

Pilkington Library

Author/Filing Title FERREGRINO URIBE

Vol. No. Class Mark T

**Please note that fines are charged on ALL
overdue items.**

FOR REFERENCE ONLY

0402693094



A NOVEL APPROACH FOR THE HARDWARE IMPLEMENTATION OF A PPMC STATISTICAL DATA COMPRESSOR

by

Claudia Feregrino Uribe


A Doctoral Thesis

Submitted in partial fulfilment of the requirements
for the award of

Doctor of Philosophy
of
Loughborough University

December, 2001

© by Claudia Feregrino Uribe, 2001

 Loughborough University Physical Library	
Date	Dec. 01
Class	
Acc No.	040269309

ABSTRACT

This thesis aims to understand how to design high-performance compression algorithms suitable for hardware implementation and to provide hardware support for an efficient compression algorithm.

Lossless data compression techniques have been developed to exploit the available bandwidth of applications in data communications and computer systems by reducing the amount of data they transmit or store. As the amount of data to handle is ever increasing, traditional methods for compressing data become insufficient. To overcome this problem, more powerful methods have been developed. Among those are the so-called statistical data compression methods that compress data based on their statistics. However, their high complexity and space requirements have prevented their hardware implementation and the full exploitation of their potential benefits.

This thesis looks into the feasibility of the hardware implementation of one of these statistical data compression methods by exploring the potential for reorganising and restructuring the method for hardware implementation and investigating ways of achieving efficient and effective designs to achieve an efficient and cost-effective algorithm. The aim is to reduce the complexity of the method while maintaining its compression performance, offering the possibility of implementing the system using current technologies.

Three investigations were set for achieving our objectives. The first investigation explores an efficient compression method and identifies its main computational requirements. The second investigation looks into hardware structures used in compression chips and their impact in the statistical method. Also some methods are studied for simplifying the complexity of the model. The third investigation uses the results of the previous investigations to develop a new algorithm capable of hardware implementation, analysing so the interaction and tradeoffs between algorithmic desired characteristics and hardware capabilities for ensuring its effective mapping into compression architectures.

This thesis presents a study of a hardware friendly statistical compression method. It reports a new algorithm, which simplifies the computational tasks while maintaining the algorithmic functionality and performance results. It also explains design issues and provides hardware support for the algorithm.

The performance of the model results contained in this thesis enable us to identify under what conditions statistical compression methods offer performance benefits. This may help designers to incorporate statistical compression into future compression applications.

ACKNOWLEDGEMENTS

I wish to thank my supervisor, Professor Simon Jones, for his guidance and encouragement during my research. I must also thank him for enabling me to attend many international conferences during these years.

I also want to thank my colleagues of the Electronic Systems Design Group at Loughborough University both past and present, whom have always provided support and encouragement.

For my financial support I thank the National Council for Science and Technology of Mexico (CONACyT), without whom this work would have not been possible.

Thanks to Dr. Martin Litherland who went meticulously over the entire text and made me numerous corrections and suggestions.

And special thanks to my parents, Gabriel and Ma. del Carmen, for their unconditional love and support throughout the years. To my husband René with whom I discussed many aspects of my research, for his encouragement and support, and the wonderful times we have had together. And finally to Gabriel and Paty and the rest of my family, I am indebted to you all.

TABLE OF CONTENTS

Chapter 1

Introduction	1
1.1 Data Compression	1
1.2 Applications and Implementations	2
1.3 Motivation	4
1.4 Aims of the Research	6
1.5 Structure of the Thesis	7

Chapter 2

Review	9
2.1 Objectives of the Chapter	9
2.2 Data Compression	9
2.3 Data Compression Techniques	10
2.3.1 Ad-hoc Methods	11
2.3.2 Dictionary-based Methods	12
2.3.3 Other Methodologies	17
2.4 Statistical Compression	18
2.4.1 Statistical Coders	19
2.4.2 Statistical or Markov Models	23
2.4.3 Hybrid Methodologies: Statistical and Dictionary-based	27
2.5 Existing Data Compression Implementations	28
2.5.1 Software Implementations	29
2.5.2 Hardware Implementations	30
2.6 Comparison of Data Compression Implementations	34
2.7 Summary	36

