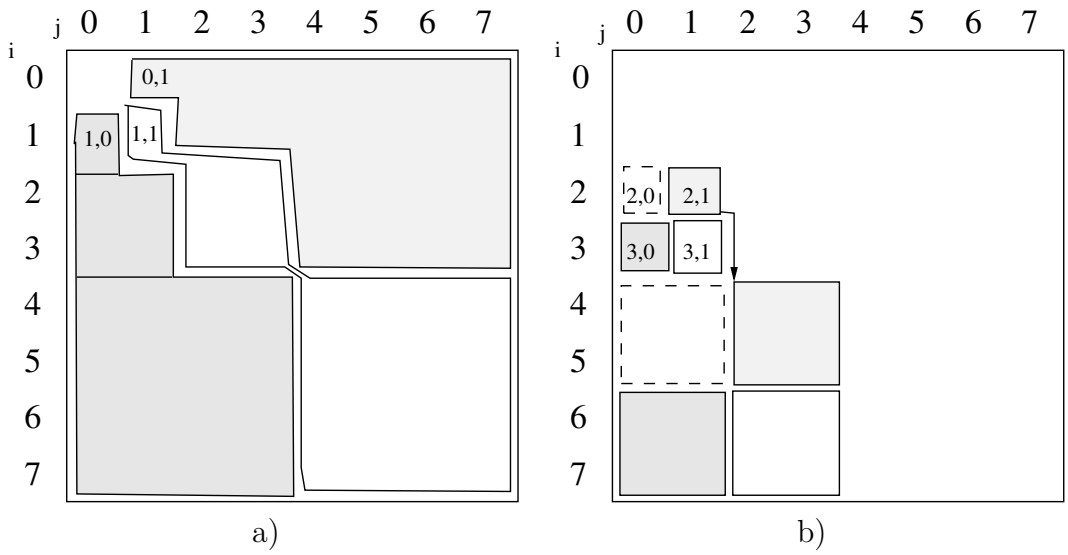


Figure 3.17: Three-level quadtree in figure a) and two-level quadtree in figure b). A quadtree defines a tree with a root node, where each node has four descendants. If each node in the quadtree is insignificant, the quadtree is called a *zerotree*. Zerotrees are summarized in the EZW algorithm, as these structures are typical for a wavelet transformed picture.

3.6.3 Embedded Zerotree Wavelet (EZW)

In this subsection, the bit-plane-encoding procedure is slightly extended to form the *embedded zerotree wavelet* (EZW) algorithm [88]. More precisely, the significance pass is modified in that the locations of insignificant values are summarized using *zerotrees*, thus achieving compression. Remember that an insignificant value has a magnitude smaller than a given threshold T. To explain a zerotree it is useful to first define a *quadtree*.

Quadtree A quadtree is a tree of locations in a multi-level wavelet transform. The tree root [i,j] has four descendants/children, which are located at

$$\begin{array}{ll} [2i, 2j], & [2i, 2j+1], \\ [2i+1, 2j], & [2i+1, 2j+1]. \end{array}$$

as illustrated in figure 3.17. Each of the descendants again has four descendants, and so on. A quadtree is defined by its root node. In figure a), three quadtrees are enclosed with the root nodes at the positions $[1, 0], [0, 1]$, and $[1, 1]$. These quadtrees each have three levels. In figure b), four two-level quadtrees are given, with the root nodes at positions $[2, 0], [3, 0], [2, 1]$, and $[3, 1]$. A zerotree is a quadtree which has insignificant wavelet transform values at each of its locations, i.e., all wavelet coefficients in the tree have lower magnitudes than a given threshold T.

Significance pass EZW For the EZW algorithm the significance pass of the bit-plane encoding procedure has to be refined:

$\forall\ w_q(m) == 0$

if $|w(m)| \geq T$

1. transmit sign of $w(m)$
2. set $w_q(m) = T$

else

if m denotes the element at position [0,0] transmit **I**

else if position m denotes a zerotree transmit **R**

else transmit **I**

The symbol I denotes an *isolated* value, and R denotes the *root* of a zerotree. A possible coding scheme for all symbols of the significance pass is given as follows:

symbol	meaning	bitstream
+	positive sign	10
-	negative sign	01
I	isolated value	11
R	zerotree	00

Please note some details on EZW:

1. The wavelet value at position $[0, 0]$ can only be coded by its sign or as an isolated value. It cannot be a zerotree.

2. The values of a zerotree have to be maintained in a list in order to exclude them for the rest of the scanning procedure with a given threshold T.

3. Similarly than with bit-plane encoding, a value that has already been coded by its sign (+ or -) is excluded in the following significance pass, as its magnitude still will be larger than the current T, which is given by $T_{current} = T_{previous}/2$.

In the next section, an improved version of the EZW algorithm is described.

3.6.4 SPIHT-Set Partitioning in Hierarchical Trees

Set partitioning in hierarchical trees (Spiht) was introduced in [89] by Said and Pearlman. According to the book in [82], some of the best results concerning PSNR values have been measured with Spiht. It is stated that Spiht is probably the most widely used wavelet-based algorithm for image compression and provides a basic standard of comparison for all subsequent algorithms. The idea of Spiht is to encode the wavelet coefficients in a tree more precisely by organizing the insignificant values in larger subsets and using a significance map for efficient coding with binary signals. Arithmetic coding is not needed as the bitstream is already very efficient.

Spiht is a refined version of EZW that gives better compression. Improvements over EZW are achieved by using a *state transition model* [82] for the pixels, where transitions to the next level result in a new state. Such a state model keeps track of the current states of the pixels using lists. The state of a pixel can be *significant root, significant isolated, insignificant root,* or *insignificant isolated.* In contrast to EZW the states are not signaled but their transition. The state *significant root*, which refers to a significant pixel that is the root of a zerotree, was

not present in EZW and leads to further compression. A state transition can be signaled by two bits, one bit, or no bit. The two bits are not sent in a stream.

To explain the algorithm some definitions are given. $w(n)$ denotes a pixel value at the position n. $S_k[J]$ denotes the significance of a set J of pixel values regarding the given threshold T_k:

$$S_k[J] = \begin{cases} 1, & if\ max|w(n)| \geq T_k \qquad \forall\ n \in J \\ 0, & if\ max|w(n)| < T_k \qquad \forall\ n \in J \end{cases} \tag{3.15}$$

J can also refer to a single value m. The sets D(m),C(m), G(m), and H are defined as follows:

D(m)	set of all descendant indices (not only four) of the index m
C(m)	set of the four child indices of the index m
G(m)	D(m)-C(m) = set of grandchildren indices and their descendants of the index m
H	set of indices of the Lth level, where L is the maximum wavelet level

H includes the indices of the wavelet values of all subbands at the maximum level. The all-lowpass subband at this level does not have any descendants. There exist three lists that keep track of the states of the sets of indices:

LIP	list of insignificant pixels
LSP	list of significant pixels
LIS	list of insignificant sets

An index of the list LIS can denote a set D(m) or a set G(m). Thus, each index of the list has an additional entry to denote if the set is of type D or of type G.

Similar than with EZW, the initial threshold is given as

$$T_0 = 2^{\lfloor log_2(max[w(m)])\rfloor}. \tag{3.16}$$

The **main loop** of SPIHT is thus given as follows:

1. Set $T = T_0$
2. Set LIP to H
3. Set LSP to the empty set {}
4. Set LIS to all indices in H that have descendants and set them to type D
5. do while $T > T_{min}$
   (a) Sorting pass $(T, LIP, w(m), w_q(m))$
   (b) Refinement pass $(T, LSP, w(m), w_q(m))$ *//refine pixels in LSP*
   (c) $T := T/2$

Steps 1) - 4) concern some initializations. In step 1) T is set to the value estimated in equation 3.16, which holds true for the initial loop. In step 2) the list of insignificant pixels (LIP) is set to H. H concerns the pixels which are the roots of the zerotrees. Pixels that are in the LIP list either remain there or can move to the LSP list. They are not addressed by the sorting pass. In step 3) it is made sure that the LSP list is empty, and in step 4) the list of insignificant sets (LIS) is set to the roots of the trees with type D. Then in step 5) the loop is started, which calls the sorting pass and the refinement pass to be discussed as follows.

The **refinement pass** is similar to the refinement pass in EZW. All indices m in the list LSP with a threshold $w_q(m)$ higher or equal $2T$ are scanned for a possible update:

$\forall\ w_q(m)$ in LSP $\geq 2T$

if $|w(m)|$ is in interval $[w_q(m), w_q(m) + T)$ transmit bit 0

else if $|w(m)|$ is in interval $[w_q(m) + T, w_q(m) + 2T)$

1. Transmit bit 1
2. update $w_q(m) = w_q(m) + T$

For $T = T_0$ the refinement pass is skipped. The refinement pass actually only works on pixels in the LSP list and refines them in accordance to the current T.

The **sorting pass** is given as described in figure 3.18. It has two main parts - part 1 concerning pixels in the LIP list, and part 2 concerning pixels in the LIS list. In part 1 all pixels in the LIP list are traversed. For each insignificant pixel a 0 bit is sent. Such a pixel remains in the LIP list and is checked again in the next loop for $T = T/2$. If a pixel is significant, however, a 1 bit and a sign bit are sent, and then the pixel is moved to the LSP list. Such a pixel remains in the LSP list and is finally removed from the LIP list.

In part 2 of the sorting pass the trees in the list of insignificant sets (LIS) are addressed. The trees are denoted in the LIS list by their root node. Note that the root node itself is not analyzed in part 2 of the sorting pass. This is indicated by the *type* add on, which either concerns the descendants (type D) or the grandchildren (type G) of the root node.

In part 2(a) the type D is addressed. There it is checked if the tree with the root node in the LIS is a zerotree. (As previously mentioned, the significance of the root node itself is excluded here. Regarding the EZW algorithm, this is a new type of a zerotree.) If it is a zerotree, a 0 bit is sent and the node remains in the LIS list. It will stay there till the next loop with $T = T/2$. If it is not a zerotree, the root node is either removed from the LIS list forever (see 2(a)ii.B) or if there exist grandchildren moved to the end of the LIS list with type G. (The grandchildren will be addressed in part 2(b)) For the children of the root node a 0 bit is sent if they are insignificant or a 1 bit is sent if they are significant (see 2(a)ii.A). If they are significant, they are appended to the LSP list, a sign bit is sent, and they are quantized with the current T. If they are not significant they are appended to the LIP list. Note that the four child pixels are only addressed once in part 2(a) of the sorting pass when $S_k|D(m)| == 1$, as they then move to either the LSP or the LIP list, and the type D root node is then removed. Root nodes of type D remain in the LIS list until the loop where the correspondent tree becomes significant.

If a tree in the LIS list becomes significant, the correspondent root node is moved to the LIS list with type G if grandchildren do exist. These grandchildren are addressed in part 2(b). A type G pixel actually denotes four trees where the roots are given by the children of the type G pixel. If all the four trees are insignificant a zero bit is sent and the type G pixel stays in the list until the next loop with $T = T/2$. If one of the four trees is significant, a 1 bit is sent and the type G pixel is removed from the list. Furthermore the four trees are appended to the LIS list with type D, so that they are further processed in 2(a).

A numerical coding example of the algorithm is given in the appendix on page 143. (The example will be employed in the next subsection for a comparison.)

The Spiht algorithm can require several MBytes of memory to handle the lists. In the next subsection a backward version of Spiht will be introduced that does not need these lists anymore and thus can result into tremendous memory savings.

1. $\forall$ m in LIP do
   //sorting pass insignif. pixels; check if insignif. pixels are still insignif.
   //if the pixel is still insignif. send 0; else send 1 and sign
   (a) output $S_k[m]$
   (b) if $S_k[m] == 1$
      i. move m to end of LSP (and delete it from LIP)
      ii. output sign of $w(m)$
      iii. set $w_q(m) = T_k$
      //else keep m in LIP for next loop with $T = T/2$
2. $\forall$ m in LIS do *//sorting pass insignif. sets*
   (a) if m is of type D (type Child) then do
      //check if $D(m)$ is zerotree; yes: send 0; else send 1
      i. Output $S_k[D(m)]$
      ii. If $S_k[D(m)] == 1$ then
         A. $\forall$ n in $C(m)$ do
            1) Output $S_k[n]$
            2) If $S_k[n] == 1$
            2 a) Append n to LSP
            2 b) Output sign of $w(n)$ *//code signif. child with sign*
            2 c) Set $w_q(n) = T_k$
            else if $S_k[n] == 0$ then append n to LIP
         B. If $G(m)! = \{\}$ then move m to end of LIS as type G
            else remove m from LIS
   (b) else if m is of type G (type Grandchild) then do
      i. output $S_k[G(m)]$*//if grandchildren are zerotrees just code a 0*
      ii. if $S_k[G(m)] == 1$ then
         A. Append the children $C(m)$ to LIS as type D indices *//grandchilds are not zerotrees; to be further processed in 2a)*
         B. Remove m of type G from LIS
         //else m of type G stays in LIS for next loop with $T = T/2$

Figure 3.18: Sorting pass of the SPIHT algorithm. The sorting pass actually reflects the difference to the EZW algorithm. The wavelet trees are more meaningfully analyzed. The procedure contains two main parts - a part 1 where the entries of the list of insignificant pixels (LIP) are traversed, and a part 2 where the entries of the list of insignificant sets (LIS) are analyzed. In this second part the wavelet trees are addressed. A wavelet tree remains in the LIS list until it becomes significant. The part of the tree (starting from the descendants of the root) is coded until insignificant child trees are found, while the current bitplane level (denoted as T) is regarded. The insignificant child trees are inspected by the next loop with $T = T/2$.

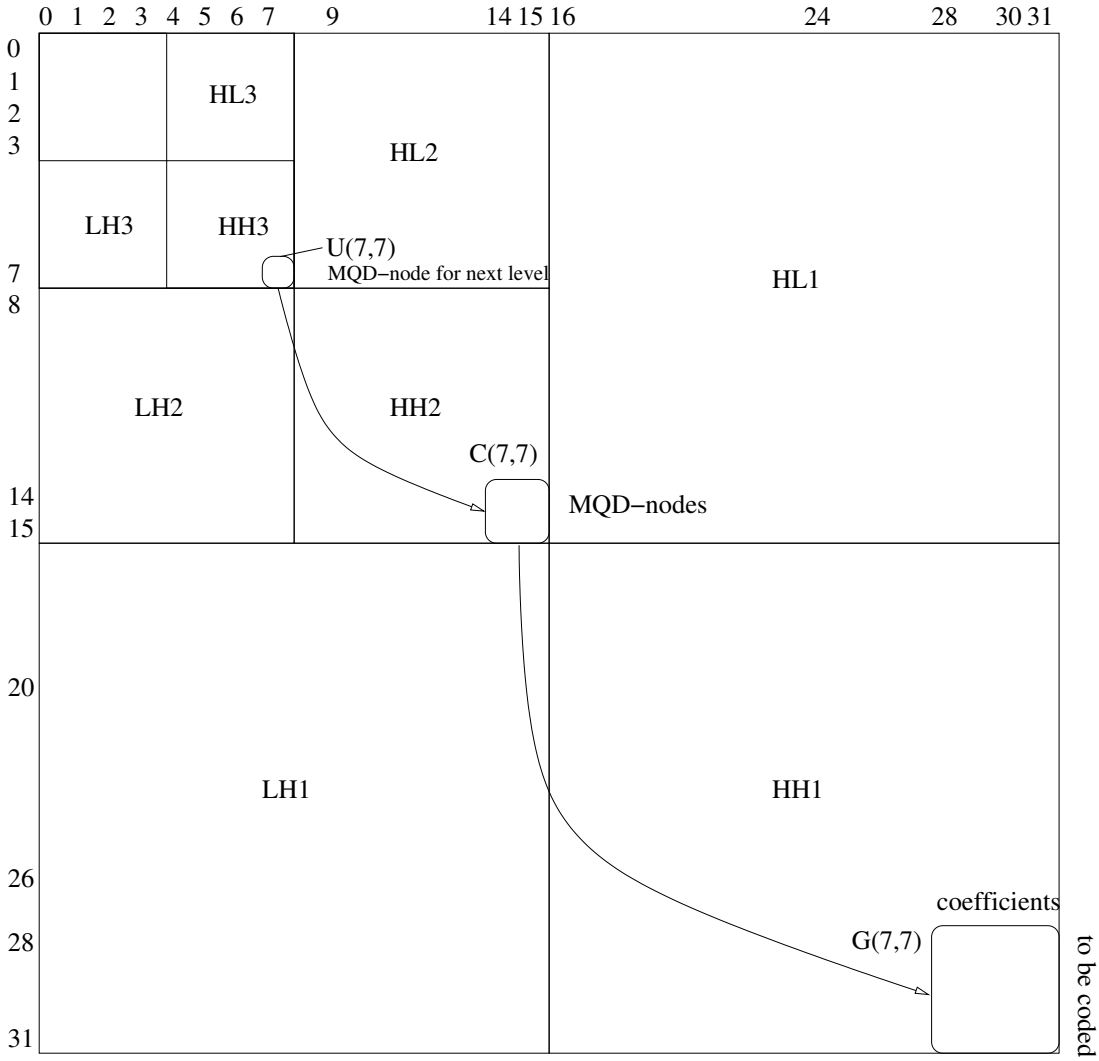


Figure 3.19: Coding the wavelet unit U(7,7) with the Bcwt algorithm. The 16 grandchild-coefficients of the unit node U(7,7) are to be coded. The map of the four MQD nodes m(7,7), m(7,8), m(8,7), and m(8,8) is computed by the previous level units. Bcwt codes all units of all three subbands of a wavelet level and then moves one level up. Instead of the three lists that are needed with the Spiht algorithm, Bcwt only employs a map of MQD nodes thus saving memory.

3.6.5 Backward Coding of Wavelet Trees (Bcwt)

The Spiht algorithm starts coding the wavelet coefficients from the highest level. A zerotree to be detected can span all levels. Thus, all coefficients have to be kept in the random access memory (RAM). In [70], a technique for *backward coding of wavelet trees* (Bcwt) is detailed. Bcwt is different from Spiht in that it starts coding units of 16 wavelet coefficients from the first wavelet level and goes up one level when all the units of the subband have been coded. Bcwt builds a map of *maximum quantization levels of descendants* (MQD). (In a later section these levels are simply denoted as *maximum quantization levels* (MQL) instead.) For one unit five MQD nodes are computed. When the unit is finished, the coded wavelet coefficients are not needed any more, but one of the MQD-nodes has to be stored to be processed by the next level unit.

Figure 3.19 illustrates the principle of coding one wavelet unit. The unit U(7,7) codes the

16 grandchild coefficients G(7,7). When all the units of the HH3 subband have been processed, all wavelet coefficients of the HH1 subband are coded, and the algorithm can move one level up. More details on the steps coding a unit are given in subsection 3.6.5.1. The following notation is employed for describing the Bcwt algorithm. Parts of it were already denoted for describing the Spiht algorithm:

$c(i,j)$	wavelet coefficient at position (i,j)
$C(i,j)$	set of the four child positions of (i,j)
$D(i,j)$	set of all the descendant positions of (i,j)
$G(i,j) = D(i,j) - C(i,j)$	set of the grandchild positions of (i,j)
$q(i,j)$	quantization level of $c(i,j)$, see equation 3.17
$q_C(i,j) = \max\limits_{(k,l)\in C(i,j)} \{q(k,l)\}$	maximum quantization level of the children of (i,j)
$q_G(i,j) = \max\limits_{(k,l)\in G(i,j)} \{q(k,l)\}$	maximum quantization level of the grandchildren of (i,j)
$m(i,j) = \max\limits_{(k,l)\in D(i,j)} \{q(k,l)\}$	maximum quantization level node, see equation 3.18
q_{min}	minimum quantization threshold

The quantization level $q(i,j)$ is given as

$$q(i,j) = \begin{cases} \lfloor \log_2 |c_{i,j}| \rfloor, & |c_{i,j}| \geq 1 \\ -1, & \text{otherwise} \end{cases} \tag{3.17}$$

Remember that $\lfloor(\ldots)\rfloor$ rounds to the nearest integer lower or equal the input. The Bcwt algorithm completely codes the wavelet coefficients and the MQD nodes in one run as well. The MQD nodes are similar to the maximum magnitude coefficients in SPIHT. The Bcwt algorithm codes a map of maximum quantization levels of the tree - that is, the maximum quantization level of all descendants of one wavelet coefficient:

$$m(i,j) = \begin{cases} q_C(i,j), & \text{if (i,j) is in a level 2 subband HL,HH, or LH} \\ \max\limits_{(k,l)\in C(i,j)} \{(m(k,l), q(k,l)\}, & \text{otherwise} \end{cases} \tag{3.18}$$

The computation of the MQD nodes is recursive, as previous MQD-nodes are employed for each calculation.

The binary operations $B[\,]$, $T[\,]$, and $b|_n^m$ are needed for Bcwt coding:

- $B[x]$: binary code of x
- $T[n]$: binary code with a single one at position 2^n, e.g.,$T(3)$=

2^4	2^3	2^2	2^1	2^0
0	1	0	0	0

- $b|n^m$: A binary section of the number b that includes the binary digits from 2^n to 2^m, e.g., $10011|_0^2$=

2^4	2^3	2^2	2^1	2^0
		0	1	1

Bcwt codes wavelet coefficients $c(i,j)$, maximum quantization levels $m(i,j)$, and grandchild maximum quantization levels $q_G(i,j)$. Using the given binary notation, the encoding and decoding of these numbers can be performed as illustrated in table 3.5.

a) Binary coding of numbers:

1. Coding of a coefficient $c(u,v)$:
   if $c(u,v) \geq 0$ output 1; else output 0
   Output $B(|c(u,v)|)|_{q_{min}}^{m(k,l)}$
   First the sign bit is sent followed by the coefficient's binary numbers. q_{min} is the lower quantization level bound of $|c|$, and $m(k,l)$ is the upper quantization level bound of $|c|$, both known by the decoder. Therefore only the coefficient's binary numbers ranging from $2^{q_{min}}$ to $2^{m(k,l)}$ have to be coded.

2. Coding of a quantization level MQD node $m(k,l)$:
   Output $T[m(k,l)]|_{max\{m(k,l),q_{min}\}}^{q_G(i,j)}$
   The number to be coded has a single one at position $2^{m(k,l)}$. The upper bound of this number is $q_G(i,j)$, therefore bits left to this position are not coded. The least significant zero bits are not sent either.

3. Coding of a maximum quantization level $q_G(i,j)$:
   Output $T[q_G(i,j)]|_{max\{q_G(i,j),q_{min}\}}^{m(i,j)}$
   Similar principle than with coding $m(k,l)$. A binary number with a single one has to be coded. $m(i,j)$ is an upper quantization level bound of q_G. The least significant zero bits are not coded.

b) Decoding of binary numbers:

1. Decoding of a coefficient $c(u,v)$:
   The decoder knows that it will receive $m(k,l) - q_{min}$ bits.

2. Decoding $m(k,l)$:
   The decoder can receive a stream of zero bits which are optionally followed by a single one that gives the quantization level $m(k,l)$ and also acts as a stop bit. The number of received bits is $\leq (q_{min} + q_G + 1)$. The first received bit is at the most significant position $2^{q_G(i,j)}$, the second bit at position $2^{q_G(i,j)-1}$, and so on. Thus, the levels of the following bits are known by the decoder.

3. Decoding of $q_G(i,j)$:
   Similar principle than decoding $m(k,l)$.

Table 3.5: Bcwt coding of a number. A coded number can be a wavelet coefficient, a maximum quantization level, or a maximum grandchild quantization level. Encoding of numbers is always performed with respect to a lower and upper bound. The encoder first sends out the least significant bit. As the bitstream is reversed, the decoder's first received bit is the most significant bit. The last received bit is thus the least significant bit. The binary stream is equivalent to the bitstream of Spiht, with the difference that in Bcwt the bits of a coefficient are sent in a stream, whereas Spiht uses a progressive procedure where only a single bit is sent regarding the current T.

3.6.5.1 Coding a Bcwt Unit

The algorithm for coding and decoding a Bcwt unit is given in table 3.6. The encoder starts encoding the units from wavelet level 3. Recall that a unit spans three levels (as illustrated in figure 3.19) where the unit root node is located at the highest level, the four MQD pixels located in the middle level, and the 16 coefficients to be encoded located at the lowest level.

If the unit root node (i, j) is at level 3, the encoder first computes in **step 1)** the four level 2 MQD nodes $m(k, l), (k, l) \in C(i, j)$. At level 3, each of these four MQD nodes only gives the maximum quantization level of one of the four sets of wavelet coefficients. (Note that at higher levels the MQD nodes also include the maximum quantization levels of the complete subtree and are retrieved from the MQD list.) Each of the MQD levels later serves to encode four wavelet coefficients.

In **step 2)** the maximum grandchild quantization level $q_G(i, j)$ is computed as the maximum of the four MQD nodes. This level later serves to encode the MQD levels. In **step 3)** the 16 wavelet coefficients and their successors are checked for significance. In case of no significance the encoder will proceed with step 4). If a significance is detected, the 16 coefficients are processed in four groups. If a single group is significant, the included set of four coefficients is coded using the q_{min} and relevant MQD level.

In **step 4)** the maximum quantization level $m(i, j)$ required for the relevant unit in the next level is computed as the maximum of q_G (computed in step 2)) and the maximum quantization level of the four child pixels of (i, j). The level is stored in a list while the four MQD levels of the current unit are deleted. The computed level will be retrieved by the relevant unit of the next higher wavelet level. The Bcwt unit encoding procedure generally puts one MQD level into the list while it retrieves four MQD levels from the list, see step 1). Each entry of the MQD list contains a position to denote the pixel and a level. The number of MQD levels for one wavelet subband of level lev is given as $N^2 \cdot 2^{-2(lev+2)}$, where N denotes the picture dimension. In **step 5)** the q_G level is encoded using q_{min} and $m(i, j)$.

The encoder can now encode other units of the same wavelet level. For each unit a MQD level for the next higher level is put into a list. There is no specific order required concerning the subbands LH, HL, and HH. Also, the order of units within a subband is not of importance. This does not hold true if the binary stream shall be compressed further.

Concerning the Bcwt unit decoder operation illustrated in table 3.6 b), two issues are discussed in the following that are not explained in the table. The first issue concerns the *stop condition* that signals a zerotree, and the second issue the decoder's coefficient rounding method.

Decoder Stop Condition: When a unit is decoded, it has to decode the four MQD levels for the next lower units and to put them to the list. (Note that the decoder starts at the top wavelet level.) It is possible that the decoder detects a *stop condition* that signals a zerotree and tells the decoder to not deeper scan the tree. There exist two possible stop conditions. The first one can occur when $m(i, j) < q_{min}$, which is detected in step 1) of the algorithm, see the line

1. If $m(i, j) \geq q_{min}$ decode $q_G(i, j)$.

The second stop condition is located in step 2) of the algorithm:

2. If $q_G(i, j) \geq q_{min}$ do: decode the 16 grandchildren of (i, j)

If $q_G(i, j) < q_{min}$ the function for decoding a unit can stop. In case of a stop condition the MQD levels for the next lower units are not decoded. Even if a zerotree was detected, the next lower level units will still ask the list for the relevant MQD levels.

Two options are considered here to solve this problem. First, it is possible to generate and

a) Coding a unit:

1. If (i,j) is in level 3 subband, compute the four level 2 MQD nodes
2. Compute $q_G(i,j) = \max_{(k,l)\in C(i,j)} \{m(k,l)\}$
3. If $q_G(i,j) \geq q_{min}$ do $\forall(k,l) \in C(i,j)$ {
   //do for each group of coefficients:
   (a) if $m(k,l) \geq q_{min}$ do $\forall(u,v) \in C(k,l)$ {
   //do for all coefficients in this subgroup:
   i. If $q(u,v) \geq q_{min}$ output sign $c(u,v)$
   ii. Encode the coefficient $c(u,v)$:
   Output $B\{|c(u,v)|\}|_{q_{min}}^{m(k,l)}$
   }
   (b) Encode $m(k,l)$:
   Output $T[m(k,l)]|_{max\{m(k,l),q_{min}\}}^{q_G(i,j)}$
   }
4. Compute $m(i,j) = \max\left\{q_{G(i,j)}, \max_{(k,l)\in C(i,j)}\{q(k,l)\}\right\}$
   and put the result to the MQD list.
   //This MQD node is computed for the next level unit;
   Delete the MQD nodes $m(k,l)\forall(k,l) \in C(i,j)$
5. If $m(i,j) \geq q_{min}$ encode q_G:
   output $T[q_G(i,j)]|_{max\{q_G(i,j),q_{min}\}}^{m(i,j)}$

b) Decoding a unit:

1. Get $m(i,j)$ from MQD list and remove it from the list. If $m(i,j) \geq q_{min}$ decode $q_G(i,j)$ using $m(i,j)$
   //else all descendant coefficients remain zero, as all destination matrix elements were initialized with zero→stop condition for this tree;
2. If $q_G(i,j) \geq q_{min}$ do $\forall(k,l) \in C(i,j)$:
   //else stop decoding this tree
   (a) Decode $m(k,l)$ using $q_G(i,j)$ and put it to the list
   (b) If $m(k,l) \geq q_{min}$ do $\forall(u,v) \in C(k,l)$
   i. Decode the coefficient $c(u,v)$ using $m(k,l)$ and q_{min}
   ii. Compute $q(u,v)$
   iii. If $q(u,v) \geq q_{min}$ decode the sign of $c(u,v)$

Table 3.6: Coding a Bcwt Unit U(i,j). The encoder starts from the level 3 subband. Encoder and decoder are illustrated side by side to clarify their interdependence. The decoder backwards goes over the actions of the encoder, e.g., step 5) of the encoder is processed by step 1) of the decoder. A list of maximum quantization levels helps to keep track of the current maximum level in the tree, thus only a small subpart of the picture needs to be kept in memory.

initialize the four child MQD-nodes that the lower level decode-unit functions will see. This can be done when a decode unit function is called. When the relevant unit at the next lower level is to be decoded, the function will hit on an initialized MQD level and immediately return without decoding operation. The second option would be to not initialize the levels and to let the lower level functions not find any node in the list, so that the zerotrees can be assumed. This would give the drawback that all MQD-nodes in the list have to be searched.

Decoder Coefficient Rounding Method: In figure 3.5 b) the process of decoding a wavelet coefficient is given. To improve the quality of the decoded picture, the decoder has to perform a rounding method as it is detailed in [82] on page 285:

> *It is standard practice for the decoder to then round the quantized values to the midpoints of the intervals that they were last found to belong to during the encoding process (i.e.,add half of the last threshold used to their magnitudes).*

Rounding is of importance in the lossy mode when a pixel to be decoded has lost some bits due to the given threshold q_{min}. In this case, the error of the coefficient has to be moved to the midth - that is, rounding each transform value into the midpoint of the bin in which it lies.

In the next subsection an example to demonstrate how Bcwt codes a unit is given.

3.6.5.2 Numerical Coding/Decoding Example

In this section a numerical example for coding a Bcwt unit is given using the numerical values given in figure 3.20. The operations with $q_{min} = 2$ (that gives a minimum quantization level of $2^{q_{min}} = 4$) are given as follows:

1. Compute the level 2 MQD-Nodes:
   $m(0,6) = max\{5,3,2,2\} = 5$
   $m(0,7) = max\{5,2,3,2\} = 5$
   $m(1,6) = max\{2,3,3,2\} = 3$
   $m(1,7) = max\{-1,0,1,1\} = 1$

2. Compute $q_G(0,3) = max\{5,5,3,1\} = 5$

3. $q_G \geq q_{min} \Rightarrow$ Encode the coefficients in four groups:
   First Group:

   (a) $m(0,6) = 5 \geq q_{min} \Rightarrow$ significant group
   Encode c(0,12): Output sign bit 1;Output $B(39)|_2^5 = 100111|_2^5 = 1001$
   Encode c(0,13): Output sign bit 1;Output $B(15)|_2^5 = 1111|_2^5 = 0011$
   Encode c(1,12): Output sign bit 0;Output $B(4)|_2^5 = 100|_2^5 = 0001$
   Encode c(1,13): Output sign bit 0;Output $B(7)|_2^5 = 111|_2^5 = 0001$

   (b) Encode $m(0,6)$: Output $T(5)|_5^5 = T(100000)_5^5 = 1$

   Second Group:

   (a) $m(0,7) = 5 \geq q_{min} \Rightarrow$ significant group
   Encode c(0,14): Output sign bit 0;Output $B(34)|_2^5 = 100010|_2^5 = 1000$
   Encode c(0,15): Output sign bit 0;Output $B(7)|_2^5 = 111|_2^5 = 0001$
   Encode c(1,14): Output sign bit 0;Output $B(9)|_2^5 = 1001|_2^5 = 0010$

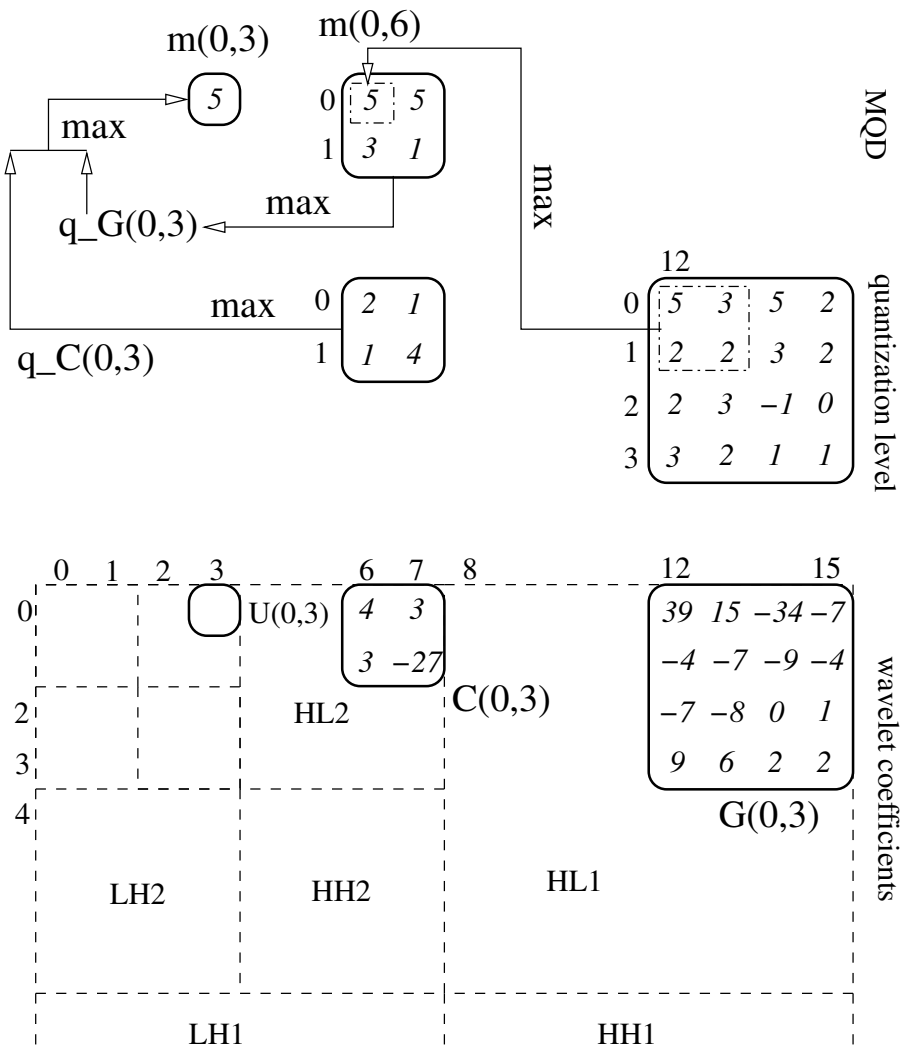


Figure 3.20: Numerical example for encoding the 16 coefficients of the Bcwt unit U(0,3). The picture at the bottom illustrates the pixels of a transformed picture, and the picture at the top the quantization levels of the pixels. Note that for retrieving the MQD levels $m(0,6), m(0,7), m(1,6)$, and $m(1,7)$ the MQD list cannot be employed as it is done for the higher levels, as the lowest level wavelet coefficients are to be encoded. The $m(0,3)$ level is the maximum of q_G and the four child levels of the root. It is stored in the list to be retrieved by the relevant next level unit.

Encode c(1,15): Output sign bit 0;Output $B(4)|_2^5 = 100|_2^5 = 0001$

(b) Encode $m(0,7)$: Output $T(5)|_5^5 = 1$

Third Group:

(a) $m(1,6) = 3 \geq q_{min} \Rightarrow$ significant group
Encode c(2,12): Output sign bit 0;Output $B(7)|_2^3 = 111|_2^3 = 01$
Encode c(2,13): Output sign bit 0;Output $B(8)|_2^3 = 1000|_2^3 = 10$
Encode c(3,12): Output sign bit 1;Output $B(9)|_2^3 = 1001|_2^3 = 10$
Encode c(3,13): Output sign bit 1;Output $B(6)|_2^3 = 0110|_2^3 = 01$

(b) Encode $m(1,6)$: Output $T(3)|_3^5 = 1000|_3^5 = 001$

Fourth Group:

(a) $m(1,7) = 1 < q_{min} \Rightarrow$ insignificant group
$\Rightarrow$ do not code this group

(b) Encode $m(1,7)$: Output $T(1)|_2^5 = 1|_2^5 = 0000$

4. Compute $m(0,3) = max\{2,1,1,4,5\} = 5$

5. $m(0,3) \geq 2 \Rightarrow$ Encode q_G output $T(5)|_5^5 = 1$

The resulting bitstream is given as:

$$\underbrace{\overbrace{1}^{sign}\ \overset{LSB\rightarrow MSB}{1001}}_{c(0,12)}\ \underbrace{1\ 1100}_{c(0,13)}\ \underbrace{0\ 1000}_{c(1,12)}\ \underbrace{0\ 1000}_{c(1,13)}\ \underbrace{1}_{m(0,6)} \qquad \textit{first group}$$

$$\underbrace{0\ 0001}_{c(0,14)}\ \underbrace{0\ 1000}_{c(0,15)}\ \underbrace{0\ 0100}_{c(1,14)}\ \underbrace{0\ 1000}_{c(1,15)}\ \underbrace{1}_{m(0,7)} \qquad \textit{second group}$$

$$\underbrace{0\ 10}_{c(2,12)}\ \underbrace{0\ 01}_{c(2,13)}\ \underbrace{1\ 01}_{c(3,12)}\ \underbrace{1\ 10}_{c(3,13)}\ \underbrace{100}_{m(1,6)} \qquad \textit{third group}$$

$$\underbrace{0000}_{m(1,7)} \qquad \textit{fourth group}$$

$$1 \qquad q_G(0,3)\ \textit{for all groups}$$

Note that for each number the least significant bits are written at first. The bitstream is reversed so that for each number the decoder first receives the most significant bits. Note that $m(0,3)$ is in memory and larger than q_{min}, thus the first step is to decode q_G. The received bitstream is given as

$1 \hat{=} q_G$	$m(0,3) \geq q_{min}$ in memory $\Rightarrow$ decode $q_G(0,3) = 6$
$\underbrace{0000}_{m(1,7)}$	decode m(1,7)=1 for the fourth group using q_G
	$m(1,7) < q_{min} \Rightarrow$ fourth group is insignificant
	$\Rightarrow$ next decode third group
$\underbrace{001}_{m(1,6)}\ \underbrace{01\ \underbrace{1}_{sign}}_{c(3,13)}\ \underbrace{10\ 1}_{c(3,12)}\ \underbrace{10\ 0}_{c(2,13)}\ \underbrace{01\ 0}_{c(2,12)}$	*third group*
...	...

The bitstream is decoded as detailed in table 3.6 b).

Now that the bitstream for the example picture in figure 3.20 for Bcwt has been explained, it makes sense to compare the bitstream of Bcwt with the corresponding bitstream of the Spiht algorithm given in the appendix on page 143. The binary output of Spiht for the example pixels is given as follows:

	T=32	$T = 16$	$T = 8$	$T_{min} = 4$	quantized value
pixel (0,12)=39	11 → LSP	0	0	1	$w_q(0,12) = 36$
pixel (0,13)=15	0 → LIP	0	11 → LSP	1	$w_q(0,13) = 12$
pixel (1,12)=-4	0 → LIP	0	0	10 → LSP	$w_q(1,12) = -4$
pixel (1,13)=-7	0 → LIP	0	0	10 → LSP	$w_q(1,13) = -4$

Now the Spiht output is compared with the relevant Bcwt output:

$$\underbrace{\overbrace{1}^{sign}\ \overbrace{1001}^{LSB\rightarrow MSB}}_{c(0,12)}\ \underbrace{1\ 1100}_{c(0,13)}\ \underbrace{0\ 1000}_{c(1,12)}\ \underbrace{0\ 1000}_{c(1,13)}\ \underbrace{1}_{m(0,6)} \quad \textit{first group}$$

More specifically, if for instance the output of pixel (0,13) is compared, the binary output of Spiht is 0 0 11 1, while the output for Bcwt is 1 11 0 0. The Bcwt output is just a reversed version of the Spiht output with the difference that Bcwt gives out all bits in a stream, whereas Spiht only sends one bit per loop to allow for a progressive transmission.

Bcwt and Spiht further code level information. Level information concerning a group of four coefficients is coded as follows, see figure 3.18 for Spiht and table 3.6 for Bcwt:

for Spiht:

2.(a) if m is of type D (type Child) then do
//check if D(m) is zerotree; yes: send 0; else send 1
i. Output $S_k[D(m)]$

for Bcwt:

3.(b) Encode $m(k,l)$:
Output $T[m(k,l)]|_{max\{m(k,l),q_{min}\}}^{q_G(i,j)}$

Spiht here signals for the current T if any of the four children or their descendants is significant. This gives the decoder the information to decode the four children and to check if the grandchildren or their descendants are significant. The signaled bit is actually one bit of the respective maximum quantization level. The Bcwt algorithm instead signals the concrete maximum quantization level with respect to q_{min} and the map quantization level q_G, as it does not operate for one given threshold T but for the range of all possible Ts.

A maximum quantization level concerning a group of 16 coefficients and their descendants is coded for Spiht and Bcwt as follows:

for Spiht:

2.(b) i. output $S_k[G(m)]$ *//if grandchildren are zerotrees just code a 0*

for Bcwt:

5. If $m(i,j) \geq q_{min}$ encode q_G:
output $T[q_G(i,j)]|_{max\{q_G(i,j),q_{min}\}}^{m(i,j)}$

The principle is similar to the level information concerning a group of four coefficients. Whereas Spiht only codes one bit with respect to the current threshold T, Bcwt codes all bits of the maximum quantization level regarding the map level information and q_{min}. Spiht and Bcwt both code a maximum quantization level map, where Spiht in contrast to Bcwt works forwards

and progressively. Both coders rely on the principle that the pixels are organized in trees, where the values tend to become smaller with more distance to the root.

3.6.5.3 Main Loop and Coding at highest Level

The Bcwt coding procedure is introduced by Guo et. al. in [70], where precise information is given about coding one Bcwt unit. The overall coding procedure is described by the authors as follows:

> *In the overall encoding process, BCWT simply starts coding the units at level 3; after all of them are coded, BCWT moves one level up; BCWT repeats until the top level is reached; then BCWT encodes the coefficients in the LL-band using simple uniform quantization, the same method used in SPIHT.*

According to this description, a main loop for the encoder can look as follows:

1. for level=3:MaxLevel do
   (a) Code all units in HL(level)
   (b) Code all units in HH(level)
   (c) Code all units in LH(level)
2. Code all Coefficients in the LL(MaxLevel) band using uniform quantization

Using this main loop, the coefficients in the HL, HH, and LH subbands coefficients in the levels $MaxLevel$ and $MaxLevel - 1$ are not coded: Each unit spans three levels, and only the coefficients of the lowest level are coded. When a unit U(i,j) is coded at the highest level, the four coefficients at C(i,j) as well as the root coefficient at (i,j) have to be coded. Furthermore, the MQD nodes of the highest HL, HH, and LH subbands are not coded, see figure 3.21.
To solve this problem the procedure *CodeLast2Levels()* is proposed here to be called when the units at the highest levels have been coded. The procedure encodes

- the children coefficients of the last MQD nodes,
- the last MQD nodes, and
- all the coefficients of the highest subbands, which are all the coefficients of set H.

Note that for each unit one MQD node is in memory. The quantization level $q_G(0)$ and the MQD node $m(0)$ are introduced as follows:

$q_G(0)$**:** Maximum of the MQD nodes at the top level in the HL,HH, and LH subbands

$m(0)$**:** Maximum quantization level of all wavelet coefficients

q_{max}**:** Maximum quantization level of all pixels in the given set of pictures

For simplicity two sets are employed:

H**:** Set of indices in all subbands at the Lth level, where L is the maximum wavelet level

$H_{\setminus LL}$**:** Set of positions H excluding the LL subband

a) Encoding:

1. Compute the maximum of the $m(k,l)$ nodes:
   $q_G(0) = \max_{\forall(k,l)\in H_{\setminus LL}} m(k,l)$
2. Do $\forall$ unit nodes $(k,l) \in H_{\setminus LL}\{$
   (a) If $m(k,l) \geq q_{min}$ encode the child coefficients of the $m(k,l)$ nodes:
   Do $\forall(u,v) \in C(k,l)$
      i. If $c(u,v) \geq q_{min}$ output sign(c(u,v))
      ii. Encode c(u,v) using m(k,l):
      Output $B\{|c(u,v)|\}|_{q_{min}}^{m(k,l)}$
   (b) Encode $m(k,l)$ using $q_G(0)$:
   Output
   $T[m(k,l)]|_{max\{m(k,l),q_{min}\}}^{q_G(0)}$
   Remove $m(k,l)$ from list.

   $\}$
3. Compute $m(0) = max\{q_G(0), \max_{\forall(k,l)\in H}\{q(k,l)\}\}$
4. Encode $q_G(0)$ using q_{min} and $m(0)$:
   Output $T[q_G(0)]|_{max\{q_G(0),q_{min}\}}^{m(0)}$
5. Encode all coefficients of the highest subbands:
   Do $\forall(u,v) \in H$ $\{$
   (a) Output sign(c(u,v))
   (b) Encode c(u,v) using m(0):
   Output $B\{|c(u,v)|\}|_{q_{min}}^{m(0)}$

   $\}$
6. Encode m(0) using q_{min} and q_{max}, where q_{max} is a given constant:
   Output $T[m(0)]|_{max\{m(0),q_{min}\}}^{q_{max}}$

b) Decoding:

1. Decode $m(0)$ using q_{max} and q_{min}
2. Decode all coefficients of H using $m(0)$ and q_{min}
3. Decode q_G using $m(0)$
4. Do $\forall$ nodes at (k,l) in $H_{\setminus LL}$
   (a) Decode $m(k,l)$ using $q_G(0)$ and add it to the list
   (b) If $m(k,l) \geq q_{min}$ decode all children coefficients of position (k,l)
5. Delete the level 2 MQD nodes

Table 3.7: Procedure *CodelLast2Levels* for the encoder (figure a)) and the decoder (figure b)). This function is proposed to be called after the top Bcwt units have been coded. The function encodes the child coefficients of the last unit nodes (2.(a)), the maximum quantization levels for these nodes (2.(b)), the maximum $q_G(0)$ of these levels (4.), the coefficients of the highest subbands (5.), and the maximum quantization level $m(0)$ of all given pixels in the picture (6.).

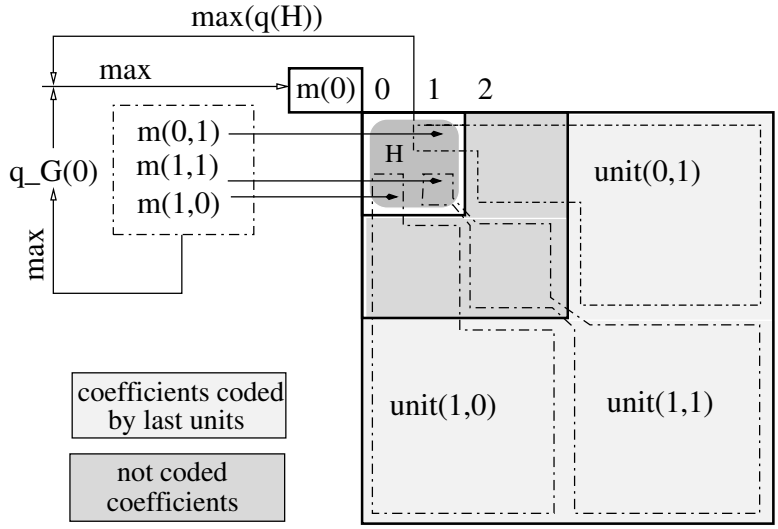


Figure 3.21: Problem of not coded coefficients: The last three Bcwt-units u(0,1), u(1,0), and u(1,1) do not code the children coefficients of positions (0,1), (1,0), and (1,1), and also do not code the coefficients of the set H (the nodes at the highest subbands). Furthermore, the MQD nodes at (0,1), (1,0), and (1,1) are not coded. A solution is given by the proposed function *CodeLast2Levels()*, which is detailed in table 3.7.

The procedure for coding and decoding the last two levels is detailed in table 3.7. The function exactly encodes the remaining levels and coefficients. A simpler solution would be to assume a fixed maximum quantization level for the remaining levels and to use this fixed value to encode the remaining coefficients.

As the Bcwt algorithm serves as a starting point for the own investigations an own implementation of it was realized. In the next subsection an evaluation of Bcwt and thus a verification of this implementation is performed. The implementation itself is not discussed because it exactly reflects the here extensively detailed Bcwt algorithm.

3.6.6 Verification of the own Bcwt Reference Implementation

In this subsection the own Bcwt reference implementation is verified by comparing the performance results with a given Spiht coder. The evaluation is conducted for two methodologies - first the transform is computed using floating point numbers (with Octave) and second the own fixed-point wavelet transform is employed. The wavelet transformed picture is compressed and decompressed by the Bcwt algorithm using different parameters. The inverse transform then reconstructs the picture to be compared to the original picture by a quality measure.

For the quality evaluation the *GreySet1* test images from the *Waterloo Repertoire* were employed equally as it was done in section 3.5. This set contains twelve 256x256x8 greyscale images in the graphics interchange format (GIF). The pictures were converted to the portable network graphics (PNG) format using the *convert* command of the software suite *ImageMagick*. Finally, the data was converted to plain text with signed char numbers (INT8) using the software *Octave*. The data was then transformed with levels $2 \ldots 7$ for the floating-point evaluation and with levels $3 \ldots 6$ for the fixed-point evaluation.(The measurements for the floating-point evaluation reveal that the levels $3 \ldots 6$ are sufficient for the fixed-point evaluation, as lower or higher levels give no reasonable compression or compression improvements.) Bcwt compressed

wavelet level	wavelet values min	max	ExpFirst=5 Qn.m	range	ExpFirst=6 Qn.m	range
0	-128	127				
1	-845	915.7	Q10.5	-1024..1023.97	Q9.6	-512..511.98
2	-1402	1409	Q11.4	-2048..2047.94	Q10.5	-1024..1023.97
3	-1648	2267	Q12.3	-4096..4095.88	Q11.4	-2048..2047.94
4	-2856	4172	Q13.2	-8192..8191.75	Q12.3	-4096..4095.88
5	-5355	8039	Q14.1	-16348..16383.5	Q13.2	-8192..8191.75
6	-1.125e+04	1.574e+04	Q15.0	-32768..32767	Q14.1	-16384..16383.5
7	-1.81e+04	2.392e+04				
8	-4.037e+04	4.562e+04				

Table 3.8: Minimum (min) and maximum (max) values of the complete set of transformed reference pictures for different wavelet levels $lev = 0 \ldots 8$. The fixed-point evaluation of the Bcwt reference implementation is performed for two different number format configurations, the $ExpFirst = 5$ for using the Q10.5 format for the first wavelet level, and the $ExpFirst = 6$ for using the Q9.6 format for that level. When moving to the next higher level the exponent is reduced by one bit to account for the larger number range.

and decompressed the pictures to be reconstructed by the inverse transform. The reconstructed picture was compared to the original by computing the peak signal-to-noise ratio (PSNR). The measurements were repeated for the minimum quantization levels $q_{min} = 0 \ldots 8$. The same PSNR computation was done using the Spiht executable programs *fastcode* and *fastdecd* from Amir Said and William A. Pearlman, available at `http://www.cipr.rpi.edu/research/SPIHT/spiht3.html`, where the previously achieved bits per byte (bpb) rates from Wi2l for the different q_{min}s were taken as an input parameter for the given *fastcode* program.

Note that the roughly outlined methodology that was given in figure 3.8 (section 3.4, page 75) for computing a PSNR value holds true for the floating point transform. For the fixed-point transform the methodology is refined. Importantly, the Bcwt algorithm expects all the coefficients in the same integer data format (rounded real values), so that a rescaling procedure as illustrated in the appendix in figure 5.14 (on page 160) is required. If each wavelet level would have different scaling factors, it would not be possible to exploit the zerotrees.

For the fixed-point wavelet transform the Qn.m data format for the different levels of the transform has to be considered similarly as it was done in section 3.5. Recall that m in Qn.m denotes the number of bits reserved for the *exponent*. A larger exponent gives more precision while the possible range of numbers is reduced. The exponent $ExpFirst = 5, 6$ is denoted here to define the number format of the coefficients in the first wavelet level. Table 3.8 gives the measured range of numbers for the set of the 12 pictures side by side with the fixed-point number range of the relevant data formats.

The results of the evaluation for Bcwt with the floating-point transform compared to Spiht are given in figure 3.22. The range of wavelet levels $lev = 2 \ldots 7$ was selected to find a meaningful level for the given picture dimension of 256x256. From the figure it can be concluded that a level of 5 is sufficient to nearly give the same performance than Spiht. For the very high qualities some Bcwt lines seem to be broken or not complete. This is due to perfectly reconstructed pictures that gave an infinite PSNR value. The next step is now to look at the achieved performance for Bcwt with a fixed-point transform, as this may allow to perform the complete compression procedure using integer numbers.