Chapter 3

Overview of Investigations	38
3.1 Objectives of the Chapter	38
3.2 Identification of Research Topics	38
3.3 Introduction to Investigations	39
3.3.1 PPMC Algorithmic Compression Investigation	39
3.3.2 Reorganisation of the PPMC Algorithm	40
3.3.3 Shift Model Implementation and Performance	41
3.4 Tools and Verification	41

Chapter 4

PPMC Algorithmic Investigation	44
4.1 Objectives of the Chapter	44
4.2 The PPMC Model	45
4.3 The Arithmetic Coder	48
4.4 The PPMC Algorithm: PPMC Model + Arithmetic Coder	51
4.5 Key Computational Requirements of the PPMC Model	53
4.5.1 Software Implementation of the PPMC Model	54
4.5.2 Searching Process	55
4.5.3 Updating Process	55
4.5.4 Arithmetic Coding Operations	57
4.5.5 Other Issues	57
4.5.6 Discussion	58
4.6 PPMC Compression Performance	60
4.6.1 PPMC Compression Performance - Experiment	61
4.6.2 Order of the Model - Experiment	64
4.6.3 Block Size - Experiment	66
4.6.4 Dictionary Size - Experiment	68

4.6.5	Discarding Policy – Experiment	70
4.6.6	Summary	75

Chapter 5

Reorganisation of the PPMC Algorithm		77
5.1	Objectives of the Chapter	77
5.2	Algorithmic Redesign	77
5.3	Simplification of Arithmetic Coding Operations	78
5.4	Reorganisation of the PPMC Model	79
5.4.1	‘Biggest plus one EF’ Method for Model Updating	85
5.4.2	Other Methods - Experiment	88
5.4.3	Combining Strategies – a New Method	93
5.4.4	Effect of Sorting - Experiment	95
5.4.5	Conclusions	98
5.5	Summary	99

Chapter 6

Shift Algorithm	100
6.1 Review of Objectives	100
6.2 Optimisation of the PPMC Model for Hardware Implementation	100
6.2.1 Order of the Model - Experiment	101
6.2.2— Dictionary Size - Experiment	104
6.2.3 Multi-dictionary Model - Experiment	106
6.2.4 Multi-dictionary Model Experiment – Resizing the Dictionary	109
6.2.5 Constant Total Frequency Counts - Analysis	111
6.2.6 Number of Tokens - Experiment	113
6.2.7 Positions to Shift Frequencies - Experiment	119

6.2.8	Frequencies and Positions Shifted - Experiment	124
6.2.9	Parallel Frequency Updating - Discussion	127
6.2.10	An Approximation for Updating Symbol Frequencies - Experiment	128
6.3	Shift Algorithm	133
6.3.1	Assumptions	136
6.3.2	Methodology	136
6.3.3	Results and Analysis	136
6.3.4	Conclusions	139
6.4	Summary	140

Chapter 7

Chapter 7		141
Hardware Modelling		
7.1	Objectives of the Chapter	141
7.2	Hardware Modelling	141
7.2.1	Design Tools	141
7.2.2	Assumptions	142
7.2.3	Model Architecture	142
7.2.4	Test bench	145
7.3	Hardware Requirements	145
7.4	Summary	148

Chapter 8

Chapter 8		149
Conclusions		
8.1	Objectives of the Chapter	149
8.2	Review of the Objectives	149
8.3	Conclusions	151
8.4	Measurements of Success	153