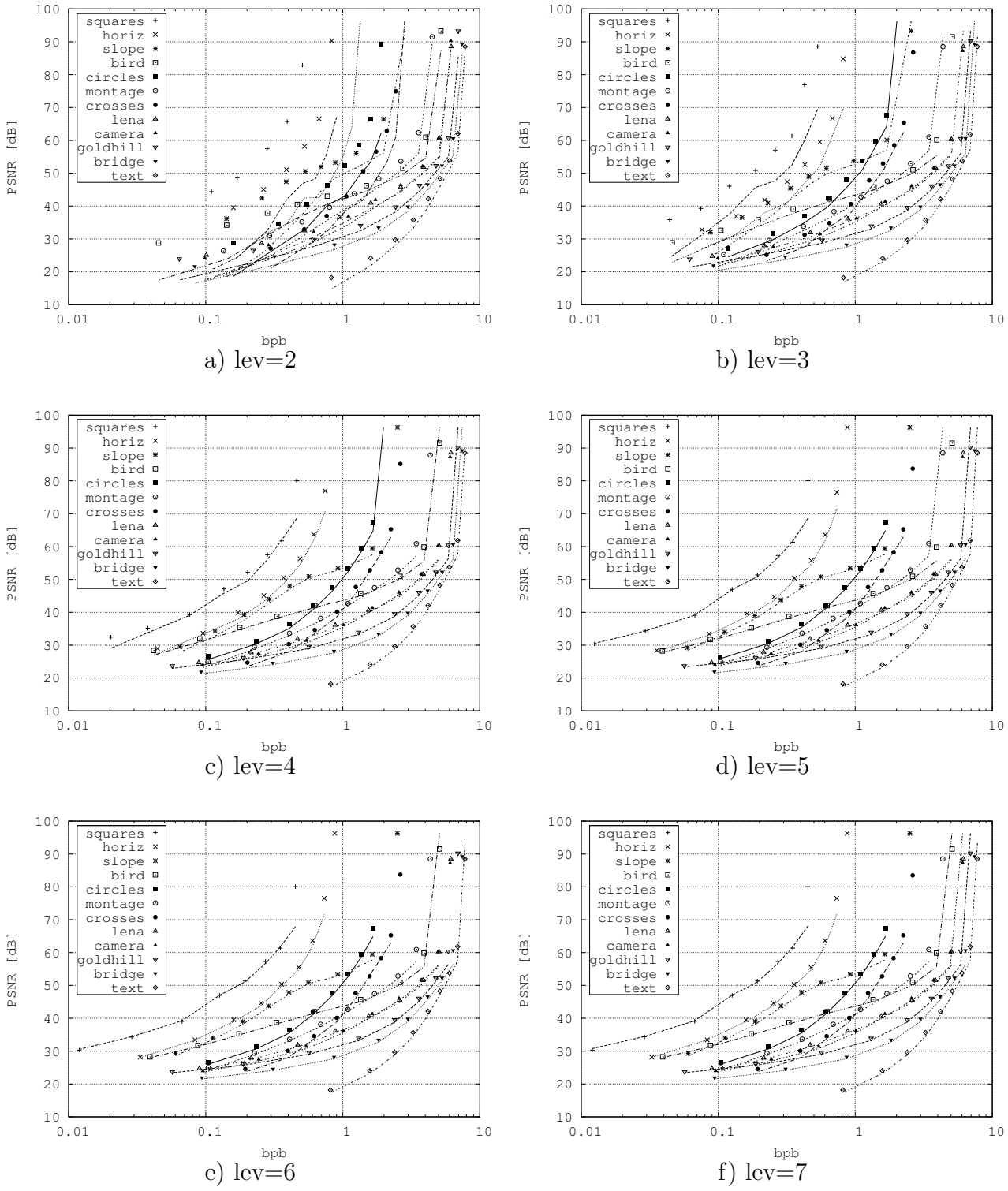


Figure 3.22: PSNR picture quality over bits per byte (bpb) for the own Bcwt reference implementation using the **floating-point** wavelet transform compared to a given Spiht software. The wavelet levels $lev = 2 \ldots 7$ were considered. The lines refer to Bcwt with the different quantization levels $q_{min} = 0 \ldots 8$ while the dots likewise refer to Spiht. The levels 5 and 6 almost give identical performance than Spiht.

The performance of Bcwt with a fixed-point wavelet transform is compared to Spiht for wavelet levels $lev = 4, 5, 6$ in figure 3.23. The measurements are given for the exponents $ExpFirst = 5$ and $ExpFirst = 6$ concerning the first wavelet level, recall table 3.8. As it was already measured for the floating-point transform, wavelet levels 5 and 6 are required to give competitive results. The achieved picture qualities for these levels are almost identical than for Spiht for the higher compression rates. For level $lev = 5$ and the first exponent $ExpFirst = 5$ (figure c)) qualities up to 48 dB are achieved, whereas for $ExpFirst = 6$ (figure d)) the quality goes up to 53 dB. The similar effect is visible for wavelet level $lev = 6$, where for $ExpFirst = 5, 6$ the qualities arrive at 47 dB (figure e)) and 51 dB (figure f)), respectively. The chosen number format $ExpFirst = 6$ is more precise and results in better qualities for low compression rates, even if the range of possible numbers of the format Q9.6 is not sufficient for all given samples in wavelet level 1. The higher wavelet level $lev = 6$ results into marginal improvements for the high compression rates.

In this subsection the performance of the backwards coding algorithm Bcwt was verified (with an own implementation) to give the same results than Spiht. This finding is actually in accordance with the principle coding techniques of Spiht and Bcwt, which were extensively discussed in this section in that their bitstreams were demonstrated to be highly related. The question to be answered now is if it is possible to further reduce the required memory by designing a novel backwards coding technique. This would allow for wavelet-based picture compression on extremely limited platforms.

3.7 A Novel Coding Algorithm for Picture Compression

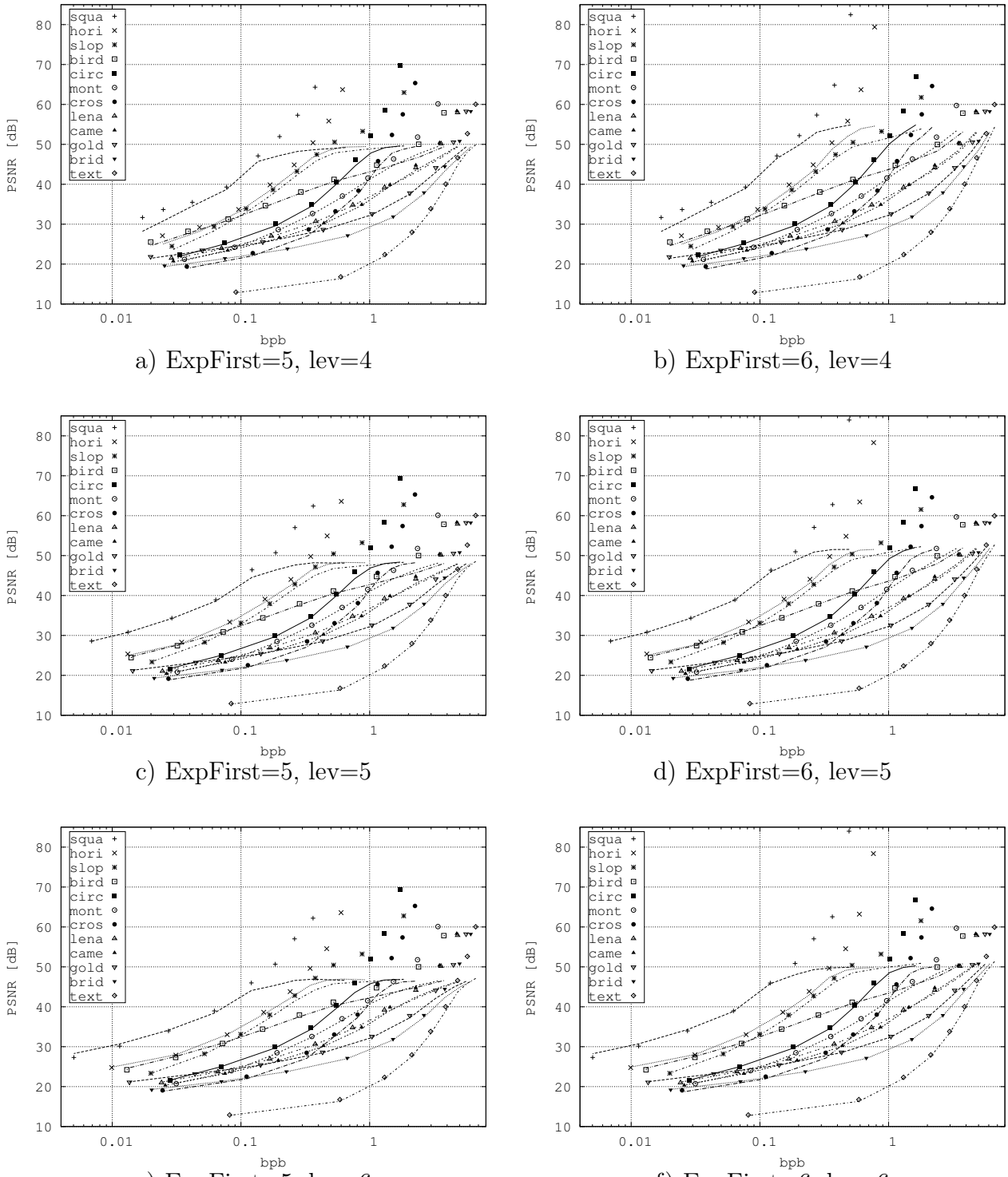
3.7.1 Introduction

A wireless sensor network consists of small devices that typically are equipped with a low-complexity microcontroller. These systems can gather and transfer environmental information, e.g., temperature, acceleration, and sound. Pictures are rarely taken by these low-cost systems, as the controllers are considered to be of too low complexity to process picture data and the network itself is designed for very small data rates. A low-complexity picture compression technique that gives good quality for high compression rates may solve these problems in that it prevents network congestion and reduces transmission times. The compression can also save energy within the network, as the coding energy is lower than the transmission energy.

In section 3.5 the fractional wavelet filter was introduced to allow the computation of a picture wavelet transform with very little memory and 16 bit integer calculations. For picture compression on a small camera sensor node an appropriate wavelet coding technique is obligate that similarly as the fractional filter fulfills the low-complexity requirement.

In the previous section the technique of backwards coding was verified to give promising compression results while the memory requirements are very low compared to Spiht. Spiht can require several MBytes of memory only for its lists, which are necessary to support its progressive feature. Whereas Spiht codes the picture forwards and thus retains the complete picture in memory for a zerotree detection, the technique of backwards coding only requires a subpart of the picture to be kept in memory. This subpart and a required list of maximum quantization levels would allocate approximatively 20 kByte of memory when a greyscale picture of 256x256 8-bit pixels is to be compressed.

The given requirement for the camera sensor node detailed in section 3.3 is that the compression should allocate less than 2 kByte, which reflects a further memory reduction of ten



a) ExpFirst=5, lev=4

b) ExpFirst=6, lev=4

c) ExpFirst=5, lev=5

d) ExpFirst=6, lev=5

e) ExpFirst=5, lev=6

f) ExpFirst=6, lev=6

Figure 3.23: PSNR picture quality over bits per byte (bpb) for the Bcwt reference implementation using the **fixed-point** wavelet transform compared to a given floating-point Spiht software. The lines refer to Bcwt with the different quantization levels $q_{min} = 0 \ldots 8$ while the dots likewise refer to Spiht. The two exponents $ExpFirst = 5, 6$ give the fixed-point number format of the first wavelet level. For the higher compression rates and wavelet levels $lev = 5, 6$ Bcwt almost achieves identical picture qualities than Spiht. The novel algorithm to be introduced in section 3.7 achieves identical compression while the memory requirements are tremendously lower than for Bcwt.

times. The 2 kBytes shall include a buffer for reading or writing pixels of the transformed picture, a buffer for reading or writing compressed binary data, and all the required memory for intermediate data and local variables. In ideal case the buffers for reading and writing data should have a size of 512 bytes, as the memory card (a standard multimedia card (MMC) or secure digital (SD) memory card) can only read blocks of 512 byte.(Such a memory card can of course also access a block of a much smaller size than 512 bytes, or even only access a single pixel. The time however for retrieving this pixel will be the same as for a complete block of 512 bytes.) The data of the memory card should be accessed line by line.

This section details a novel *wavelet image two-line coder* (Wi2l) coder that is designed to fulfill the memory constraints of a typical wireless sensor node. The algorithm operates line-wisely on picture data stored on the sensor's flash memory card while it requires approximatively 1.5 kByte RAM to compress a monochrome picture with the size of 256x256 bytes. Wi2l needs ten times less of memory than the Bcwt algorithm as it does not code the typical units of 16 coefficients. Instead it codes sets of four coefficients of two lines of a wavelet subband, while the amount of required intermediate data is reduced by a recursive function, see figure 3.24. The function stores intermediate data of these lines in a very small buffer with a fixed dimension. The achieved data compression rates, however, are the same as with the Spiht and the Bcwt algorithm.

The section is organized as follows. In the next subsection the notation is resumed. The novel coding algorithm Wi2l is detailed in subsection 3.7.3. The core of the algorithm is a recursive function which reads two lines of a wavelet subband. The function is called by a main loop that codes each wavelet subband separately and finally calls a procedure for encoding the last two levels. In subsection 3.7.4 the compression of the algorithm is evaluated by a PC-implementation in C, while time measurements are conducted on the own hardware platform from section 3.3. This hardware is actually a typical low-complexity wireless sensor node that is extended with a small camera and a standard MMC memory card. The data of the MMC card is accessed through the library in [5]. In the last subsection the findings are summarized.

3.7.2 Binary Notation and Quantization Levels

In this subsection the notation of backwards coding is shortly resumed, i.e., the coding of coefficients and levels within a wavelet tree. A coefficient c is coded using a minimum quantization level q_{min} and a maximum quantization level m_i. q_{min} is a constant threshold and determines the compression rate. If $\mathrm{B}(c)|_a^b$ denotes the binary sequence of a coefficient c starting at the ath and ending at the bth right most bit, c is coded as $\mathrm{B}(c)|_{q_{min}}^{m_i}$, where m_i denotes the maximum quantization level relevant for the current set of coefficients. As c can be negative, a sign bit precedes this binary sequence. Coding a level q_G with m_i as an upper bound is done by writing the binary output $\mathrm{T}(q_G)|_{\max\{q_G,q_{min}\}}^{m_i}$, where $\mathrm{T}(q_G)$ denotes the bit sequence of $\mathrm{B}(2^{q_G})$. Encoding a quantization level m_i is done similarly by encoding the bit sequence $\mathrm{T}(m_i)|_{\max\{m_i,q_{min}\}}^{q_G}$, where q_G denotes an upper bound of m_i. Note that the quantization level for a coefficient c is given as $\lfloor \log_2 |c| \rfloor$ if $|c| \geq 1$ and as -1 otherwise. Levels require unsigned 8-bit variables (INT8) while coefficients require signed 16-bit variables (INT16).

The section also uses the typical Spiht notation given in section 3.6 in that a coefficient $c(i,j)$ has four children $C(pos(i,j))$ at the positions $(2i,2j),(2i,2j+1),(2i+1,2j),(2i+1,2j+1)$. The level m_i denotes the maximum quantization level (MQL) of set i of four coefficients and all the descendant coefficients including the children, children of the children, and so on. (Note that the MQL levels here are equivalent to the MQD levels in Bcwt.) A quantization level

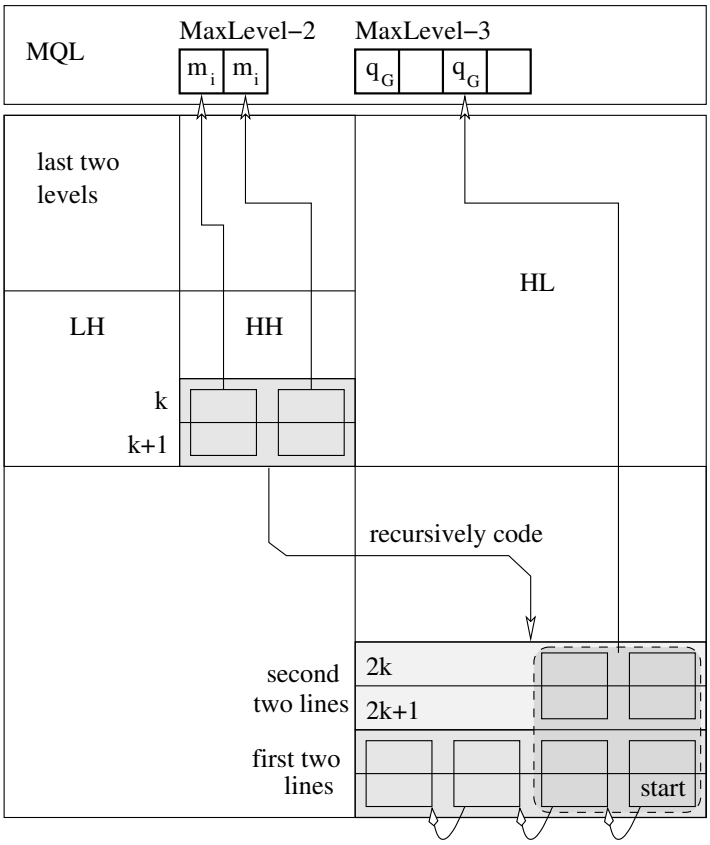


Figure 3.24: Principle of the novel Wi2l coder. The algorithm encodes blocks of four coefficients within two lines, where intermediate level information is stored in a small buffer. Such a buffer exists for each wavelet subband from the third highest level. Generally, the first two lines put some information into the buffer, i.e., maximum quantization levels (MQL) m_i of sets of four coefficients. This information is retrieved and updated by the second two lines within a subband. The updated information concerns maximum quantization levels q_G of 16 coefficients. The relevant two lines of the next higher level of a wavelet subband retrieve this information for the own encoding of their sets of four coefficients. The recursive nature of Wi2l ensures that the buffers always contain the relevant information. More specifically, the lines k and $k+1$ first require the encoding of lines $2k \ldots 2k+3$.

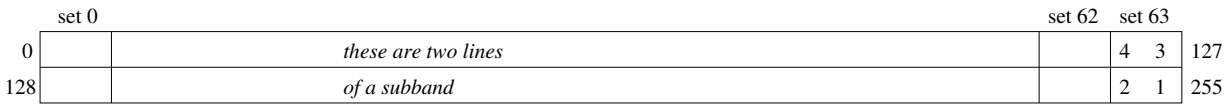


Figure 3.25: Buffer employed by the recursive *Code2Lines()* function of the Wi2l coder to store two lines of a wavelet subband. The data is line-wisely loaded from the flash memory card of the sensor. For a picture dimension of $N = 256$ the buffer requires 512 bytes (256 coefficients each of them taking 16 bits). The function here processes the sets $63, 62, \ldots, 0$ of four coefficients.

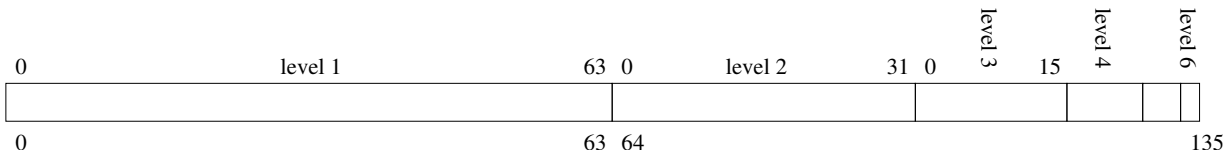


Figure 3.26: Data structure for the maximum quantization level (MQL) buffer. This buffer stores the required intermediate data of the recursive Wi2l coder, i.e., quantization levels m_i for four coefficients and levels q_G for 16 coefficients. The figure illustrates the required fields for a picture dimension of $N = 256$. For each wavelet level i there exist $N/2^{i+1}$ entries of one byte. In the C-implementation the buffer is accessed through the functions *PutMql(level, position)* and *GetMql(level,position)*, which can set and retrieve quantization levels from different wavelet levels at different positions, where the position number is in accordance to the set index of the line buffer illustrated in figure 3.25.

q_{Gi} is the maximum quantization level of a set of 16 coefficients (that are the children of the coefficients in set i) and all their descendants.

3.7.3 Wavelet Image Two-Line (Wi2l) Coding Algorithm

3.7.3.1 Coding of Two Lines

The core of the novel algorithm is a recursive function that codes two lines of a wavelet subband. Three buffers - the line buffer for image data, the maximum quantization level (MQL) buffer, and a buffer for compressed binary data are employed within the function. The line buffer illustrated in figure 3.25 is loaded with two lines from a wavelet subband located on the flash memory. Its dimension is given as $N \cdot 2$ bytes, where $N \cdot N$ is the picture dimension. The MQL buffer maintains maximum quantization levels of the coefficients and is illustrated in figure 3.26. Its size in bytes for a picture dimension N is given as

$$size[bytes] = \sum_{i=1}^{\log_2(N)-2} \frac{N}{2^{i+1}} \tag{3.19}$$

A function $q_{Gi}(level, n)$ retrieves the nth element of the buffer for a given level. A third buffer to read or write the binary sequence is needed, which is set to 512 bytes. As each recursive function call puts variables onto the processor's stack, 256 extra bytes for stack content are estimated. This results into a total memory of 512+512+126+256=1406 bytes for the picture dimension $N = 256$.

Now the function given in table 3.9 is explained in detail. The function is called with parameters specifying the two lines to be coded. These include the subband given as HL, LH

Code2Lines(PMemoryCard,subband,level,k) {
//code line (k) and line (k+1):

1. Encode (level-1) lines at (2k+2) and (2k): *//will put q_G (level-1) into buffer*
   if (level>1)
   (a) Code2Lines(PMemoryCard,subband,level-1,(2k+2))
   (b) Code2Lines(PMemoryCard,level-1,(2k))
2. Read lines (k) and (k+1) from the card into the line buffer
3. DimMql=DimPic/$2^{level+1}$
   for TwoSets=DimMql-1:-2:1 *//start at right-hand side*
   (a) for i=1:-1:0 *//set 1 is right-hand set*
   set(i)=TwoSets+i-1
      i. if (level>1) get q_{Gi}(level-1,2·set(i)) from buffer
      else $q_{Gi} = -1$
      ii. Compute m_i
      $m_i = \max\left\{q_{Gi}, \max_{\forall j \in set(i)}\{q_j\}\right\}$
      iii. if ($m_i \geq q_{min}$)
         A. If (level>1) encode q_{Gi} (level-1) using m_i:
         output $T[q_{Gi}]|_{\max(q_{Gi},q_{min})}^{m_i}$
         B. Encode set(i) of 4 coefficients using m_i:
         $\forall$ j in set(i) do {
         if $q(c_j) \geq q_{min}$ output sign c_j
         output $B\{|c_j|\}|_{q_{min}}^{m_i}$}
   (b) if k/2==odd {Write $m_{0,1}$ to the buffer to positions (TwoSets) and (TwoSets-1)}
   else *//second two lines*
      i. Get $m_{2,3}$ (previously denoted as $m_{0,1}$) of previous two lines from buffer
      ii. Compute q_G:
      $q_G = \max\{m_0, m_1, m_2, m_3\}$
      iii. If $q_G \geq q_{min}$ encode $m_{0...3}$ using q_G:
      for i=3:-1:0 output $T[m_i]|_{Max(m_i,q_{min})}^{q_G}$
      iv. Write q_G to buffer on left-hand position (TwoSets-1)

} *//end of function Code2Lines()*

Table 3.9: Core function *Code2Lines* of the novel Wi2l coder. For **encoding the two lines** k and $k+1$ the algorithm first recursively codes the lines $2k+2$ and $2k$ in step 1). In step 2) the line buffer is loaded with the wavelet coefficients to be coded. In step 3) a loop is started that traverses all units of two sets (denoted by $TwoSets$) in the line buffer. In step 3a) each of these sets is performed separately. The operations on a single set of four coefficients are given as follows. Information that has been computed in step 1) is retrieved in 3a(i.) to compute the m_i of each set of four coefficients in (ii.). Then the relevant q_{Gi} of the previous level is encoded in (iii.A) followed by the four coefficients in (iii.B). In step 3b) it is detected if the two lines are the *first* or the *second* two lines. For the first two lines level information of 8 coefficients is written to the buffer. For the second two lines this level information is retrieved to compute (ii.) and store (iv.) q_G for 16 coefficients. The relevant levels $m_i, i = 0..3$ are encoded using q_G (iii.).

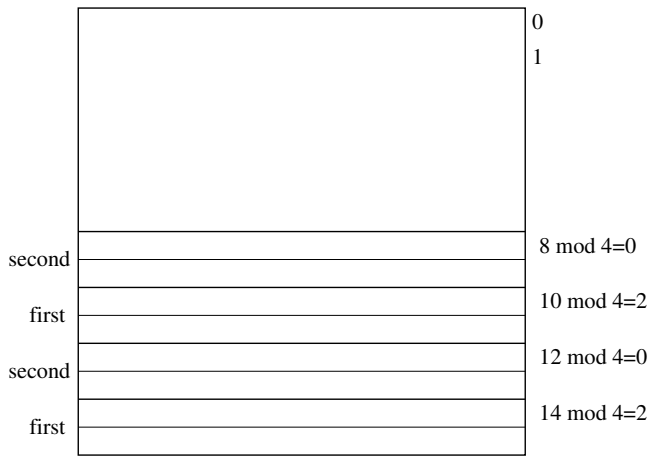


Figure 3.27: The function *Code2Lines()* has to decide in 3.(b) whether it codes the first or the second two lines. The lines within a wavelet subband of level i are numbered from $0, 1, \ldots, N/2^i - 1$, where N is the picture dimension. A modulo operation can thus perform the distinction. When for instance $k = 2, 6, 10, 14$, k modulo 4 equals 2, and the first two lines are coded. If $k = 0, 4, 8, 12$, k modulo 4 equals 0, and the second two lines are coded.

or HH, a wavelet level, and the line index k, which results into coding blocks of four coefficients within the lines k and $k+1$. The steps in the encoding procedure are given as follows.

1. In step 1) the four child lines of lines k and $k+1$ are coded. This makes sure that the q_G levels of all child coefficients of lines k and $k+1$ are computed. One of these MQL levels refers to 16 child coefficients.

2. In step 2) the two lines of coefficients to be coded are read into the line buffer. There only exists one line buffer for all function calls. Even if there exist recursive calls, the buffer is relieved when the function returns.

3. In step 3) the main loop for the function is started. This loop goes through all sets of 8 adjacent coefficients in the two lines from right to left. Such a set is denoted by the variable $TwoSets$.

3.(a) In 3a) this set is divided into two subsets $set(i = 0, 1)$ each of them having 4 coefficients. For each $set(i)$ the function goes through three steps: In **i)** q_{Gi} is retrieved from the buffer and used in **ii)** together with the coefficient quantization levels $q_j, j \in set(i)$, to compute the level m_i. In **iii)** q_{Gi} and $set(i)$ of coefficients are coded. More specifically, coding a set i of coefficients is interluded by coding the corresponding q_{Gi} of the next lower level.

3.(b) In 3b) the function detects if the first two or the second two lines are coded. This can be realized by a modulo operation as illustrated in figure 3.27. Actions for each of these situations are different, as a set of eight coefficients denoted by $TwoSets$ in two lines forms a unit with the eight corresponding coefficients in the next two lines. Such a unit is coded as a whole in Bcwt and Spiht, but is broken up here into several parts that are intermitted by parts of other units and quantization levels. If the function concerns the first two lines, then the two quantization

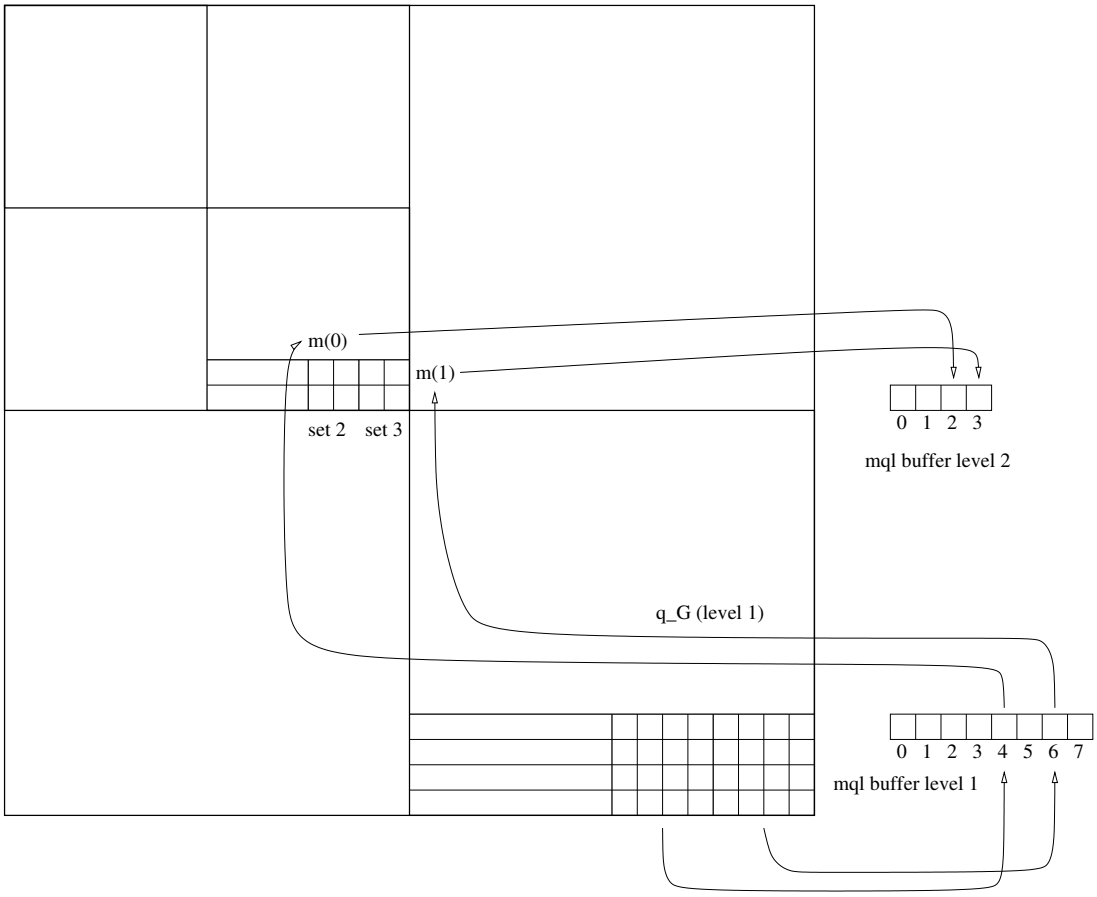


Figure 3.28: Maximum quantization level (MQL) buffer operations for encoding two sets (eight coefficients) of the **first two lines** from a unit of four lines within a subband with Wi2l. Information of the next lower level MQL buffer is retrieved to compute the current two MQL buffer quantization levels. The dimension of the MQL buffer for a level is half the dimension of the relevant subband. The number of coefficient sets equals the size of the MQL buffer. To access q_G of the $(level - 1)$ subband, the current set index has to be multiplied by 2 to get the appropriate buffer index. Set 3 for instance requires the MQL buffer element of the next lower wavelet level with index 6.

levels m_0 and m_1 are written to the MQL buffer at the positions $(TwoSets, TwoSets - 1)$ of the current level, see figure 3.28. The MQL buffer then serves to store intermediate results. When the function is called for the next two lines, these two quantization levels will be retrieved in 3b) **i)** and here denoted as m_2, m_3, as it is illustrated in figure 3.29. As illustrated in figure 3.30 the buffer of a wavelet level thus serves as a link to encode the Spiht typical units of 16 coefficients. Importantly, the two parts of such a unit are not coded in succession, as all parts of different units concerning two lines are coded in a series. The levels $m_i, i = 0 \ldots 4$ are used in **ii)** to compute the quantization level q_G. In **iii)** the levels $m_{0,1,2,3}$ are coded and in **iv)** q_G is written to the MQL buffer at position $(TwoSets - 1)$ for the current level.

As can be concluded now, four adjacent lines have to be coded in step 1) to compute the corresponding q_G levels, which are then written to the MQL buffer at position $TwoSets - 1$ to be accessed by function calls at the next higher level. The value in the MQL buffer at position $TwoSets$ is then not needed any more. For completeness the decoding algorithm is listed in table 3.10.

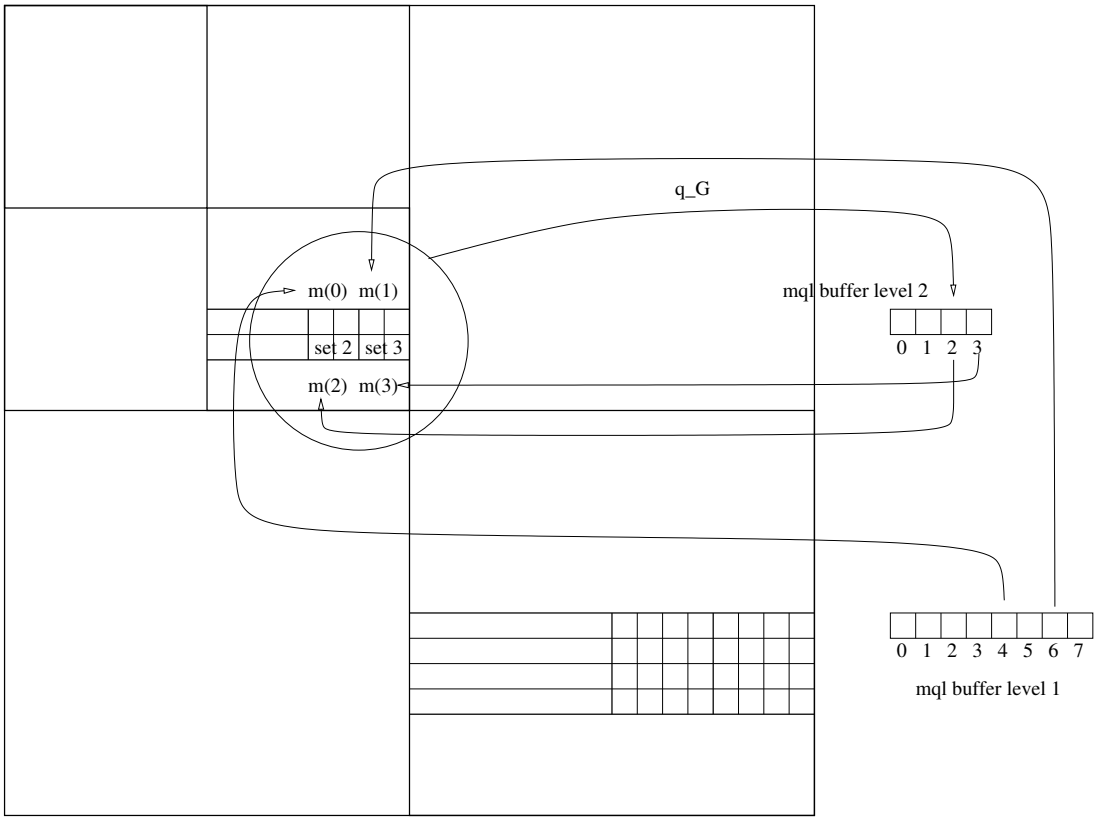


Figure 3.29: Maximum quantization level (MQL) buffer operations for encoding two sets of the **second two lines** from a unit of four lines within a subband with Wi2l. The maximum of the current two MQL values and the two MQL values of the previous two lines, which are accessed through the buffer, gives the q_G of a set of 16 coefficients. This q_G level (concerning four lines) is finally written to a field in the buffer to be accessed by the next level operations.

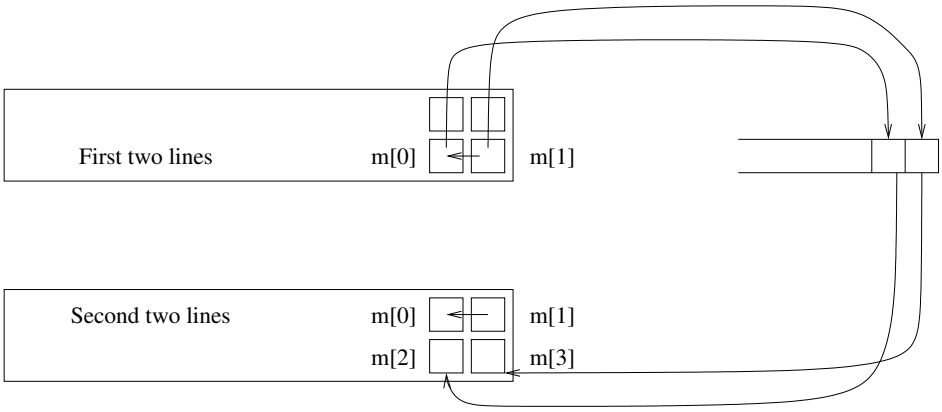


Figure 3.30: MQL buffer as a link to two parts of eight coefficients, which form a unit of 16 coefficients in Spiht. In conjunction with the recursive nature of Wi2l, the algorithm only requires two lines of a wavelet subband in memory. Note that when reading the second two lines, the MQL levels that have previously denoted by m_0 and m_1 are now denoted by m_2 and m_3.

Decode2Lines(PMemoryCard,subband,level,k) {
//decode line (k+1) and line (k):

1. Init coefficients in line buffer with 0
2. DimMql=DimPic/$2^{level+1}$
   for TwoSets=1:2:DimMql-1
   (a) if k/2==even
      i. Get q_G from buffer at position (TwoSets-1)
      ii. If $q_G \geq q_{min}$ {decode m_3, m_2, m_1, m_0 using q_G}
         else init these levels with -1
      iii. Put m_2, m_3 to buffer at position (TwoSets-1) and (TwoSets)
      //if k/2 is odd:
      else Get m_0, m_1 from buffer at position (TwoSets-1) and (TwoSets)
   (b) for i=0:1
      i. If $m_i \geq q_{min}$ decode set i of 4 coefficients using m_i and write to line buffer
      ii. If (level>1)
         A. if $m_i \geq q_{min}$ {decode q_{Gi}(level-1) using m_i}
            else q_{Gi}(level-1)=-1
         B. Put q_{Gi}(level-1,2·(TwoSets+i-1)) to buffer
3. Write lines (k+1) and (k) from line buffer to the memory card
4. If level>1 decode (level-1) lines at (2k) and (2k+2):
   (a) Decode2Lines(PMemoryCard,subband,level-1,(2k))
   (b) Decode2Lines(PMemoryCard,subband,level-1,(2k+2))

} *//end of function Decode2Lines()*

Table 3.10: Counterpart of the recursive encoding function of Wi2l for **decoding two lines** of a wavelet subband. The parameters $subband, level$, and k denote that the lines $k+1$ and k of one of the three wavelet subbands HL, LH, or HH in the given level are to be decoded. The parameter *PMemoryCard* is a pointer to the location of the picture on the flash card. The function reads binary data from the card using a buffer of 512 Bytes and line-wisely writes decoded picture samples to the card using a line buffer of $2N$ Bytes, where N is the picture dimension. The required memory including the buffer for intermediate data (MQL buffer) is the same as for the encoding function. The recursion is now located at the end of the function in step 4) and ensures that the trees are decoded starting from the highest level.

A) ENCODING:
main() {

1. *// Code 3 subbands:*
   list=(**HL**,**LH**,**HH**)
   for i=1:3
   //subband line index starts with 0
   (a) Code2Lines(list(i), level=MaxLevel-2,k=2)
   (b) $q_G(i)$=Code2Lines(list(i), level=MaxLevel-2,k=0)
2. CodeLast2Levels($q_G(i = 1, 2, 3)$)

}

B) DECODING:
main() {

1. $q_G(i = 1, 2, 3)$=DecodeLast2Levels()
2. *// Decode the 3 subbands:*
   list=(**HL**,**LH**,**HH**)
   for i=3:-1:1
   (a) Write $q_G(i)$ to appropriate buffer position
   (b) Decode2Lines(list(i), level=MaxLevel-2,k=0)
   (c) Decode2Lines(list(i), level=MaxLevel-2,k=2)

}

Table 3.11: Main loop of the Wi2l algorithm for A) encoding and B) decoding a picture. Calling the *Code2Lines()* function two times for the four lines in the third highest level ensures that all lower levels of a subband are encoded. Thus this call has to be done for each of the subbands HL,LH, and HH. The function *CodeLastLevels()* encodes the tree level information $q_G(i), i = 1, 2, 3$ of the lower levels and the not yet coded pixels of the two highest levels.

3.7.3.2 Main Loop and Coding of the Last Levels

The main loop for the coding process is illustrated in table 3.11. For each of the three possible wavelet subbands the recursive function *Code2Lines()* has to be called two times, one time for the first two lines in the level $MaxLevel - 2$, and a second time for the second two lines of this level. As illustrated in figure 3.31 the coder then goes recursively through all the levels of the given subband, starting to encode the lines from the lowest level. Calling the core function of Wi2l for the two lines k and k+1 of a subband in the third highest level ensures that all the descendant coefficients in the tree are encoded as well.

The q_G level of the $MaxLevel - 2$ subband is passed to the function *CodeLast2Levels()*. Coding of the last two levels is performed through this separate function, as the last two wavelet levels are given as a 4x4 matrix and the typical q_G levels for sets of 16 coefficients do not exist any more, see figure 3.32. The function is listed in table 3.12 and works as follows.

In **step 1)** the coefficients are read into the line buffer. In **step 2)** the set H of coefficients, which refers to the highest wavelet subbands, is denoted by the position array pos[]. When the first position p[0] is left, the array denotes the set $H_{\setminus LL}$, which is the set H excluding the LL subband. **Step 3)** computes the m_i levels $i = 1, 2, 3$ for the children of the set $H_{\setminus LL}$. In **step 4)** $q_G(0)$ is defined as the maximum of these m_i. In **step 5)** $q_G(i = 1, 2, 3)$ of the $MaxLevel - 2$ subbands and the coefficients with their m_i levels for the $MaxLevel - 1$ subbands are coded. In **step 6)** $m(0)$ is defined as the maximum quantization level of all coefficients. In the **steps 7)** and **8)** $q_G(0)$ and the coefficients of set H are coded. In the last **step 9)** $m(0)$ is coded using a predefined maximum quantization level q_{max}, which is set in the source code in accordance to the binary word length.

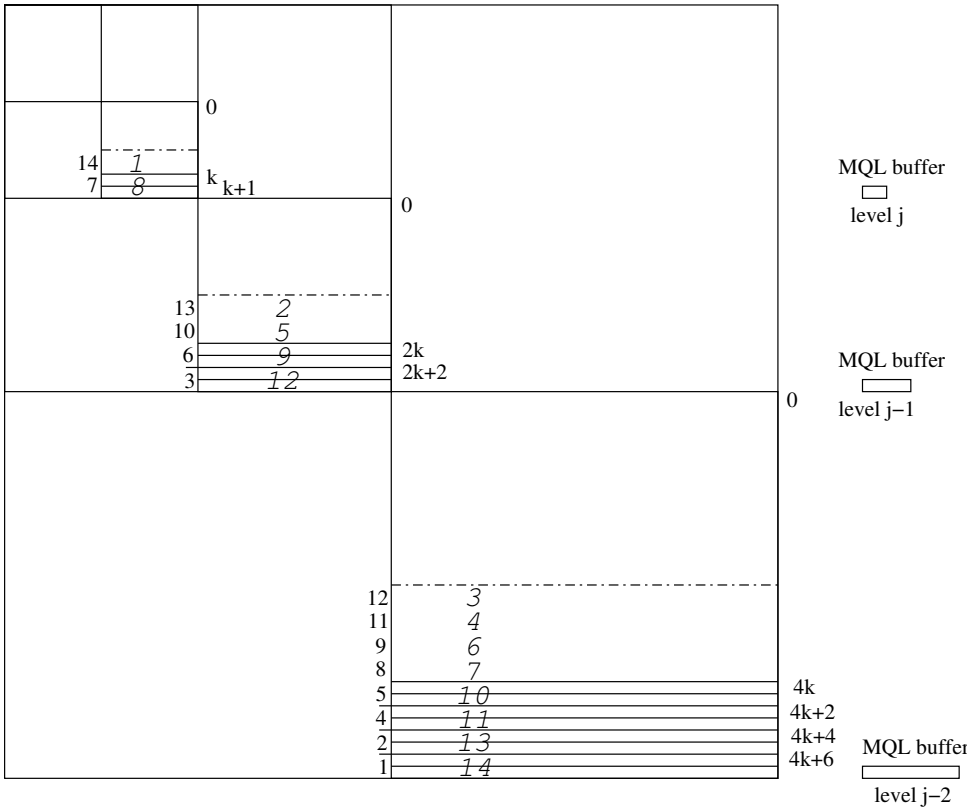


Figure 3.31: Recursion of the Wi2l coder. Calling the function *Code2Lines()* for two lines k and k+1 of a wavelet subband ensures that all the descendant coefficients in the tree are encoded as well. The recursive nature of Wi2l allows to line-wisely encode a complete picture with a minimum of intermediate tree data, which is stored in a small maximum quantization level (MQL) buffer. In the encoding process the tree is coded starting from the lowest wavelet level. The numbers within the lines indicate the order of function calls. Specifically, the call of the function for lines k and k+1 (indicated as call *8*) recursively results into the call of the function for lines 2k and 2k+1 (indicated as call *9*), which similarly results in new recursive calls.

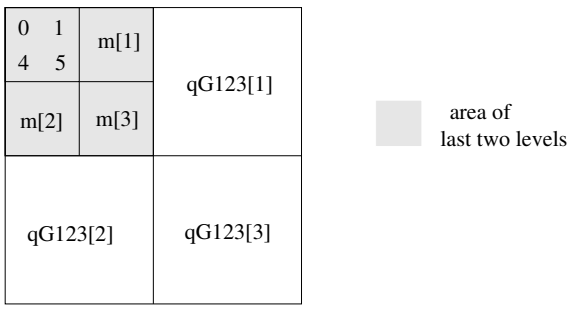


Figure 3.32: Required quantization levels for encoding the last two wavelet levels with Wi2l. The procedure is actually very similar to the corresponding procedure in the reference implementation of Bcwt. $q_G(i), i = 1, 2, 3$ is given to the function as a parameter. This parameter reflects the maximum quantization levels of all the trees with a root in the second highest wavelet level. These quantization levels have to be coded together with the 16 coefficients of the two highest levels. m[1], m[2], and m[3] denote the maximum quantization levels of the coefficients in the second highest wavelet level. In the implementation the last two levels are handled as a 4x4 matrix and loaded row-wisely into the line buffer.

A) ENCODING:
CodeLast2Levels($q_G(i = 1, 2, 3)$) {

1. Read 4x4 matrix of last two levels into input line buffer
2. *//Denote positions of H and $H_{\setminus LL}$:* pos[4]={0,1,4,5}
3. *//Compute $m(i)$:* for i=1:3 $m(i) = \max\left\{q_G(i), \max_{j \in C(pos(i))}\{q_j\}\right\}$
4. $q_G(0) = \max_{i=1,2,3}\{m(i)\}$
5. for i=1:3
   (a) Encode $q_G(i)$ using m(i)
   (b) *//Encode the coefficients of the //MaxLevel-1 HL,LH, and HH //subbands* if $m(i) \geq q_{min}$ encode child coefficients of pos(i) using $m(i)$
   (c) Encode $m(i)$ using $q_G(0)$
6. $m(0) = \max\left\{q_G(0), \max_{i=0...3}\{q(pos(i))\}\right\}$
7. Encode $q_G(0)$ using $m(0)$
8. *//Encode coefficients of H:* for i=0:3 Encode coefficient at pos(i) using $m(0)$
9. Encode $m(0)$ using q_{max}

}

B) DECODING:
DecodeLast2Levels() { *//a 4x4 matrix is //decoded*

1. Decode $m(0)$ using q_{max}
2. pos[4]={0,1,4,5}
3. for i=3:-1:0 Decode coefficient at pos(i) using $m(0)$
4. Decode $q_G(0)$ using m(0)
5. for i=3:-1:1
   (a) Decode $m(i)$ using $q_G(0)$
   (b) if $m(i) \geq q_{min}$ decode child coefficients of pos(i) using $m(i)$
   (c) Decode $q_G(i)$ using $m(i)$
6. Write decoded coefficients from line buffer (4x4 matrix) to destination matrix
7. return $q_G(i = 1, 2, 3)$ to decoder main function

}*//end of DecodeLast2Levels() function*

Table 3.12: Wi2l procedures for A) encoding and B) decoding of the last two levels. As the typical units of 16 coefficients - which are in contrast to Spiht and Bcwt not encoded as a whole anymore but still the subject of the coding principle, do not exist anymore, a standard technique related to Spiht is employed to encode the last two wavelet levels. These levels concern the pixels of the set H and their children. Besides the ordinary quantization levels $q_G(i), i = 1, 2, 3$ and $m(i), i = 1, 2, 3$ that refer to the three trees that span a given picture, the levels $q_G(0)$ and $m(0)$ are introduced to encode the $m(i), i = 1, 2, 3$ and the coefficients of H, respectively. $m(0)$ is the maximum quantization level of the complete picture and is encoded using a fixed level q_{max}, which is defined through the data format of the implementation (here 16 bit).

3.7.4 Performance Evaluation

In this subsection the performance of the Wi2l coder concerning compression and operation times is evaluated. The compression is measured by a PC-implementation in C, while the time measurements are conducted on the low-complexity sensor from section 3.3. For the time measurements a modified version of the PC-code is employed, where the picture data is read and written to a flash memory card.

3.7.4.1 Wi2l Compression Results in Comparison to Spiht

For the quality evaluation of Wi2l a similar methodology is employed than for the verification of the Bcwt algorithm in section 3.6.6. The *GreySet1* test images from the *Waterloo Repertoire* with twelve 256x256x8 greyscale images are reused as a reference. The picture data is transformed with five and six wavelet levels using the fractional wavelet filter introduced in section 3.5, where the first level coefficients are in the Texas Instrument's Q10.5 data format, the second level coefficients in the Q11.4 format, and so on. This filter uses fixed-point arithmetic to transform the picture with only integer number calculations. The wavelet coefficients are stored in the Q15 data format, as justified in section 3.6.6. The picture is coded using an own C-implementation of the Wi2l coder, and then reconstructed the reverse way. The reconstructed picture is compared to the original by computing the peak signal-to-noise ratio (PSNR). The measurements are repeated for the minimum quantization levels $q_{min} = 0 \ldots 8$. The same PSNR computation is done using the Spiht executable programs *fastcode* and *fastdecd* from Amir Said and William A. Pearlman, where the previously achieved bits per byte (bpb) rate from Wi2l for the different q_{min}s were taken as an input parameter. The resulting comparison is illustrated in figure 3.33 a) for wavelet level 6, and in figure b) for level 5.

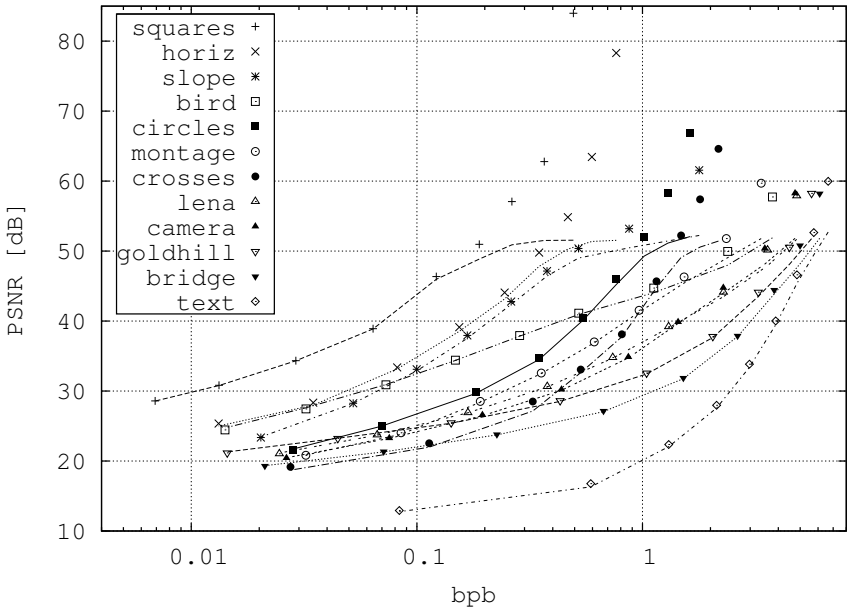
For quantization levels $8, 7, .., 4$, the performance of Spiht and Wi2l is nearly identical. For the levels $3, 2, .., 0$, the PSNR values of Spiht go beyond the values of the Wi2l coder. This is because Wi2l uses a lossy, fixed-point wavelet transform for sensor nodes. For wavelet level 5, qualities up to 53 dB are achieved, while for wavelet level 6 the qualities go up to 50 dB. Wavelet level 6 does not result in a better compression performance as the higher level implicates more fixed-point calculations that imply more errors. Furthermore, as demonstrated in section 3.6.6, a higher level than 5 does not give better performance even with floating-point numbers.

The compression performance of Wi2l is exactly the same as for the Bcwt coder from section 3.6.6. This is due to their related bitstreams: Wi2l sends the same bits than Bcwt (and even the same as Spiht, as Bcwt and Spiht are related as well, see section 3.6) using a different order thus reducing the required memory by a factor of 10. In the next subsection, the compression results of the proposed system are compared to the results obtained with the JPEG techniques.

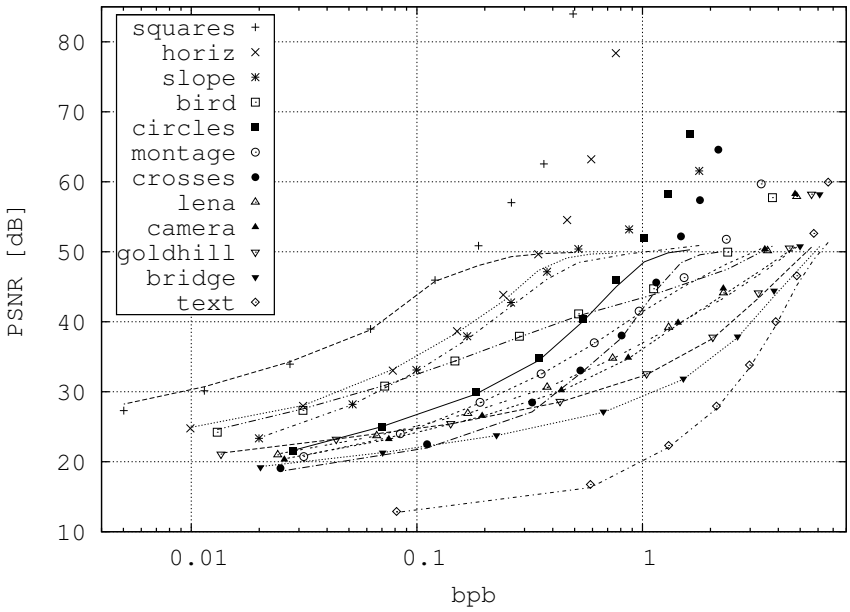
3.7.4.2 Wi2l Compression Results in Comparison to Jpeg / Jpeg2000

In this subsection, the compression performance of Wi2l is compared to the performance of JPEG and JPEG2000. Figure 3.34 shows the results for the images a) circles, b) goldhill, c) lena, and d) text. The results for the eight remaining pictures of the *GreySet1* test images from the *Waterloo Repertoire* are given in the appendix in figure 5.11 on page 157. The curves for Wi2l and Spiht were reproduced from the previous subsection. The results for JPEG and JPEG2000 were obtained using the *jasper* library, see `http://www.ece.uvic.ca/~mdadams/jasper/`.

As reported in the previous subsection, the compression performance for Wi2l declines for



a) level 5



b) level 6

Figure 3.33: Compression performance of the Wi2l coder compared to the performance of Spiht for maximum wavelet level 5 in figure a) and level 6 in figure b). The charts indicate the PSNR values for the quantization levels $q_{min} = 8, 7, .., 0$, where the dots refer to Spiht and the lines to Wi2l. For quantization levels $0, 1, .., 6$, the achieved picture quality is nearly the same. As Wi2l uses a lossy, fixed-point wavelet transform, Spiht goes beyond Wi2l from roughly 53 dB for wavelet level 5 (figure a)), and from roughly 50 dB for wavelet level 6 (figure b)).

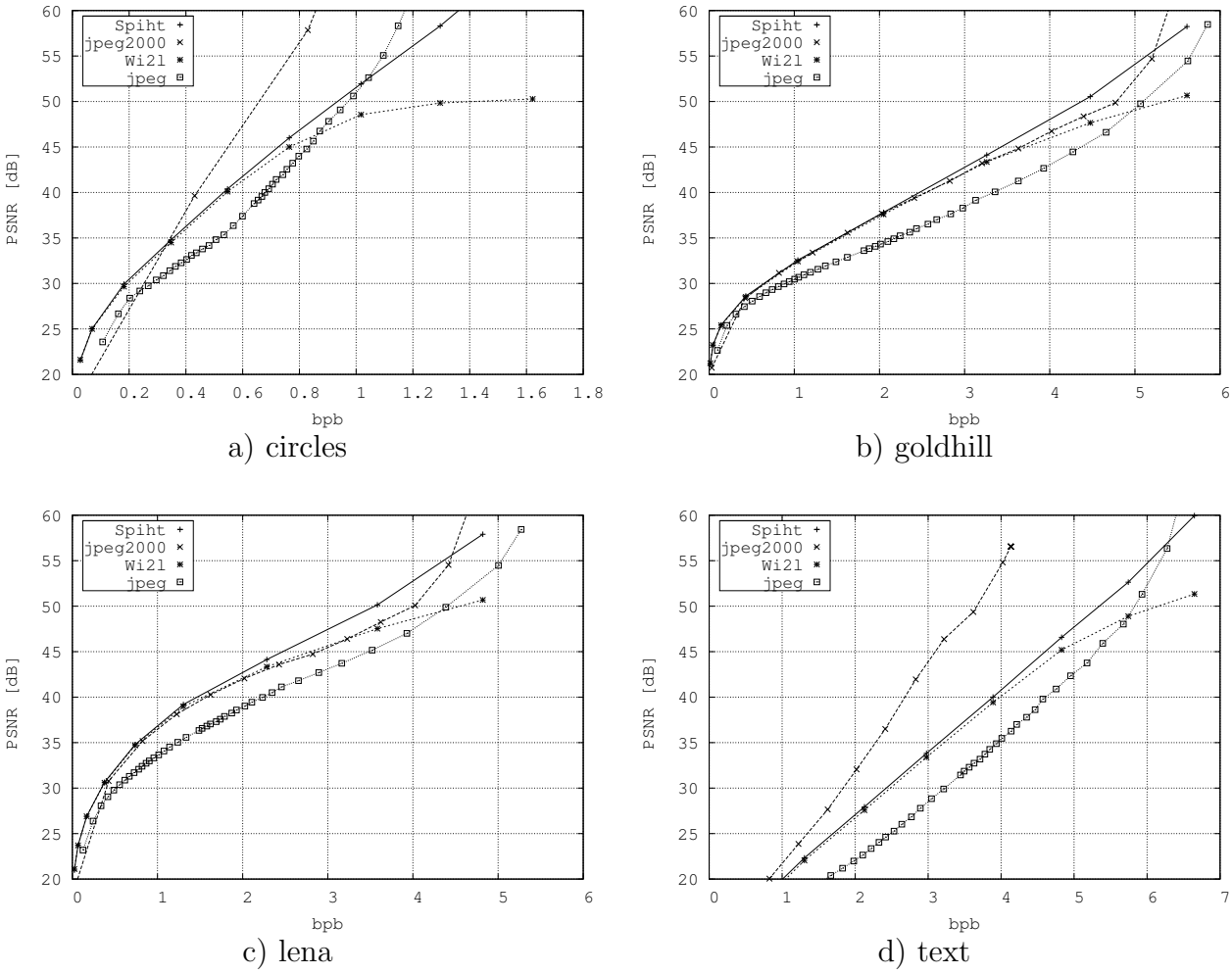


Figure 3.34: Compression performance of the Wi2l coder (which is combined with the proposed fractional filter) and Spiht in comparison to JPEG and JPEG2000. For high compression rates, the proposed system clearly outperforms the JPEG technique by roughly 5 dB. For the natural images in b) and c), Wi2l performs similar than JPEG2000. For the artificial images in a) and d), however, JPEG2000 gives much better results than Wi2l and Spiht, because it is designed for a wider range of image types.

image qualities higher than 45 dB due to the employed fractional filter. For the lower image qualities, Wi2l clearly outperforms the JPEG technique. For the natural images goldhill and lena, Wi2l is very competitive to JPEG2000. For the artificial images circles and text, however, JPEG2000 gives much better results than Wi2l and Spiht. As it is reported in [85], JPEG2000 is more robust against different types of images than the Spiht related techniques.

3.7.4.3 Wi2l Coding Time on Sensor

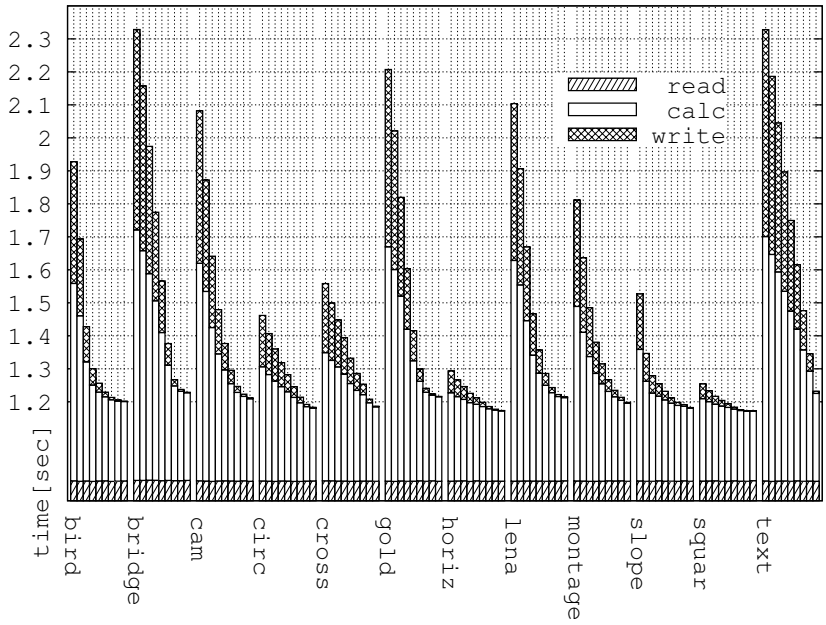
For the coding time measurements the real wireless sensor platform from section 3.3 was employed with the camera and flash memory extension described in subsection 3.5.1. The PC-code of the Wi2l coder was modified in that a MMC-card is used for data exchange. The data can be wavelet coefficients or the compressed binary data stream, for which a write buffer of 512 bytes was allocated. On the flash card each wavelet level is stored on a separate location. Thus it is possible to access two wavelet subband lines by one read operation.

Figure 3.35 shows the results for the a) coding and b) decoding times. The results are categorized by access times (read and write) and operational times (calc). For each picture, q_{min} was varied from $8 \ldots 0$. The lower the quantization levels, the longer the coding time. For the encoder the read access times stay almost constant at 0.96 seconds and take the largest proportion, as each coefficient has to be read. For $q_{min} = 4$ most pictures give coding times shorter than 1.3 seconds. For the decoder the write access takes 9.3 to 9.5 seconds, thus total decoding time is approximatively 10 seconds. This is because the decoder writes two lines, i.e., a block of 512 bytes at different locations. As detailed in [5], writing single and not consecutive blocks on the MMC card is very slow in contrast to the read operation for a block.

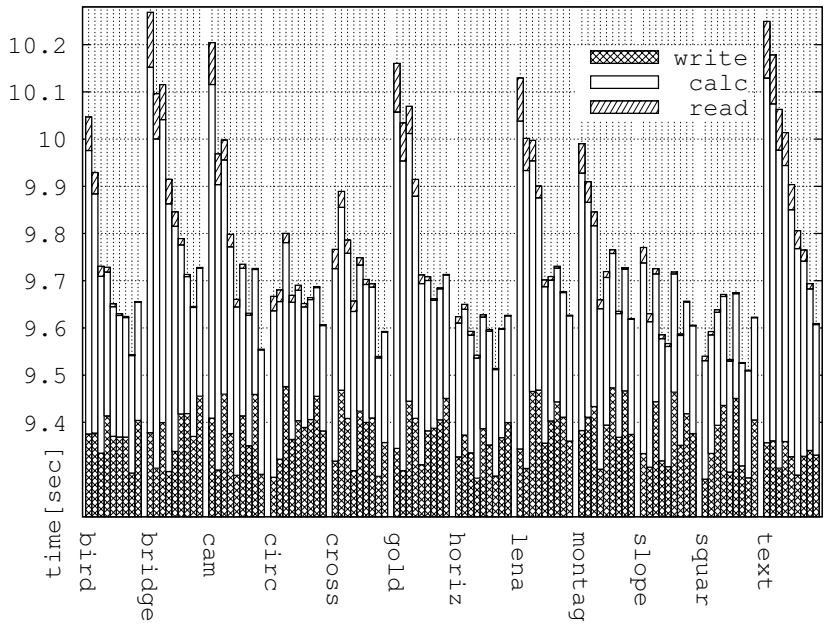
3.7.5 Summary Wi2l Coder

In this section, the Wi2l coder for wavelet image compression was introduced. Wi2l is designed to fulfill the constraints of low-complexity sensor nodes with typical microcontrollers, which in the past were considered to be not sufficient for image processing. The typical hardware for wavelet transforms and image compression employs field programmable gate arrays (FPGA) or digital signal processors (DSP).

The Wi2l coder reads two lines of a wavelet subband and codes the image backwards. For the image transform, the in section 3.5 introduced fixed-point wavelet filter is employed so that the complete system only performs integer calculations. As the coder works recursively, there only exists a small buffer for each wavelet level where intermediate results for the two lines are stored. Thus the memory requirements are extremely low: As it is demonstrated, a 256x256x8 bit monochrome picture can be compressed using the dsPIC30F4013, a low-cost controller with a total of 2 kByte RAM. As Wi2l is a backward version of Spiht, it achieves the same compression rates. Due to the employed lossy wavelet transform, the system is not suitable for lossless image compression. (Note that Wi2l of course can also be employed with a floating-point transform and then can give lossless compression.) However, the loss of picture quality is barely visible for PSNR values higher than 40 dB, and lossless compression may be a minor issue regarding the application of wireless camera sensor nodes, which typically have very low data transfer rates. Similarly, the long decoding times may be of secondary interest, as the pictures may be decoded outside of the sensor network.



a) Coding



b) Decoding

Figure 3.35: Wi2l coding (figure a)) and decoding (figure b)) times for the 256x256x8 pictures from the reference set on a real controller (dsPIC4013) with 29 MIPS. For each picture the coding time is given by nine bars for $\mathbf{q}_{min}$=8..0, where the bars are categorized by access times (read and write) and operational times (calc). The higher quantization level is located at the left-hand side. Times for decoding are much less stable as decompressed picture lines have to be written on the flash card, which gives lower performance for random write access [5]. For a camera sensor network the Wi2l encoder is of more relevance as the pictures may be decoded outside of the network.

3.8 Conclusion Picture Compression

The task to be solved in chapter 3 was to develop appropriate algorithms that allow a wavelet-based picture compression on a typical wireless sensor node. Wavelet techniques are generally regarded to give the best performance in this field, especially for very high compression rates - a feature that may be essential for wireless sensors. Due to the required low costs, the small size, and the battery, these systems are very limited. Conditions that go along with these limitations concern RAM, data rates, execution speed, word-length, and floating-point numbers.

The wavelet picture compression is two-fold - first the transform of the picture has to be computed. This results in a picture of the same dimension, however, due to the real number format the transformed picture needs more memory. (Integer wavelet transforms were not considered as they do not give the superior performance.) Second a coding algorithm has to exploit the given patterns in the transform.

The literature review revealed that work has been devoted to reduce the complexity and requirements of the techniques. Concerning the wavelet transform, line-based systems were introduced that allow to compute the transform with reduced memory requirements. Concerning the coding techniques, a backward version of the Spiht coder - a very competitive yet simple algorithm, was recently proposed. The current techniques do however not yet fulfill the low memory requirement of typical sensor nodes. These nodes need to run a set of programs and algorithms only for the wireless communication and have little resources for data processing. The in the literature given low-memory solutions would require more than 10 kBytes for the transform or rather more than 20 kBytes of memory including the coding. They thus can rather be applied to FPGAs or DSPs, which are generally not employed for the sensors. Therefore the work on a novel low-complexity compression system was investigated.

A first investigation of this part of the work concerned the design of a sensor node including the hardware for wireless connectivity and the software for some basic wireless measurements. The hardware was selected to allow for the verification of possible novel algorithms. The measurements especially gave an intuition on the difficulties and required efforts in algorithm design on such systems. An external flash memory library was considered to be necessary to store and compress pictures. This library was developed in [5]. From then on the investigation on the picture algorithms was followed.

For the wavelet transform, a low-memory version of the line-based transform was developed. The so-called fractional filter solves two problems - first it is a fixed-point filter that uses 16 bit integer numbers to compute the real values of the wavelet coefficients, and second it employs a novel computation scheme that computes fractions of the wavelet coefficients that are updated in many passes. Thus less than 1.2 kByte are needed for the transform. The scheme just needs memory for one picture input line and two picture destination lines. The C-code of the filter was first tested on a PC and later verified on the wireless sensor. Execution speeds are highly dependent to the external flash memory of the sensor.

For the development of a coding technique, the backwards coding system Bcwt was selected as a starting point. Bcwt is a backward version of Spiht and does only need a small subpart of the original picture to be in memory. To verify the technique and to gather some insights on possible refinements, an own implementation was provided and evaluated for a given set of reference pictures.

The finding concerned the development of a novel algorithm for wavelet based image coding. The wavelet image two-line (Wi2l) coder in contrast to previous algorithms only reads two lines of a wavelet subband and immediately processes them. It breaks the typical units of 16

coefficients of Spiht into subunits that are processed in a stream. The recursion of Wi2l allows to reduce the amount of intermediate data to a minimum. Thus a picture of 256x256 8 bit pixels can be compressed with only 1.5 kByte of memory.

The Wi2l coder is evaluated to give almost the same compression performance than Spiht for high compression rates. If lossless compression with Wi2l is required, a different transform than the fractional filter has to be selected. To measure the operation times for compression and decompression on the sensor the flash card is employed for data input and output. Encoding a picture approximatively takes two seconds, while decoding can take roughly ten seconds. The slow decoding times are generated by the long access times of the flash card for writing lines at not related locations.

Future work on the fractional filter and Wi2l may concern the reduction of operational times. This may be achieved by the improvement of the writing access times of the MMC card or by a refinement of the algorithms to especially serve the needs of the card, e.g., reduce or reorder the write access. For the Wi2l coder, a progressive feature may be introduced, similarly as it was done for Bcwt in [90].

Chapter 4

Final Conclusion

The presented thesis gives novelties in the field of low-complexity text and image compression. The first part (chapter 2) concerns the introduction of a scheme for short message compression on mobile devices. Standard text compression techniques (e.g. the *zip* compression) can compress large files on personal computers (PC) and fail for short text.

To design a suitable compression scheme the technique *prediction by partial matching* (PPM) was selected as a starting point. An own PPM software was developed with the specific feature that a hash table with a smart hash function is employed to represent a data tree. Such tables with collision resolving techniques can be used instead of the more complicated and computationally expensive trees. The hashing technique was selected as it appeared to allow for a low-memory version of a statistical model for PPM. The software features functions to preload and to inspect the statistical data.

The results of the conducted statistical evaluation with the own software tool gave insights into the real memory requirements and the compression potential of PPM with limited memory configurations. More specifically, it was verified that preloaded statistical data gives reasonable compression while a novel idea to tremendously reduce the memory requirements of the hash table was developed: In the past, statistical context models contained a string for each of their entries. This can be compared with a library, where each book has a title that allows to check if the right book is found. The titles can require several MBytes of memory. The novel approach instead only stores a single parity byte for each title, which serves to verify if the searched book was found. It is a further compression technique inside of the PPM compression algorithm. The cost of the novel technique is that the possible amount of statistical data is limited and that a specific search function to check the context of an entry has to be employed.

The so constructed low-complexity model is scalable in memory and computational requirements by the selection of hash table sizes and model orders, respectively. A model of 32 kByte already cuts a short text sequence of 100 bytes in half using the PPM order 2. A 256 kByte model using order 3 gives a compression of 3.1 bits per byte (bpb). For these results training data from the book *Principles of Computer Speech* (600 kByte) by I.H. Witten was selected to compress short sequences of the text *Programming by Example revisited* (13 kByte) by J.G. Cleary. To our knowledge the current commercial software *smsZipper* employs the presented scheme similarly using text books to gather training data. Reported compression gains are in the range of 4 to 2.7 bpb. A requirement for the system is that typical training data has to be gathered. This requirement is very general and will hold true for each related future technique, as the compression has to start from the first few bytes and statistical data thus has to be preloaded. The conducted measurements show that the proposed low-complexity model

retains the statistically meaningful data of a training file. This means that improvements can especially be achieved by the selection of more relevant training data, e.g., a corpus of real text messages.

Benefits of the introduced low-complexity compression of short messages may concern reduced costs and battery savings for the users, as it was demonstrated in section 2.7. For the providers it may be be interesting that the cellular network is relieved thus allowing to deliver peak amounts of messages in time, as it is required for example for emergency alert services.

Future improvements of the presented text compression technique may address the further reduction of required memory, so that the compression can even be applied to more limited systems like sensor networks. Whereas a cell phone can easily allocate 32 or 64 kByte of RAM, a small wireless sensor may only supply a few kBytes for a compression algorithm. Lossless compression in sensor networks may concern signaling data or sequences of measurement data.

As the theme of this thesis was defined to cover the area of lossless and lossy compression, the future work on text was followed by supervised student work [50], while the wavelet-based low-complexity compression of images was addressed in chapter 3. To be able to verify novel algorithms and to acquire some insights on sensor networks, an own wireless sensor was designed as a first step. The primarily measurements with the own system gave some insights on the typical limitations of such microcontroller driven systems. As a result the amount of 2 kBytes of RAM was selected as a limit while the algorithm should only perform integer calculations. Another requirement concerns a low algorithmic complexity, as pictures should be sent on request with little delay and the battery should not be affected by time-consuming computations. The sensor filesystem from [5] for flash memory was selected to allow for storage of pictures, including original, intermediate, and transformed picture data. External flash memory will be a standard future component of wireless sensors [86].

The Daubechies 9/7 wavelet gives state-of-the-art image compression especially for high compression. High compression is essential in sensor networks due to the limited bandwidth. The task to be solved was thus to design a 9/7 wavelet based picture compression algorithm that requires less than 2 kBytes of RAM to compress a greyscale 256x256 picture only using 16 bit integer calculations. Further the data can only be accessed in blocks of 512 bytes from an external multimedia card (MMC). These requirements seem to be a *squaring the circle* problem according to the current literature, where wavelet transform and coding are generally performed on more advanced platforms like FPGAs or DSPs. The microcontrollers of the low-cost wireless sensors are widely estimated to be not sufficient for such advanced signal processing tasks. The most related works concern a line-based version of the transform and a backwards version of the *set partitioning in hierarchical trees* (Spiht) algorithm, each of them allocating more than 10 and 20 kByte of RAM, respectively.

As a first step the standard computational schemes for the picture wavelet transform were studied. The memory problem for the wavelet transform can be illustrated more specifically regarding the 9/7 wavelet filter. This filter has to be slided over the picture in vertical and horizontal direction. To compute one transformed picture line a block of 256x9=2304 wavelet coefficients at least has to be kept in memory. If only 16 bits are allocated for each coefficient, memory requirements would be 2304x2=4609 bytes.

As the novel scheme shall only use integer calculations and the wavelet coefficients should not allocate more than 2 bytes, the fixed-point technique was considered to be essential, as it allows to compute real number operations with integers at the cost of more programming work and less precision.

As a result of the studies the novel *fractional wavelet filter* is introduced which combines

the fixed-point arithmetic with a specific computational scheme that only computes fractions of a series of wavelet coefficients thus allowing the transform of a 256x256 with less than 1.5 kByte. Even if the transform requires several seconds of computing time on the own sensor node hardware, it can be applied to any typical camera sensor node where the data can be accessed line-wisely from a flash memory card. The novel filter may encourage future work in the emerging field of low-memory wavelet transform.

The fixed-point transform is not lossless, but the loss of precision is not of interest if a lossy coding system is employed. Similarly than for the text compression algorithm, the investigations for image coding concern a reference implementation of a related technique, which is the *backwards coding wavelet trees* (Bcwt) algorithm. This recent scheme backwards encodes a transformed picture allocating 20 kBytes of memory and achieves tremendous memory savings over the standard wavelet coder Spiht, which can require several MBytes of RAM. Through the implementation work the idea to encode a picture recursively using two lines of a wavelet subband was developed. Thereby the typical units of 16 coefficients are broken up. The novel scheme is thus called *wavelet image two-line coder* (Wi2l) and only needs a line input buffer of 512 bytes, a binary buffer for compressed data of 512 bytes, and a maximum quantization level (MQL) buffer of 126 bytes that stores intermediate wavelet tree data for two lines of each wavelet subband. Total memory requirements therefore are smaller than 1.5 kByte.

The coder is evaluated on the own sensor hardware to compress a picture in approximatively 2 seconds. Decompression is in the range of 10 seconds, as the employed library for flash memory access is very slow for write access. As the pictures are generally decoded outside of the camera sensor network, the Wi2l coder can be regarded as a useful and readily applicable addendum to current wireless sensors. The achieved compression rates are the same as with the very competitive Spiht algorithm, with the difference that Spiht is intended for PCs and has a progressive feature while Wi2l runs on a small microchip.

Together with the fractional filter a complete system for sensor image compression is provided that can run on a low-complexity 16 bit microcontroller allocating less than 1.5 kByte of RAM while only integer calculations are performed. While the requirements of the introduced system are extremely low, compression performance is state-of-the-art (competitive to JPEG2000 and outperforming JPEG). The system can save expensive and timely limited bandwidth in satellite communication or be an update to the controller of a tiny camera sensor. Future work on the fractional filter may reduce the computational complexity by incorporating the lifting scheme and for the Wi2l coder concern the integration of a progressive feature.

So the theme of the thesis - that is, to include lossless and lossy compression techniques for limited and wireless systems, could be fulfilled. The lossless text compression found its way to the general user through the *smsZipper*. The picture compression system was verified on proper hardware but is not yet applied in pico-satellite communication systems as proposed in this thesis. But this does not mean anything: The compression technique of subtractive notation, for instance, was occasionally used by the Romans, widely applied not until the middle age, and is in use till this day.

Bibliography

[1] P. Traynor, "Characterizing the limitations of third-party EAS over cellular text message services," Georgia Institute of Technology, Tech. Rep., Sept. 2008.

[2] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," in *IEEE Transactions on Information Theory*, vol. 23, May 1977, pp. 337 – 343.

[3] K. Schilling, "Small is beautiful: Die UWE-Satelliten im 1 kg-Bereich," 39. Jahrestagung des IFR, July 2008.

[4] H. Karl and A. Willig, *Protocols and Architectures for Wireless Sensor Networks.* John Wiley & Sons, 2005.

[5] S. Lehmann, S. Rein, and C. Gühmann, "External flash filesystem for sensor nodes with sparse resources," in *Proceedings of the ACM Mobile Multimedia Communications Conference (Mobimedia '08)*, July 2008.

[6] L. Ye, J. Guo, B. Nutter, and S. Mitra, "Memory-efficient image codec using line-based backward coding of wavelet trees," in *Proceedings of the IEEE Data Compression Conference (DCC'07)*, 2007.

[7] J. Cleary, W. Teahan, and I. Witten, "Unbounded length contexts for PPM," in *Proceedings of the IEEE Data Compression Conference (DCC'95)*, March 1995, pp. 52–61.

[8] M. Drinic, D. Kirovski, and M. Potkonjak, "PPM model cleaning," in *Proceedings of the IEEE Data Compression Conference (DCC'95)*, March 1995, pp. 52–61.

[9] P. Skibinski and S. Grabowski, "Variable-length contexts for PPM," in *Proceedings of the IEEE Data Compression Conference (DCC'04)*, March 2004, pp. 409–418.

[10] D. Lelewer and D. Hirschberg, "Streamlining context models for data compression," in *Proceedings of the IEEE Data Compression Conference (DCC'91)*, Snowbird UT, 1991, pp. 313–322.

[11] C. Feregrino, "High performance PPMC compression algorithm," in *Proceedings of the IEEE Fourth Mexican International Conference on Computer Science (ENC'03)*, Sept. 2003, pp. 135–142.

[12] D. Shkarin, "PPM: one step to practicality," in *Proceedings of the IEEE Data Compression Conference (DCC'02)*, April 2002, pp. 202–211.

[13] E. Hatton, "SAMC–efficient semi-adaptive data compression," in *Proceedings of the IBM 1995 conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, Ontario, Canada, Nov. 1995, p. 29.

[14] G. Korodi, J. Rissanen, and I. Tabus, "Lossless data compression using optimal tree machines," in *Proceedings of the IEEE Data Compression Conference (DCC'05)*, March 2005, pp. 348–357.

[15] J. Lansky and M. Zemlicka, "Compression of small text files using syllables," in *Proceedings of the IEEE Data Compression Conference (DCC'06)*, March 2006, p. 458.

[16] N. Kimura and S. Latifi, "A survey on data compression in wireless sensor networks," in *Proceedings of the IEEE International Conference on Information Technology: Coding and Computing (ITCC'05)*, April 2005, pp. 8–13.

[17] H. Gupta, V. Navda, S. Das, and V. Chowdhary, "Efficient gathering of correlated data in sensor networks," in *Proceedings of the 6th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc'05)*, May 2005, pp. 402–413.

[18] N. Gehrig and P. Dragotti, "Distributed sampling and compression of scenes with finite rate of innovation in camera sensor networks," in *Proceedings of the IEEE Data Compression Conference (DCC'06)*, March 2006, pp. 83–92.

[19] J. Rissanen and G. Langdon, "Arithmetic coding," in *IBM Journal of Research and Development*, vol. 23, no. 2, March 1979, pp. 149–162.

[20] S. Rein and C. Gühmann, "Arithmetic coding – a short tutorial and free source code," Technische Universität Berlin, Wavelet Application Group, Tech. Rep., April 2005.

[21] T. Bell, J. Cleary, and I. Witten, *Text Compression.* Prentice Hall, 1990.

[22] D. Salomon, *Data Compression*, 2nd ed. New York: Springer Verlag, 2000.

[23] M. Nelson, "Arithmetic coding + statistical modeling=data compression –part1: Arithmetic coding," in *Dr. Dobb's Journal*, Feb. 1991.

[24] P. Howard and J. Vitter, "Arithmetic coding for data compression," in *IEEE Visual Communication Research*, vol. 82, Holmodel, NJ, June 1994, pp. 857–865.

[25] I. Witten, R. Neal, and J. Cleary, "Arithmetic coding for data compression," in *Communications of the ACM*, vol. 6, no. 30, 1987, pp. 519–540.

[26] A. Campos, "Arithmetic coding," Tech. Rep., July 1999. [Online]. Available: http://www.arturocampos.com

[27] E. Bodden, M. Clasen, and J. Kneis, "Arithmetic coding in a nutshell," in *Proseminar Datenkompression 2001.* RWTH Aachen University, 2002. [Online]. Available: http://www.bodden.de/publications

[28] A. Campos, "Range coder," Barcelona, Tech. Rep., Nov. 1999. [Online]. Available: http://www.arturocampos.com

[29] I. Witten and T. Bell, "Calgary text compression corpus," 1987. [Online]. Available: ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus

[30] T. Bell, I. Witten, and J. Cleary, "Modeling for text compression," in *ACM Computing Surveys (CSUR)*, vol. 21, no. 4, Dec. 1989, pp. 557–591.

[31] J. Cleary and I. Witten, "Data compression using adaptive coding and partial string matching," in *IEEE Transactions on Communications*, vol. COM-32, no. 4, April 1984.

[32] J. Rissanen and G. Langdon, "Universal modeling and coding," in *IEEE Transactions on Information Theory*, ser. 1, vol. 27, Jan. 1981, pp. 12–23.

[33] A. Moffat, "Implementing the PPM data compression scheme," in *IEEE Transactions on Communications, Transactions Letters*, vol. 38, no. 11, Nov. 1990, pp. 12–23.

[34] ——, "Linear time adaptive arithmetic coding," in *IEEE Transactions on Information Theory*, ser. 2, vol. 36, March 1990, pp. 401–406.

[35] A. Campos, "Finite context modeling," Barcelona, Tech. Rep., Jan. 2000. [Online]. Available: http://www.arturocampos.com

[36] ——, "Implementing PPMC with hash tables," Barcelona, Tech. Rep., March 2000. [Online]. Available: http://www.arturocampos.com

[37] M. Nelson, "Arithmetic coding + statistical modeling=data compression –part 2: Statistical modeling," in *Dr. Dobb's Journal*, Feb. 1991.

[38] S. Rein and C. Gühmann, "A free library for context modeling with hash-functions– part I," Technische Universität Berlin, Wavelet Application Group, Tech. Rep., May 2005.

[39] W. Teahan and J. Cleary, "Modeling of English text," in *Proceedings of the IEEE Data Compression Conference (DCC'97)*, March 1997, pp. 12–21.

[40] J. Storer, *Data Structures and Algorithms.* Birkhäuser/Springer, 2002.

[41] B. Jenkin, "Algorithm alley," in *Dr. Dobb's Journal*, Sept. 1997. [Online]. Available: http://burtleburtle.net/bob

[42] S. Rein and C. Gühmann, "A free library for context modeling with hash-functions– part II: Extensions," Technische Universität Berlin, Wavelet Application Group, Tech. Rep., Aug. 2005.

[43] R. Arnold and T. Bell, "A corpus for the evaluation of lossless compression algorithms," in *Proceedings of the IEEE Data Compression Conference (DCC'97)*, 1997. [Online]. Available: http://corpus.canterbury.ac.nz

[44] M. Hart, G. Newby, and M. Perathoner, "Project Gutenberg literary archive foundation," 809 North 1500 West, Salt Lake City, UT 84116. [Online]. Available: http://www.gutenberg.org

[45] S. Rein, C. Gühmann, and F. Fitzek, "Low complexity compression of short messages," in *Proceedings of the IEEE Data Compression Conference (DCC'06)*, March 2006, pp. 123–132.

[46] ——, "Compression of short text on embedded systems," in *Journal of Computers (JCP)*, no. 6, Sept. 2006.

[47] S. Rein, F. Fitzek, and C. Gühmann, "Low complexity compression of short messages – International patent application," Patent publication number: WO/2007/104313, International application number: PCT/DK2007/000127, filing date: 14.3.2007, publication date: 20.09.2007.

[48] ——, "Low complexity compression of short messages – United States patent application," U.S. patent application number: 12/293,020 , filing date: 15.09.2008.

[49] F. Fitzek, S. Rein, M. P. G. Perucci, T. Schneider, and C. Gühmann, "Low complex and power efficient text compressor for cellular and sensor networks," in *15th IST Mobile and Wireless Communication Summit*, June 2006.

[50] H. Hammoud, "Verlustfreie Kompression für Handy und Sensor Netzwerk," Master's thesis, Technische Universität Berlin, Chair of Electronic Measurement and Diagnostic Technology, Feb. 2009.

[51] S. Rein, C. Gühmann, and F. Fitzek, *Mobile Phone Programming.* Springer, 2007, ch. Sensor Networks for Distributive Computing, pp. 397–409.

[52] S. Rein, S. Lehmann, and C. Gühmann, "Fractional wavelet filter for camera sensor node with external flash and extremely little RAM," in *Proceedings of the ACM Mobile Multimedia Communications Conference (Mobimedia '08)*, July 2008.

[53] ——, "Wavelet image two-line coder for wireless sensor node with extremely little RAM," in *Proceedings of the IEEE Data Compression Conference (DCC'09)*, Snowbird UT, 2009.

[54] T. Fry and S. Hauck, "Spiht image compression on FPGAs," in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 9, Sept. 2005.

[55] K.-C. Hung, Y.-J. Huang, T.-K. Truong, and C.-M. Wang, "FPGA implementation for 2d discrete wavelet transform," *Electronics Letters*, vol. 34, no. 7, pp. 639–640, April 1998.

[56] J. Chilo and T. Lindblad, "Hardware implementation of 1d wavelet transform on an FPGA for infrasound signal classification," *IEEE Transactions on Nuclear Science*, vol. 55, no. 1, pp. 9–13, Feb. 2008.

[57] S. Ismail, A. Salama, and M. Abu-ElYazeed, "FPGA implementation of an efficient 3D-WT temporal decomposition algorithm for video compression," *IEEE International Symposium on Signal Processing and Information Technology*, pp. 154–159, Dec. 2007.

[58] C. Chrysafis and A. Ortega, "Line-based, reduced memory, wavelet image compression," *IEEE Transactions on Image Processing*, vol. 9, no. 3, pp. 378–389, March 2000.

[59] J. Oliver and M. P. Malumbres, "On the design of fast wavelet transform algorithms with low memory requirements," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 18, no. 2, pp. 237–248, Feb. 2008.

[60] Y. Sun, H. Zhang, and G. Hu, "Real-time implementation of a new low-memory Spiht image coding algorithm using DSP chip," in *IEEE Transactions on Image Processing*, vol. 11, no. 9, Sept. 2002, pp. 1112–1116.

[61] C.-Y. Su and B.-F. Wu, "A low memory zerotree coding for arbitrarily shaped objects," in *IEEE Transactions on Image Processing*, vol. 12, no. 3, March 2003, pp. 271–282.

[62] H. Arora, P. Singh, E. Khan, and F. Ghani, "Memory efficient set partitioning in hierarchical tree (Mesh) for wavelet image compression," in *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '05)*, vol. 2, no. 7, March 2005, pp. 385–388.

[63] M. Sakalli, W. Pearlman, and M. Farshchian, "Spiht algorithms using depth first search algorithm with minimum memory usage," in *Proceedings of the 40th Annual Conference on Information Sciences and Systems*, March 2006, pp. 1158–1163.

[64] J. Lian, K. Wang, and J. Yang, "Listless zerotree image compression algorithm," in *Proceedings of the 8th International Conference on Signal Processing*, vol. 2, 2006.

[65] Y. Singh, "ISpiht-improved Spiht: A simplified and efficient subband coding scheme," in *Proceedings of the International Conference on Computing: Theory and Applications (ICCTA'07)*, March 2007, pp. 468–474.

[66] H. Pan, W. Siu, and N. Law, "Efficient and low-complexity image coding with the lifting scheme and modified Spiht," in *Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN '08)*, July 2008, pp. 1959–1963.

[67] M. Lanuzza, S. Perri, P. Corsonello, and G. Cocorullo, "An efficient wavelet image encoder for FPGA-based designs," in *Proceedings of the IEEE Workshop on Signal Processing Systems Design and Implementation*, Nov. 2005, pp. 652–656.

[68] A. Kumarayapa, X.-F. Zhang, and Y. Zhang, "Simplifying Spiht for more memory efficient onboard machine-vision codec and the parallel processing architecture," in *Proceedings of the International Conference on Machine Learning and Cybernetics*, vol. 3, Aug. 2007, pp. 1482–1486.

[69] L. Chew, L.-M. Ang, and K. Seng, "New virtual Spiht tree structures for very low memory strip-based image compression," in *IEEE Signal Processing Letters*, vol. 15, 2008, pp. 389–392.

[70] J. Guo, S. Mitra, B. Nutter, and T. Karp, "A fast and low complexity image codec based on backward coding of wavelet trees," in *Proceedings of the IEEE Data Compression Conference (DCC'06)*, March 2006.

[71] J. Hill and D. Culler, "Mica: A wireless platform for deeply embedded networks," in *IEEE Micro*, vol. 22(6), 2002, pp. 12–24.

[72] J. H. et al., "System architecture directions for networked sensors," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridgs, MA, 2000.

[73] A. Tanenbaum, Ed., *Computer Networks*. Prentice-Hall, 1996.

[74] A. Woo and D. Culler, "A transmission control scheme for media access in sensor networks," in *Proceedings of the 7. Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2001.

[75] Y. Liu, "Wireless interface for digital signal controller," Studienarbeit, Technische Universität Berlin, Fachgebiet Elektronische Mess- und Diagnosetechnik, Dezember 2005.

[76] S. Lehmann and D. Bindrich, "Capturing and analyzing pictures with a dsPIC," Project work, Technische Universität Berlin, Chair of Electronic Measurement and Diagnostic Technology, Feb. 2007.

[77] I. Daubechies, *Ten Lectures on Wavelets*, eighth printing 2004 ed. Society for Industrial and Applied Mathematics, 1992.

[78] S. Mallat, *A Wavelet Tour of Signal Processing*. Academic Press, 1999.

[79] B. Usevitch, "A tutorial on modern lossy wavelet image compression: Foundations of JPEG2000," in *IEEE Signal Processing Magazine*, Sept. 2001.

[80] S. Rein and M. Reisslein, "Identifying the classical music composition with wavelet dispersion vector and neural nets," in *Elsevier Information Sciences*, vol. 176, June 2006, pp. 1629–1655.

[81] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in C – The Art of Scientific Computing*, 2nd ed. Cambridge University Press, 1994, pp. 591–606.

[82] K. Rao and P. Yip, Eds., *The Transform and Data Compression Handbook*. CRC press, 2001, ch. 6.3.

[83] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," in *Journal of Fourier Analysis and Applications*, vol. 4, no. 3, Mai 1998, pp. 1069–5869.

[84] M. W. Frazier, *An Introduction to Wavelets Through Linear Algebra*. Springer-Verlag New York, 1999.

[85] D. Taubman and M. Marcellin, *JPEG2000- Image compression, fundamentals, standard and practice*. Kluwer Academic Publishers, 2004, ch. 8.5.1, p. 287 and 289 ff.

[86] A. Leventhal, "Flash storage memory," in *Communications of the ACM*, vol. 51, no. 7, July 2008.

[87] S. Rein, "Fixed-point arithmetic in C: A tutorial and an example on wavelet filtering," Technische Universität Berlin, Wavelet Application Group, Tech. Rep., June 2008.

[88] J. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," in *IEEE Transactions on Signal Processing*, vol. 41, no. 12, Dec. 1993.

[89] A. Said and W. Pearlman, "A new, fast, and efficient image codec based on set partitioning in hierarchical trees," in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, no. 3, June 1996.

[90] J. Guo, B. N. S. Mitra, and T. Karp, "Backward coding of wavelet trees with fine-grained bitrate control," in *Journal of Computers*, vol. 1, no. 4, July 2006, pp. 1–7.

List of abbreviations

bpb bits per byte

Bcwt *backward coding of wavelet trees*, backward version of Spiht

CRC cyclic redundancy check

CSMA carrier sense multiple access

DSP digital signal processor

EZW *embedded zerotree wavelet*, wavelet picture coding algorithm

FPGA field programmable gate array

gzip GNU zip software

GNU is a recursive acronym that stands for *GNU's Not Unix*. The GNU Project is a free software, mass collaboration project, announced on September 27, 1983, by Richard Stallman

GSM Global System for Mobile communications, most popular communication standard for mobile phones in the world

ISM industrial, scientific, and medical communication band

JPEG Joint Photographic Experts Group, picture compression standard widely employed on PCs and cameras

JPEG2000 Joint Photographic Experts Group 2000, Wavelet based successor of JPEG

kByte kilobyte= 1024 bytes

LZ77 Lempel Ziv 1977, sliding window compression algorithm published by Abraham Lempel and Jacob Ziv in 1977

MAC medium access control

MByte megabyte= 1024×1024 or 1048576 bytes

MMC multimedia card

MPEG Moving Picture Experts Group, formed by the International Organization for Standardization (ISO) to set standards for audio and video compression and transmission

MQL maximum quantization level, is employed in the Bcwt and the Wi2l algorithm

PC personal computer

PPM *prediction by partial matching*

PSNR peak signal to noise ratio

RAM random access memory

SMS short message service, a communication service standardized in the GSM mobile communication system

Spiht *Set partitioning in hierarchical trees*, refined version of the embedded zerotree wavelet (EZW) picture compression algorithm

SPI serial peripheral interface

UART universal asynchronous receiver transmitter

Wi2l *wavelet image two-line coder*, picture compression coding algorithm introduced in this thesis

zip is a data compression file format, uses a combination of the LZ77 algorithm and Huffman coding

Chapter 5

Appendix

5.1 Numerical Coding Example Spiht

In this section a short numerical example on encoding the pixels illustrated in figure 5.1 is given. The example refers to section 3.6.4 on page 94, where the Spiht algorithm is outlined. The example is retrieved in section 3.6.5.2 on page 103 to demonstrate the binary equality between Spiht and Bcwt. The entries in the LIP and LSP are not discussed in detail in the example, as the operations on them are the same as with EZW. The LIP and the LSP lists are maintained by the first part of the sorting pass and the refinement pass. Note that the numbering in the example is performed in accordance to figure 3.18 on page 97.

First some initializations in the main program are performed before the main loop is started in the last step:

1. Set $T = 32$
2. Put the root node $(0, 3)$ to the list LIP
3. Make sure that the LSP list is empty
4. Put the root node $(0, 3)$ to the LIS list with type D
5. Start the loop with the sorting pass and the refinement pass

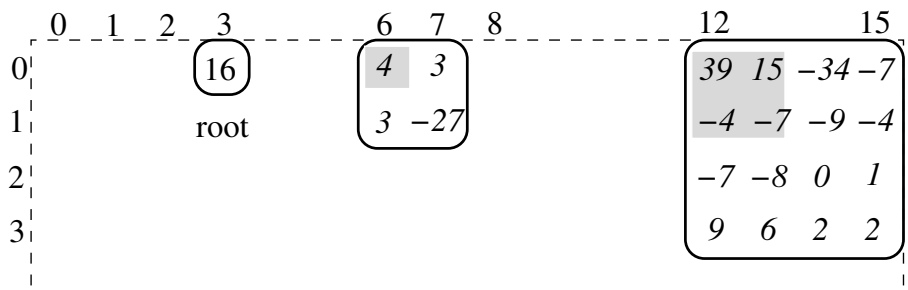


Figure 5.1: Subpart of a transformed picture which is to be encoded by the Spiht algorithm. The root node is put into the list of insignificant sets. The tree is then processed in part 2 of the sorting pass. Pixels of the tree can move to the list of significant pixels or to the list of insignificant pixels. The significant pixels are refined in the refinement pass and remain in this list. The insignificant pixels are checked in part 1 of the sorting pass and can move to the list of significant pixels.

Then the sorting pass and the refinement pass alternate each other (where the refinement pass is not called for $T = T_0$):

1. Check the LIP list:
   (0,3)=16 : Remain in LIP list and output 0

2. (a) i. (0,3)-D denotes a significant tree: Output 1
      ii. A. (0,6)=4 is insignificant: Output 0; move to LIP
         (0,7)=3 is insignificant: Output 0; move to LIP
         (1,6)=3 is insignificant: Output 0; move to LIP
         (1,7)=-27 is insignificant: Output 0; move to LIP
         B. Move (0,3)-D to end of the LIS list as (0,3)-G

   (b) i. (0,3)-G is significant: Output 1
      ii. A. append (0,6)-D to LIS
         append (0,7)-D to LIS
         append (1,6)-D to LIS
         append (1,7)-D to LIS
         B. Remove (0,3)-G from LIS

   (a) i. (0,6)-D is a significant tree: Output 1
      ii. A. (0,12)=39: Output 11 and append to LSP
         (0,13)=15: Output 0 and append to LIP
         (1,12)=-4: Output 0 and append to LIP
         (1,13)=-7: Output 0 and append to LIP
         B. Remove (0,6)-D from LIS

      i. (0,7)-D is a significant tree: Output 1
      ii. A. (0,14)=-34: Output 10 and append to LSP
         (0,15)=-7: Output 0 and append to LIP
         (1,14)=-9: Output 0 and append to LIP
         (1,15)=-4: Output 0 and append to LIP
         B. Remove (0,7)-D from LIS

      i. (1,6)-D is insignificant and remains in LIS: Output 0

      i. (1,7)-D is insignificant and remains in LIS: Output 0

Set $T = T/2 = 16$

1. Check the LIP list: not discussed here

2. (a) i. (1,6)-D is insignificant and remains in LIS: Output 0
      (1,7)-D is insignificant and remains in LIS: Output 0

Call the refinement pass (not discussed here).
Set $T = T/2 = 8$

1. Check the LIP list: not discussed here

2. (a) i. (1,6)-D is significant tree: Output 1

ii. A. (2,12)=-7: Output 0 and append to LIP
(2,13)=-8: Output 10 and append to LSP
(3,12)=9: Output 11 and append to LSP
(3,13)=6: Output 0 and append to LIP

B. Remove (1,6)-D from LIS

i. (1,7)-D is insignificant and remains in LIS: Output 0

Call the refinement pass (not discussed).
Set $T = T/2 = 4$

1. Check the LIP list: not discussed here

2. (a) i. (1,7)-D is insignificant and remains in LIS: Output 0

Call the refinement pass (not discussed here).

In this example the pixels that were appended to the LSP list or to the LIP list were not discussed any more. To exemplify the bitplane encoding procedure the binary output is given for the pixels (0,12), (0,13), (1,12), and (1,13). Note that a follow-up bit is always a sign bit:

	T=32	$T = 16$	$T = 8$	$T_{min} = 4$	quantized value
pixel (0,12)=39	11 $\rightarrow$ LSP	0	0	1	$w_q(0,12) = 36$
pixel (0,13)=15	0 $\rightarrow$ LIP	0	11 $\rightarrow$ LSP	1	$w_q(0,13) = 12$
pixel (1,12)=-4	0 $\rightarrow$ LIP	0	0	10 $\rightarrow$ LSP	$w_q(1,12) = -4$
pixel (1,13)=-7	0 $\rightarrow$ LIP	0	0	10 $\rightarrow$ LSP	$w_q(1,13) = -4$

The first line of the table is interpreted as follows. The pixel (0,12) is checked in the sorting pass in 2.(a) ii.A. for its significance. As it is larger than $T_0 = 32$, a 1 bit is output followed by a sign bit 1. Then the pixel is moved to the LSP list. For each of the next two loops with $T = 16, 8$, it is indicated that the pixel is not refined by a 0 bit. In the last loop for $T = 4$ the 1 bit indicates that the pixel is refined from value 32 to value 32+4=36.

5.2 Text Files

abbrev	source
bib	Bibliographic files (refer format)
book1	Hardy: Far from the madding crowd
book2	Witten: Principles of computer speech
geo	Geophysical data
news	News batch file
obj1	Compiled code for Vax: compilation of progp
obj2	Compiled code for Apple Macintosh: Knowledge support system
paper1	Witten, Neal and Cleary: Arithmetic coding for data compression
paper2	Witten: Computer (in)security
paper3	Witten: In search of "autonomy"
paper4	Cleary: Programming by example revisited
paper5	Cleary: A logical implementation of arithmetic
paper6	Cleary: Compact hash tables using bidirectional linear probing
pic	Picture number 5 from the CCITT Facsimile test files (text + drawings)
progc	C source code: compress version 4.0
progl	Lisp source code: system software
progp	Pascal source code: prediction by partial matching evaluation program
trans	Transcript of a session on a terminal

Table 5.1: Description of the 18 files of the Calgary corpus. This corpus was introduced by Bell et al. and is one of the earliest references for evaluation of compression techniques. It is employed for the verification of the arithmetic coder in section 2.3 and for the own PPM implementation in section 2.4. Additional information on the data files is given in table 5.2.

	lines	words	characters
bib	6280	19274	111261
book1	16622	141274	768771
book2	15634	101221	610856
geo	18	617	102400
news	10059	53939	377109
obj1	87	495	21504
obj2	1213	4600	246814
paper1	1250	8512	53161
paper2	1731	13829	82199
paper3	1100	7219	46526
paper4	294	2166	13286
paper5	320	2099	11954
paper6	1019	6753	38105
pic	0	49	513216
progc	1487	6313	39611
progl	2244	9235	71646
progp	1966	4847	49379
trans	2737	9255	93695

Table 5.2: Length of the data files of the Calgary corpus. In a later section the files are inspected for their statistical information and employed as a data base for static data compression.

file	category	size[byte]
alice29.txt	English text	152089
asyoulik.txt	Shakespeare	125179
cp.html	HTML source	24603
fields.c	C source	11150
grammar.lsp	LISP source	3721
kennedy.xls	Excel Spreadsheet	1029744
lcet10.txt	Technical writing	426754
plrabn12.txt	Poetry	481861
ptt5	CCITT test set (fax)	513216
sum	SPARC Executable	38240
xargs.1	GNU manual page	4227

Table 5.3: Files of the Canterbury corpus. The corpus is an improved version of the Calgary corpus (which is listed in table 5.1). Only the text files alice29, asyoulik, lcet10, and plrabn12 of this corpus are employed for most of the measurements in section 2.5. Similarly, from the Calgary corpus only the text files book1, book2, and paper1-6 were selected. The non-text files are not employed as the task is to develop a low-complexity compression scheme for short text.

file	category	size[byte]
E.coli	Complete genome of the E. Coli bacterium	4638690
bible.txt	The King James version of the bible	4047392
world192.txt	The CIA world fact book	2473400

Table 5.4: Files of the Large corpus. Only the text files bible and world192 were selected for the evaluation. The large files were included for statistical reasons and to find errors in the own implementation.

length [Bytes]	characters
203	Hi Stephan, thanks for the nice plot. I will meet Morten soon and report back how large the compression gain of this email will be. Would be nice if you could report the gain in dependency on the order.
291	The international system of units consists of a set of units together with a set of prefixes. The units of SI can be divided into two subsets. There are the seven base units. Each of these base units are dimensionally independent. From these seven base units several other units are derived.
302	However, few believed that SMS would be used as a means for sending text messages from a mobile user to another. One factor in the takeup of SMS was that operators were slow to eliminate billing fraud which was possible by changing SMSC settings on individual handsets to the SMSC's of other operators.
454	Alice is a fictional character in the books Alice's Adventures in Wonderland and its sequel Through the Looking-Glass, which were written by Charles Dodgson under the pen name Lewis Carroll. The character is based on Alice Liddell, a child friend of Dodgson's. The pictures, however, do not depict Alice Liddell, as the illustrator never met her. She is seen as a logical girl, sometimes being pedantic, especially with Humpty Dumpty in the second book.

Table 5.5: Short messages employed for the evaluation of the smsZipper in section 2.7.

5.3 Additional Figures

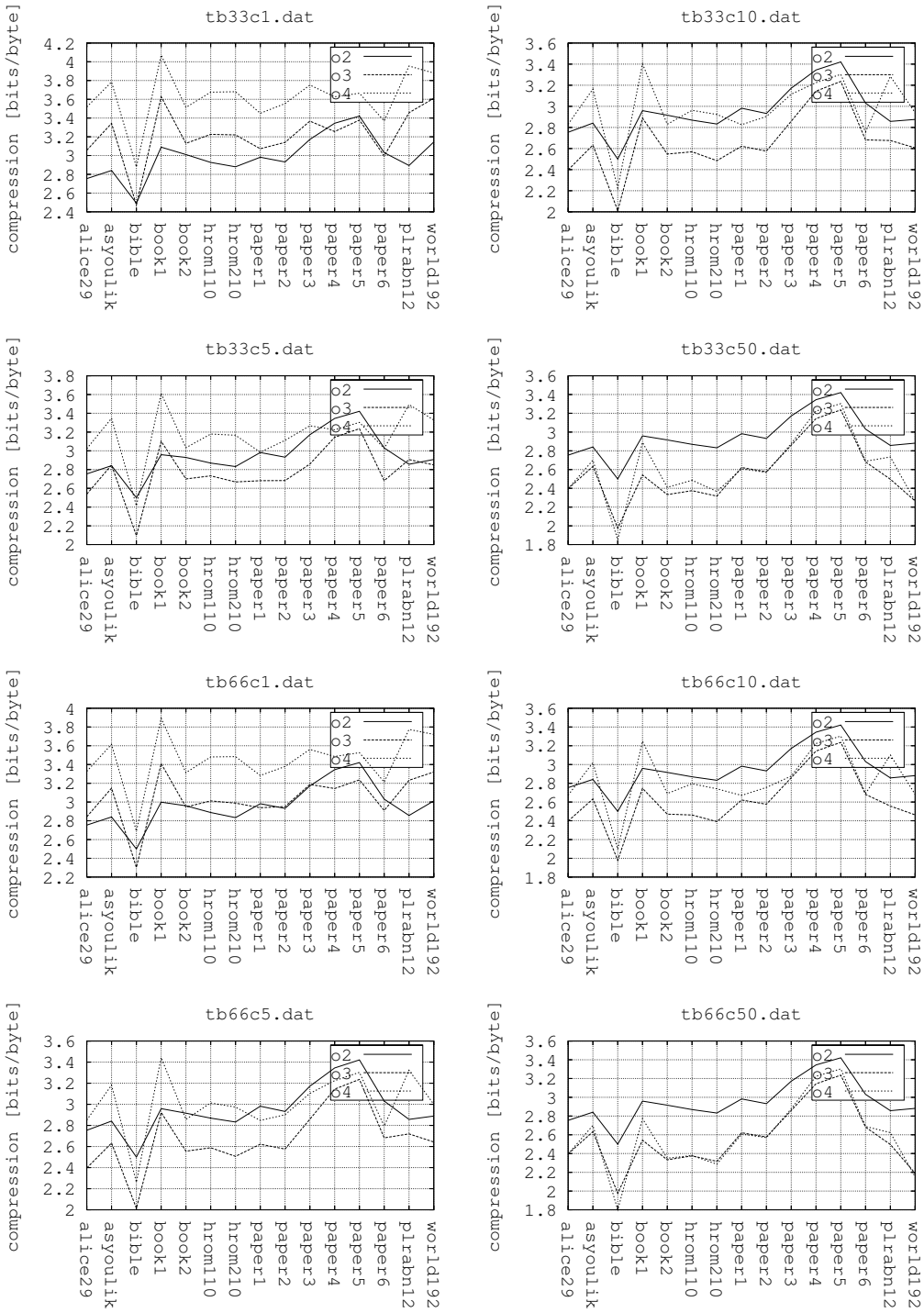


Figure 5.2: Adaptive compression performance for table sizes of **32768** and **65536** elements with a maximum number of 1000, 5000, 10000, and 50000 collisions. The measurements refer to section 2.5.3.

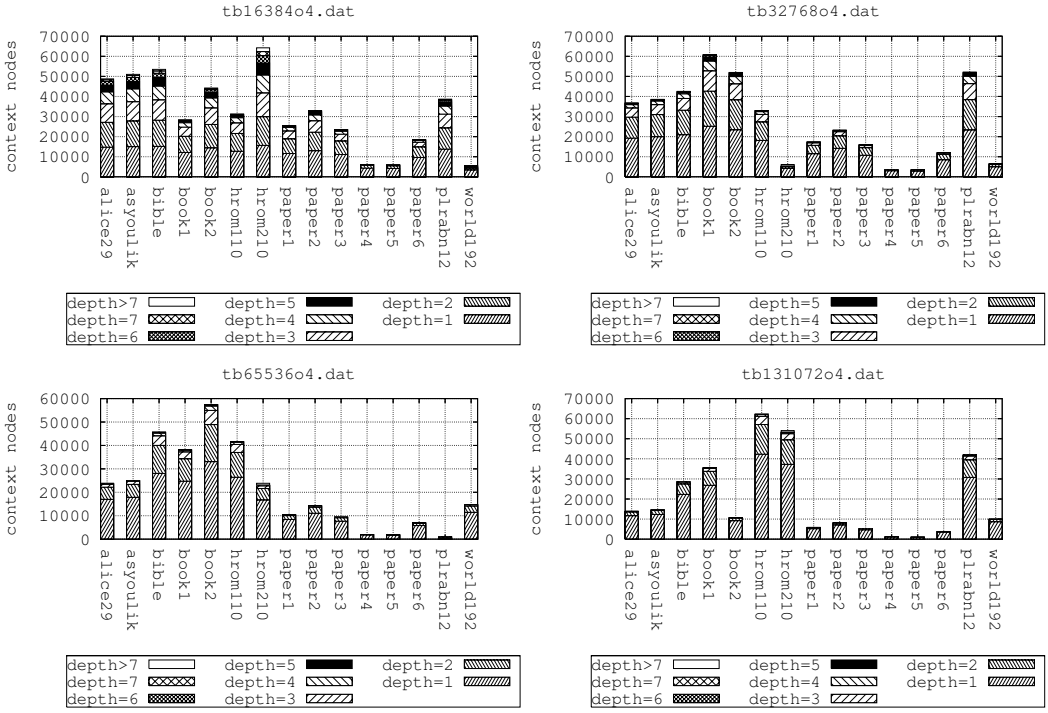


Figure 5.3: *CollisionStruct* plots for **order 4** for table sizes of 16384, 32768, 65536, and 131072. The measurements refer to section 2.5.5.

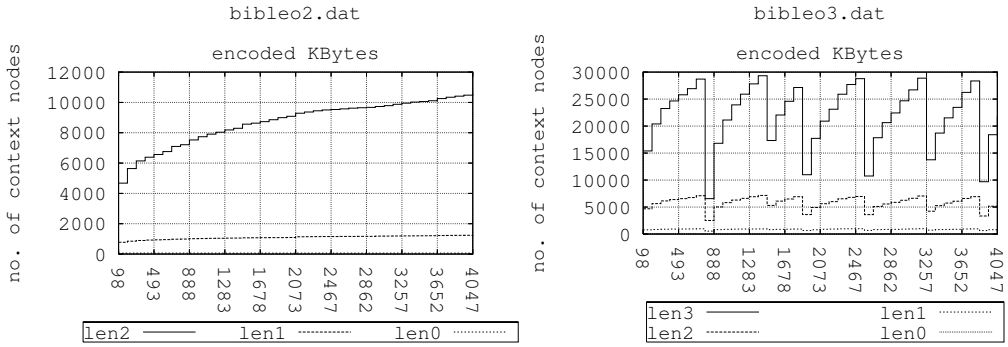
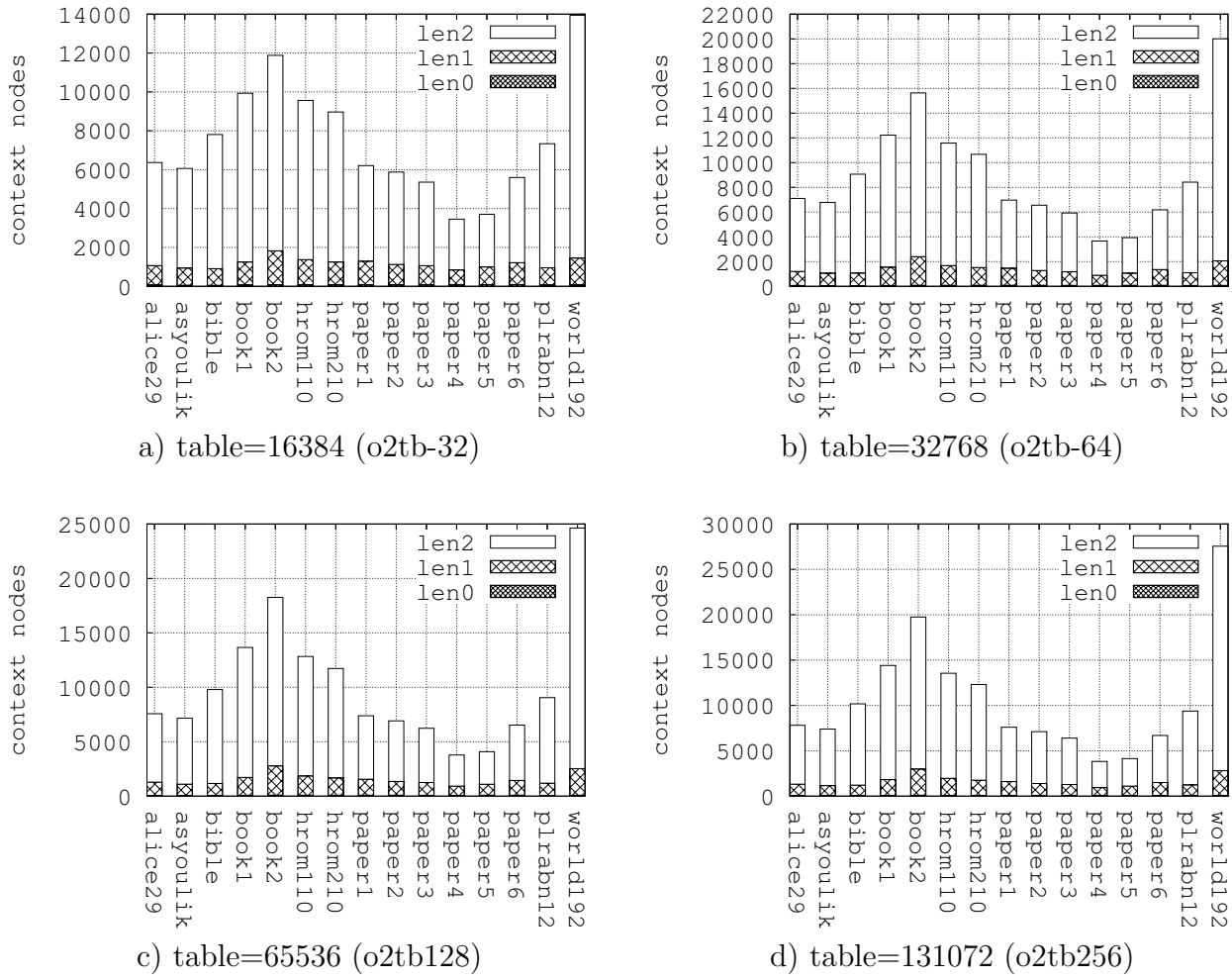


Figure 5.4: Keeping track of the number of context nodes for the file bible with orders 2 (left plot) and 3 (right plot). The memory is constrained to 32000 hash table elements and a maximum of 10000 collisions. For order 2 (left plot) there is no difference than with unlimited memory. For order 3 the model is though flushed for six times. The measurements refer to section 2.5.7.



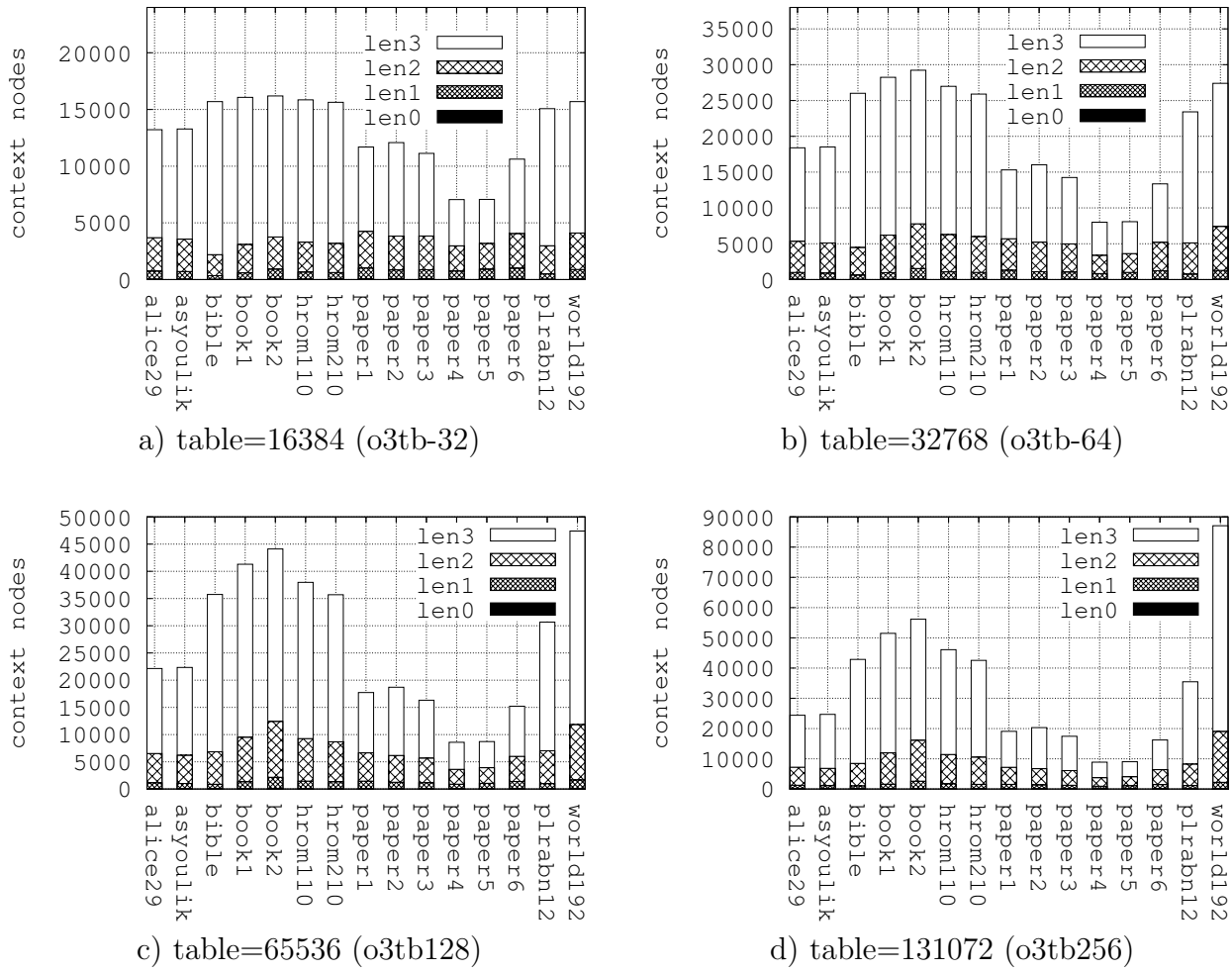
a) table=16384 (o2tb-32)

b) table=32768 (o2tb-64)

c) table=65536 (o2tb128)

d) table=131072 (o2tb256)

Figure 5.5: Number of context nodes for each training file categorized by their order (len0-len3) for model **order 2** when the collisions in the data structure are deleted. Figures a) - d) refer to the table sizes 16384, 32768, 65536, and 131072. The figure is an addendum to the figure on page 57. The here given results cannot be derived from the figures in sections 2.5.5 and 2.5.6, as the nodes that are located at the first position in the linked lists (non-collision nodes) were not categorized.



a) table=16384 (o3tb-32)

b) table=32768 (o3tb-64)

c) table=65536 (o3tb128)

d) table=131072 (o3tb256)

Figure 5.6: Number of context nodes for each training file categorized by their order (len0-len3) for model **order 3** when the collisions in the data structure are deleted. Figures a) - d) refer to the table sizes 16384, 32768, 65536, and 131072. The figure is an addendum to the figure on page 57.

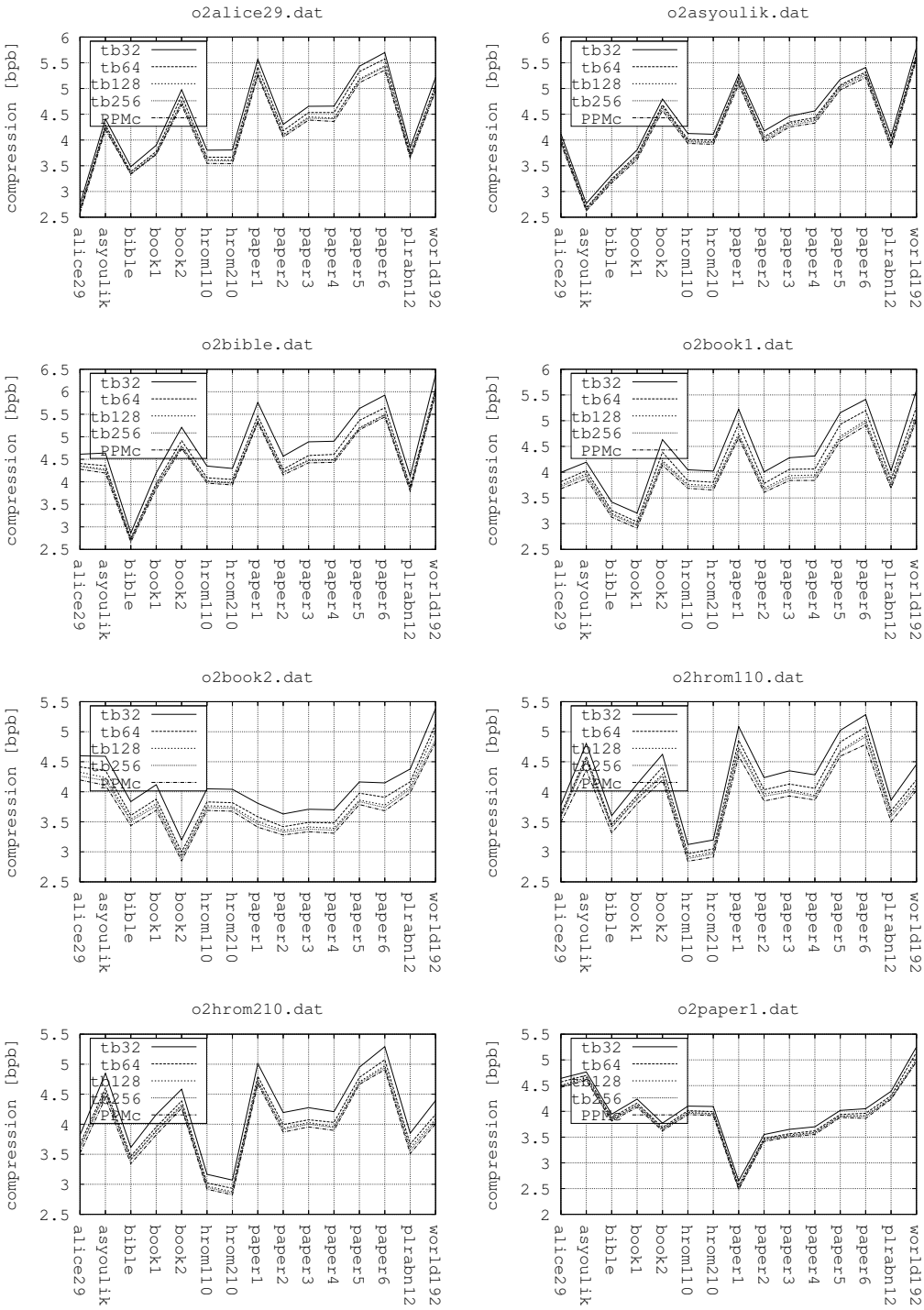


Figure 5.7: Additional plots for static context modeling in section 2.6.3.2 using the low complexity models from 32 to 256 kBytes for **order 2** with the training files *alice29*, *asyoulik*, *bible*, *book1*, *book2 hrom110*, *hrom210*, and *paper1*.

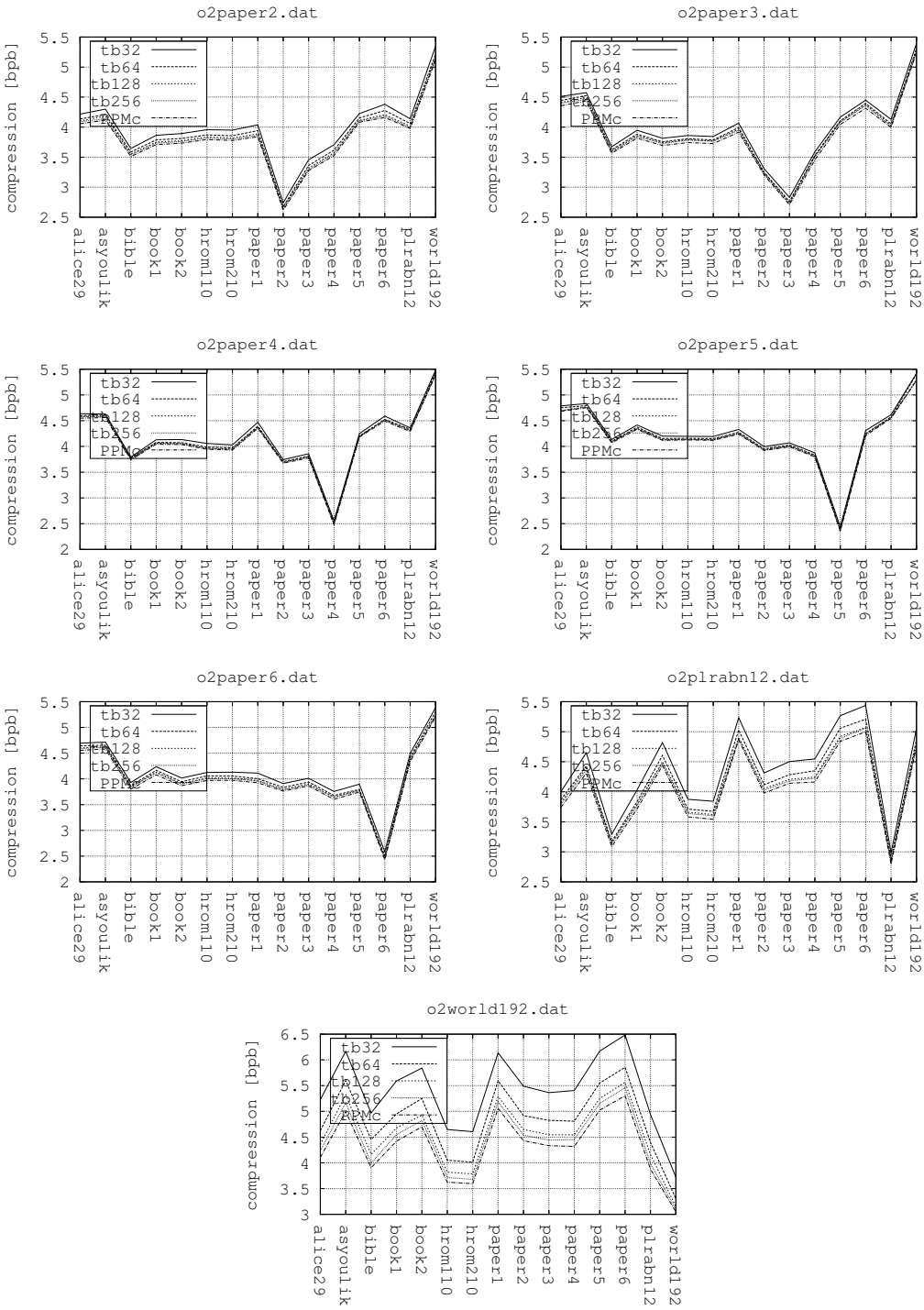


Figure 5.8: Additional plots for static context modeling in section 2.6.3.2 using the low complexity models from 32 to 256 kBytes for **order 2** with the training files *paper2*, *paper3*, *paper4*, *paper5*, *paper6*, *plrabn12*, and *world192*.

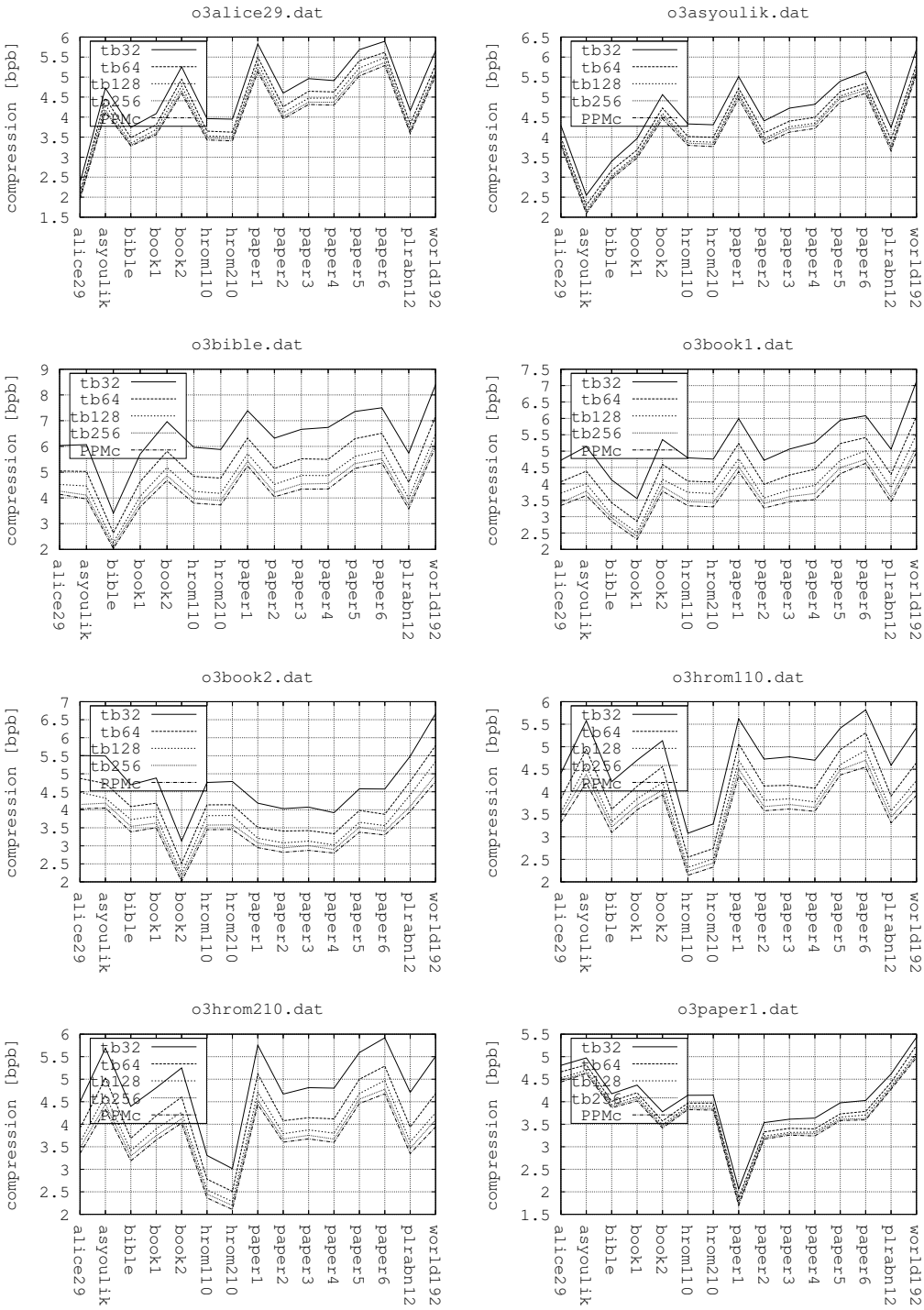


Figure 5.9: Additional plots for static context modeling in section 2.6.3.2 using the low complexity models from 32 to 256 kBytes for **order 3** with the training files *alice29*, *asyoulik*, *bible*, *book1*, *book2*, *hrom110*, *hrom210*, and *paper1*.

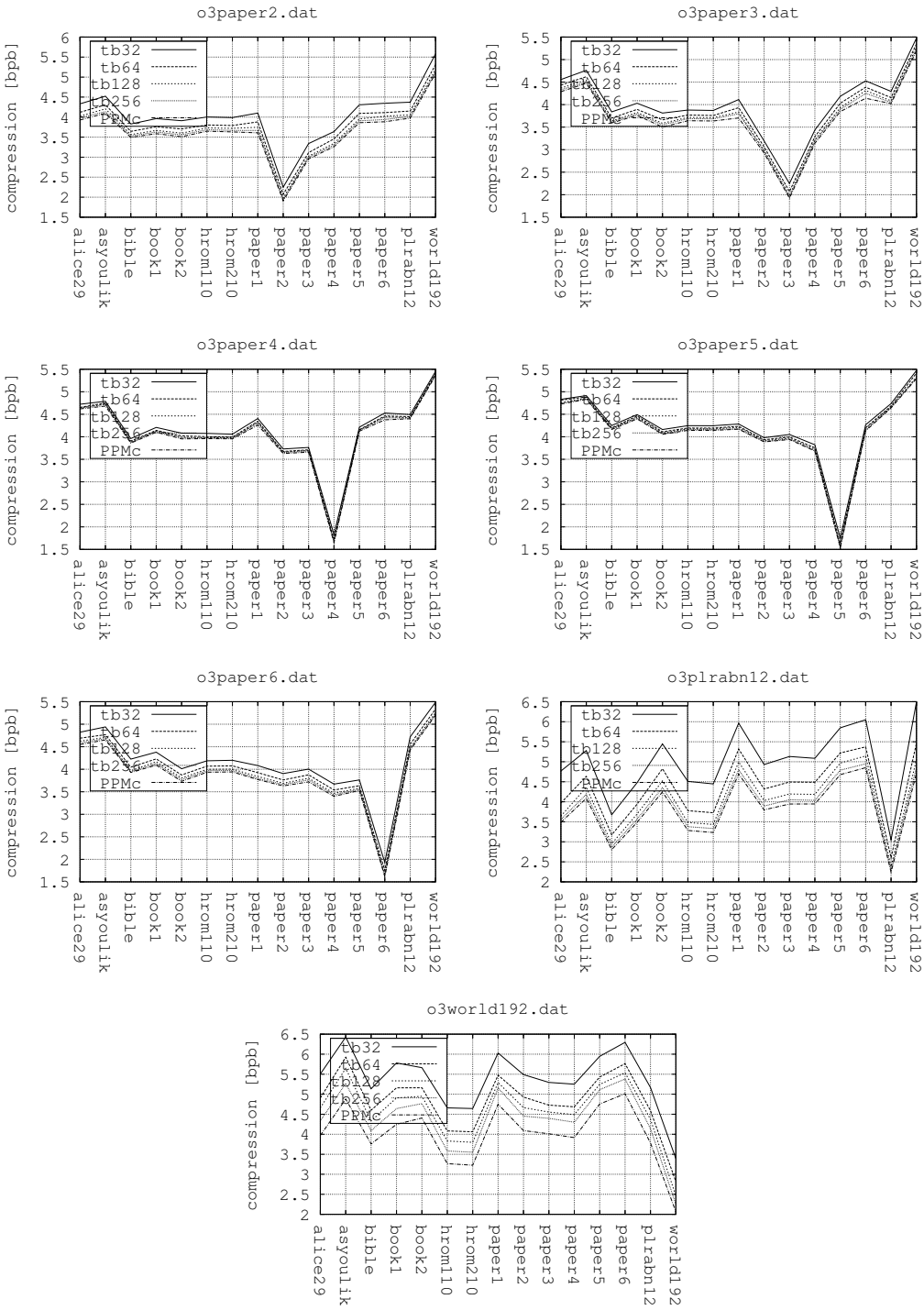


Figure 5.10: Additional plots for static context modeling in section 2.6.3.2 using the low complexity models from 32 to 256 kBytes for **order 3** with the training files *paper2*, *paper3*, *paper4*, *paper5*, *paper6*, *plrabn12*, and *world192*.

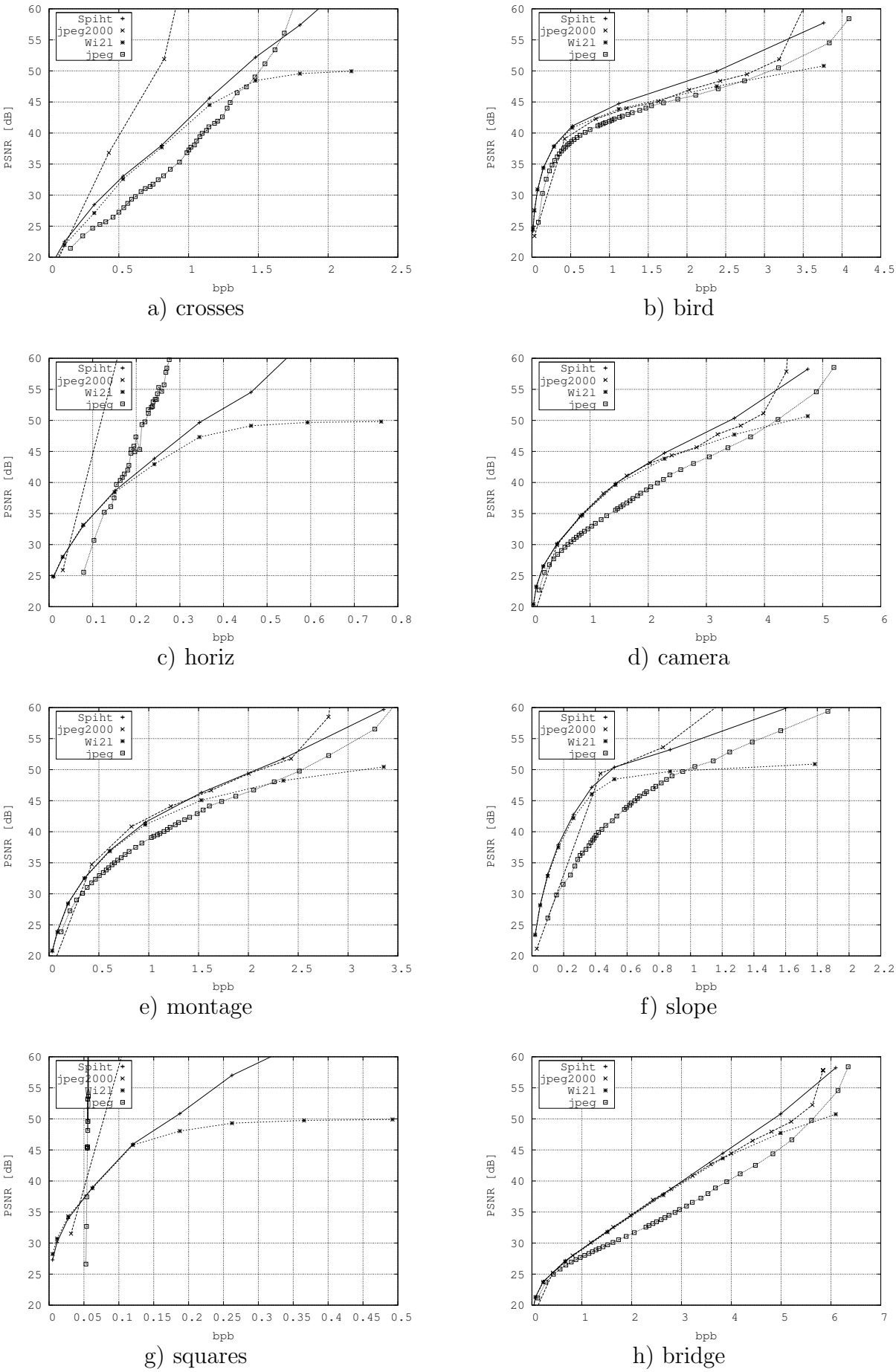


Figure 5.11: Comparison of the compression performance of the proposed system and the JPEG / JPEG2000 techniques. The figure refers to section 3.7.4.2.

Figure 5.12: Picture data base for the performance evaluation. The set contains the 12 pictures *bird*, *bridge*, *camera*, *circles*, *crosses*, *goldhill*, *horiz*, *lena*, *montage*, *slope*, *squares*, and *text*, which were employed for the evaluation of the fractional wavelet filter in section 3.5 and for the Wi2l coder in section 3.7.

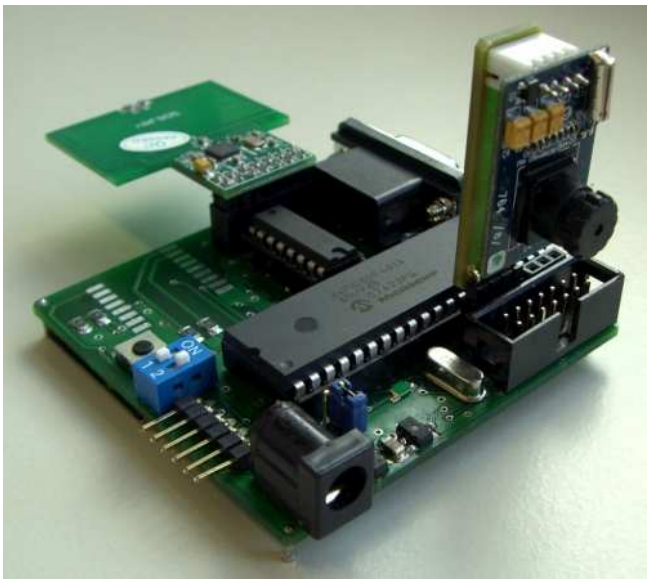


Figure 5.13: Picture of the redesigned sensor platform. The hardware is a follow-up version of the platform described in section 3.3 and was designed by Stephan Lehmann. To make the sensor easier to build the controller *dsPIC30F4013* with a total of 2 kByte RAM from Microchip was selected. The platform contains a multimedia card and a camera for the verification of the novel algorithms. It was employed to verify the fractional wavelet filter in section 3.5 and the Wi2I coder in section 3.7.

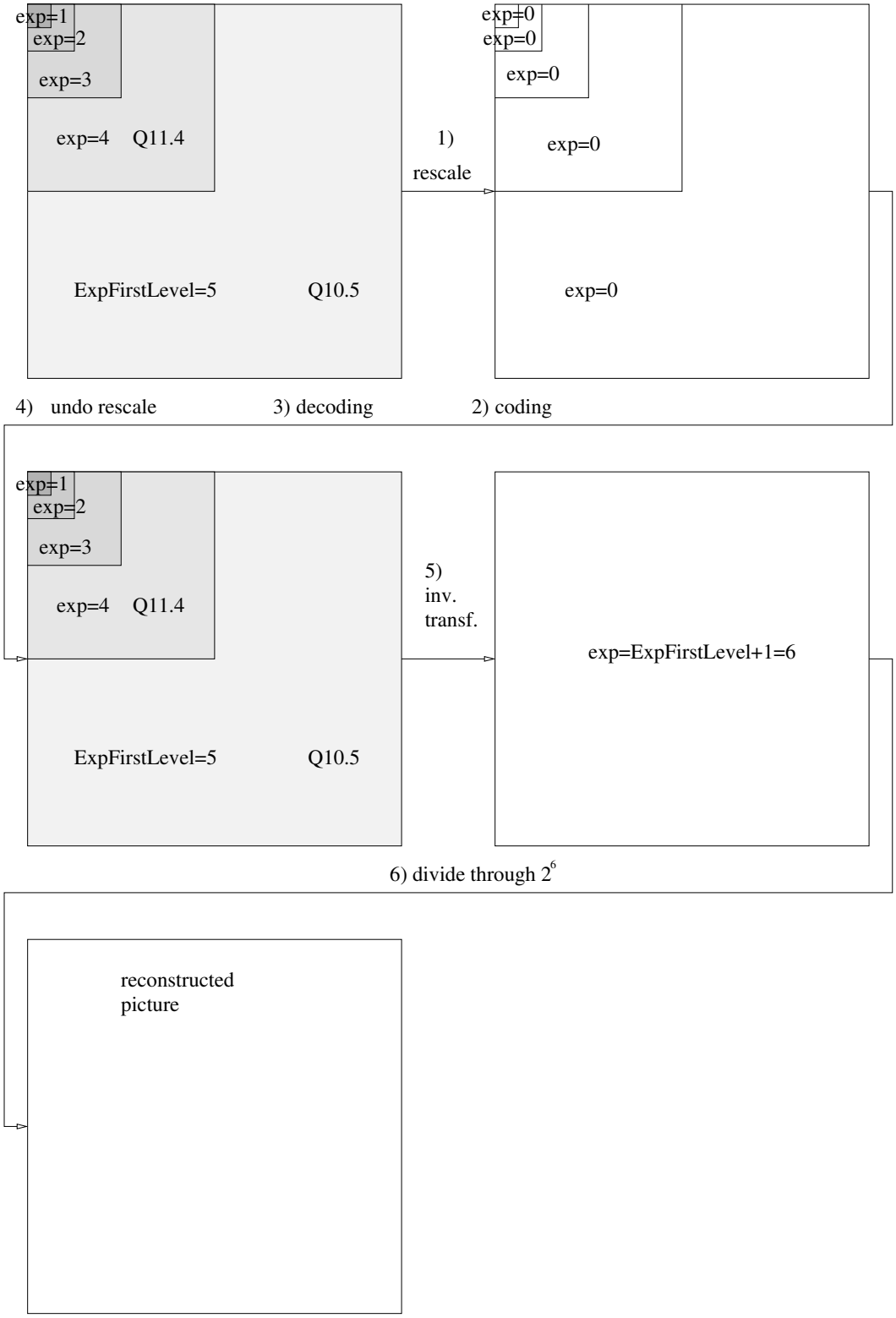


Figure 5.14: BCWT Coding with fixed-point Transform. The coding algorithm expects the wavelet coefficients in the integer data format. Thus a scaling procedure has to be performed in step 1). After coding and decoding this step has to be undone in step 4). Then the inverse transform can be performed in step 5). Finally, the picture values have to be scaled to the char data format. The figure refers to section 3.6.6, where the Bcwt implementation is verified and is also relevant for the evaluation of the novel Wi2l coder.

5.4 C-Code of the Hash-Function

HashFunction.cpp

```
unsigned int HashFunction(char* key,
                          unsigned char len,
                          unsigned int TableLen){
/* One-at-a-Time Hash, from Bob Jenkins tutorial
/  returns a number between 0 and TableLen-1
/  TableLen should be a power of 2 !!!
*/
  unsigned int hash,i;
  for (hash=0,i=0;i<len;i++){
    hash+=key[i];
    hash+=(hash << 10);
    hash^=(hash >> 6);
  }
  hash += (hash<<3);
  hash^=(hash>>11);
  hash+=(hash<<15);
  //estimate the number of bits necessary for the TableLen
  //log_b(x)=log_a(x)/log_a(b)
  unsigned int bits=
    (unsigned int) floor( log(double(TableLen))/log(2.) );
  //000...001<<8=000...100000000
  //000...100000000-1=000...011111111
  unsigned int mask=( (unsigned int) 1<<bits ) - 1;
  unsigned int index=(hash & mask);
  return index;
}
```

5.5 C-Code for One-Dimensional Fixed-Point WT

TestFixed.h

```
// TestFixed.h
// Rein Mai 2008

#include "coder.h"


void testfix();//performs an in-place wavelet transform of an input array
               //with data in INT16 format
void invtestfix();//inverse in-place wavelet transform of an input array

//analysis low- and high-pass filters:
INT16 FiltFixL(INT16 *line_pic,UINT16 index,UINT16 N,INT8 expIn,INT8 expOut);
INT16 FiltFixH(INT16 *line_pic,UINT16 index,UINT16 N,INT8 expIn,INT8 expOut);

//synthesis low- and high-pass filters:
INT16 FiltSynFixL(INT16 *line_pic,UINT16 index,UINT16 N,INT8 expIn,INT8 expOut);
INT16 FiltSynFixH(INT16 *line_pic,UINT16 index,UINT16 N,INT8 expIn,INT8 expOut);

INT16 AlFix(INT8 index);// analysis low and high-pass filters
INT16 AhFix(INT8 index);//
```

TestFixed.c

```
//TestFixed.c
//Rein Mai 2008

#include "SpisaTerm.h"
#include "TestFixed.h"
#include "FixedPoint.h"

void testfix(){
  Matrix *Mvec=GiveFirst();// input matrix is in INT16 format
  UINT16 N=GiveCols(Mvec);
  StackDataPointer vec;
  vec.INT8p=&(Mvec->vector[4]);
  INT16 j;
  INT16 dest_line[N];
  INT8 expIn=0;
  INT8 expOut=9;
  for(j=0;j<N/2;j++){
    dest_line[j]=     FiltFixL(&vec.INT16p[0],j*2,N,expIn,expOut);
    dest_line[j+N/2]=FiltFixH(&vec.INT16p[0],j*2+1,N,expIn,expOut);
  }
  for (j=0;j<N;j++) vec.INT16p[j]=dest_line[j];
}
void invtestfix(){
  Matrix *Mvec=GiveFirst();// input matrix is in INT16 format
  UINT16 N=GiveCols(Mvec);
  StackDataPointer vec;
  vec.INT8p=&(Mvec->vector[4]);
  UINT16 j;
  INT16 dest_line[N];
  INT8 expIn=9;
  INT8 expOut=9;
  for (j=0;j<N;j++){
    dest_line[j] =FiltSynFixL(&vec.INT16p[0],j,N,expIn,expOut);
    dest_line[j]+=FiltSynFixH(&vec.INT16p[N/2],j,N,expIn,expOut);
  }
  for (j=0;j<N;j++) vec.INT16p[j]=dest_line[j];

}

INT16 FiltFixL(INT16 *line_pic,UINT16 index,UINT16 N,INT8 expIn,INT8 expOut){
  INT16 c=0;
  INT8 i;
  INT16 pixel_ind;
  for (i=-4;i<=4;i++){
    pixel_ind=index+i;
    if (N==2) pixel_ind=abs(pixel_ind)%2;              //symmetric
    if (pixel_ind<0) pixel_ind=-pixel_ind;             //
    if (pixel_ind>N-1) pixel_ind=abs(2*N-2-pixel_ind);//extension
    c+=MulFixExp(line_pic[pixel_ind],AlFix(i),expIn,15,expOut);
  }
  return c;
}
INT16 FiltFixH(INT16 *line_pic,UINT16 index,UINT16 N,INT8 expIn,INT8 expOut){
  INT16 c=0;
  INT8 i;
```

```
  INT16 pixel_ind;
  for (i=-3;i<=3;i++){
    pixel_ind=index+i;
    if (N==2) pixel_ind=abs(pixel_ind)%2;                //symmetric
    if (pixel_ind<0) pixel_ind=-pixel_ind;               //
    if (pixel_ind>N-1) pixel_ind=abs(2*N-2-pixel_ind);//extension
    c+=MulFixExp(line_pic[pixel_ind],AhFix(i),expIn,15,expOut);
  }
  return c;
}
INT16 FiltSynFixL(INT16 *line_pic,UINT16 index,UINT16 N,INT8 expIn,INT8 expOut){
  INT16 c=0;
  INT8 i;
  INT16 pixel_ind;
  for (i=-3;i<=3;i++){
    pixel_ind=index+i;
    if (N==2) pixel_ind=abs(pixel_ind)%2;                //symmetric
    if (pixel_ind<0) pixel_ind=-pixel_ind;               //
    if (pixel_ind>N-1) pixel_ind=abs(2*N-2-pixel_ind);//extension
    if (!(pixel_ind%2))
      c+=MulFixExp(line_pic[pixel_ind/2], AhFix(i),expIn,15,expOut)\
          *(abs(i)%2 *-2+1);
  }
  return c;
}

INT16 FiltSynFixH(INT16 *line_pic,UINT16 index,UINT16 N,INT8 expIn,INT8 expOut){
  INT16 c=0;
  INT8 i;
  INT16 pixel_ind;
  for (i=-4;i<=4;i++){
    pixel_ind=index+i;
    if (N==2) pixel_ind=abs(pixel_ind)%2;                //symmetric
    if (pixel_ind<0) pixel_ind=-pixel_ind;               //
    if (pixel_ind>N-1) pixel_ind=abs(2*N-2-pixel_ind);//extension
    if (pixel_ind%2)
      c+=MulFixExp(line_pic[(pixel_ind-1)/2],AlFix(i),expIn,15,expOut)\
          *(abs(i)%2 *-2+1);
  }
  return c;
}
INT16 AlFix(INT8 index){
  switch(abs(index)){
  case 4:return  1240;//coefficients in Q0.15 format
  case 3:return -781;
  case 2:return -3625;
  case 1:return  12367;
  case 0:return  27941;
  }
}
INT16 AhFix(INT8 index){
  switch(abs(index)){
  case 3:return  2115;
  case 2:return -1333;
  case 1:return -13700;
  case 0:return  25837;
 default:return  0;
```

```
  }
}
```

5.6 Octave-Code for Wavelet Transform

fbi3.m

```
% function [sl,sh,al,ah]=fbi3();
% sl synthesis lowpass
% sh synthesis highpass
% al analysis lowpass
% ah analysis highpass

al = [ 0.037828 -0.023849 -0.110624  0.377403];
al = [al 0.852699 fliplr(al)];
ah = [ 0.064539 -0.040689 -0.418092];
ah = [ah  0.788486  fliplr(ah)];

sl = ah .* [-1 1 -1 1 -1 1 -1];
sh = al .* [1 -1 1 -1 1 -1 1 -1 1];
```

dwt.m

```
% The forward wavelet transform in Octave can be performed with \\
% line=[31 58 50 44 47 52 56 62];
% fbi3;  #load wavelet coefficients
% m=[dwt(line,al,1),dwt(line,ah,2)]; # wavelet transform


% dwt.m
% discrete wavelet filter for one level
% rein April 07
% row-wise action; updated feb 08
% S signal or picture
% h filter
% x where to start: use x=1 for lowpass and x=2 for highpass
function C=dwt(S,h,x);
len_h=length(h);

cl=size(S)(2);% columns of S
C=sconv(S,h);
C=C(:,x:2:cl);% sample down

endfunction
```

sconv.m

```
% symmetric convolution for a matrix
% rein, Jan 08
% filter boundary handling: half-sample symmetric

function result=sconv(S,h)
cl_S=size(S)(2);% no cols of S
len_h=length(h);


```

```
l=(len_h -1)/2;
if cl_S==1 S=repmat(S,1,len_h);
elseif cl_S<l+1
  for i = 1:l
    S= [S(:, 2*i), S, S(:,cl_S -1)];
  endfor
else
  S=[S(:,l+1:-1:2),S(:,1:cl_S),S(:,cl_S -1:-1:cl_S -l)];
endif

result=conv2(S,h)(:,len_h:len_h+cl_S -1);

return;
endfunction;
```

dwrec.m

```
% The reconstruction can be performed with
% rec=dwrec(m(1:4),sl,1)+dwrec(m(5:8),sh,2);

% dwrec.m
% discrete wavelet reconstruction filter for one level
% rein April 07
% S signal or picture
% h filter
function C=dwrec(S,h,x);
[rw_S,cl_S]=size(S);
C=zeros(rw_S,2*cl_S);% sample
C(:,x:2:2*cl_S)=S;    % up
C=sconv(C,h);
endfunction
```

5.7 C-Code for Fractional Filter

```
FLOAT LL_HL [N];
FLOAT LH_HH [N];
UINT8 line_pic [N];//current row
INT8 i;//current row index
INT8 j;//vertical filter index
INT8 k;//horizontal destination index
for (i=N/2-1;i>=0;i--){
  //init the two destination buffers:
  UINT8 b;
  for(b=0;b<N;b++){LL_HL[b]=0;LH_HH[b]=0;}
  for (j=-4;j<=4;j++){
    INT16 line_index=i*2+j;
    if (line_index<0) line_index*=-1;//sym.
    else if (line_index>N-1)          //exten
      line_index=2*N-2-line_index;    //sion
    //get the current line from the multimedia card:
    GetFromMMC(line_index*N,&line_pic,N,'pic');
    for (k=0;k<N/2;k++){
      FLOAT L=filtL(&line_pic,k*2,N);
      LL_HL[k]+=L*al(j);   //update LL
      LH_HH[k]+=L*ah(j-1);//update LH
```

```
22          FLOAT H=filtH(&line_pic ,k*2+1,N);
23          LL_HL[k+N/2]+=H*al(j);   //update HL
24          LH_HH[k+N/2]+=H*ah(j-1);//update HH
25        }
26    }
27    //here write the two buffers 'LL_HL'
28    //and 'LH_HH' to the MMC-card:
29    WriteToMMC(i*N,&LL_HL,N,'dest');
30    WriteToMMC((i+N/2)*N,&LH_HH,N,'WTpic');
31  }
```

5.8 Deutsche Zusammenfassung

In der Vergangenheit lag der Schwerpunkt bei der Entwicklung neuer Datenkompressionsverfahren auf der Verbesserung der Kompressionsrate, wobei mehr Rechenaufwand in Kauf genommen wurde, da sich die Computer Hardware immer stetig weiter entwickelt hat. Die aktuelle Entwicklung zeigt aber, dass mobile Geräte und drahtlose Sensoren immer mehr Verbreitung finden. Gängige Verfahren laufen jedoch auf diesen Systemen nicht, da sie nur wenig Rechenleistung und Speicher zur Verfügung stellen.

Der erste Teil dieser Arbeit stellt ein Verfahren vor, das kurze Nachrichten von 10 bis 400 Zeichen mit wenig Aufwand komprimieren kann. Dabei werden die Prinzipien der statistischen Kontext-Modellierung mit einem neuartigen skalierbaren Datenmodell kombiniert. Das vorgestellte Verfahren reduziert die Größe der Nachrichten auf die Hälfte, wobei es nur 32 kByte RAM Speicher benötigt. Außerdem wird durch Messungen belegt, dass Mobiltelefone damit Batterieenergie einsparen können.

Im zweiten Teil der Arbeit wird ein Verfahren mit geringer Komplexität zur Bildkompression mit Wavelets behandelt. Das Verfahren besteht aus einer neuen Berechnungsvorschrift für die Wavelet-Transformation von Bildern, dem sogenannten Fractional Wavelet Filter, und dem vorgestellten Wavelet Image Two-Line Coder (Wi2l). Der Speicherbedarf ist dabei extrem gering: Um ein 256x256x8 Bild zu komprimieren sind nur 1.5 kByte RAM Speicher notwendig, wobei nur 16 Bit Integer-Berechnungen durchgeführt werden. Die Technik wird auf einem kleinen Microcontroller mit 2 kByte RAM verifiziert und erreicht dabei trotzdem vergleichbare Ergebnisse wie aktuelle JPEG2000 Programme, die auf PCs laufen. Die neuartige Bildkompressionstechnik ermöglicht es, dass preiswerte Sensoren oder Sensornetzwerke Bildkompression anwenden können, die dem aktuellen Stand der Technik entspricht.