8.5	Shortcomings of the Work	155
8.6	Future Investigations	155
8.7	Summary	156

LIST OF FIGURES

Figure 2.1	Data compression classification	11
Figure 2.2	Classification of dictionary-based techniques	12
Figure 2.3	Sliding window in LZ77 compression method	14
Figure 2.4	Classification of statistical compression techniques	18
Figure 4.1	Example of the PPMC model information and corresponding range in arithmetic coder	52
Figure 4.2	Structure of a lexicographical tree.....	54
Figure 4.3	Compression comparison of several data compressors and PPMC	62
Figure 4.4	Impact of block size on PPMC compression performance	67
Figure 4.5	Compression ratios for different sets of data using the PPMC model with different dictionary sizes.....	69
Figure 4.6	LRU and Climb discard policies	71
Figure 4.7	Data structure for 3 rd order PPMC model	72
Figure 5.1	Array for frequency counts.....	80
Figure 5.2	Different alternatives for approximating the model updating	81
Figure 5.3	Algorithms and examples of methods for model updating in PPMC	84
Figure 5.4	Example of 'Biggest Plus One EF' method	86
Figure 5.5	Compression ratios obtained with 'Biggest Plus One EF' method	87
Figure 5.6	Compression ratios obtained with methods to approximate the updating process of the PPMC model.....	90
Figure 5.7	Stages of the compression process	90
Figure 5.8	Detail of the stages of the compression process.....	91
Figure 5.9	Compression ratio on combined strategies.....	94
Figure 5.10	Perfect and partial sorting in 'Biggest plus one EF' method.....	97

Figure 6.1	Compression ratio as <i>FNT</i> changes for 0 th order model.....	116
Figure 6.2	Compression ratio as <i>FNT</i> changes for 1 st order model.....	117
Figure 6.3	Compression ratio as <i>FNT</i> changes for 2 nd order model	118
Figure 6.4	0 th order model compression results as <i>m</i> and <i>FNT</i> varies.	121
Figure 6.5	Compression ratios with 1 st order model.....	122
Figure 6.6	Compression ratios with 2 nd order model.....	123
Figure 6.7	Architecture for frequency updating	128
Figure 6.8	Average <i>TIS</i> as the symbols seen in the current context increases	131
Figure 6.9	Shift algorithm in serial form	133
Figure 6.10	Shift model in parallel form	134
Figure 6.11	2 nd order Shift algorithm compression for Canterbury Corpus	137
Figure 6.12	Compression ratios for Memory Data.....	137
Figure 6.13	Compression ratios for Thesis Data	138
Figure 7.1	Architecture of the Shift compression model.....	143
Figure 7.2	Dictionary architecture of shift model.....	144
Figure 7.3	Flow of data in the test bench.....	145

LIST OF TABLES

Table 2.1	History of PPM class of compression models	24
Table 2.2	Space requirements for PPMC model	26
Table 2.3	Main characteristics of compression hardware implementations in research	32
Table 2.4	Main characteristics of commercial compression hardware implementations.....	33
Table 4.1	Example of the PPMC model	47
Table 4.2	Example of the PPMC model, after symbol 'u' followed context 'th'	47
Table 4.3	Example of the PPMC model with cumulative frequencies.....	49
Table 4.4	Example of the PPMC model, after symbol 'm' followed context 'th'	56
Table 4.5	Growth of the number of tokens as the order of the model increases	59
Table 4.6	Performance of different order models.....	65
Table 4.7	Discarding policies	70
Table 5.1	Methods for model updating.....	82
Table 5.2	Complexity of perfect and partial sorting.....	96
Table 6.1	Percentage predictions for each model order - Canterbury Corpus.....	102
Table 6.2	Percentage of predictions for each order of the model - Memory Data ..	103
Table 6.3	Average number of positions used in the dictionaries.....	105
Table 6.4	Compression performance of the compression model.....	107
Table 6.5	Space requirements for a single dictionary.....	108
Table 6.6	Space requirements for separated dictionaries	108
Table 6.7	Space requirements resizing the dictionaries.....	111

Table 6.8	Formulae required for the PPMC and Shift algorithms.....	114
Table 6.9	Best choice of values for <i>FNT</i> and positions to shift.....	118
Table 6.10	Best choices for <i>m</i>	123
Table 6.11	Fixed number of tokens, <i>FNT</i> , and number of positions to shift, <i>m</i>	126
Table 6.12	Compression ratio results obtained with PPMC and Shift algorithms	138
Table 6.13	Comparing complexity of PPMC and Shift algorithms.....	139
Table 7.1	Estimated size of compression components	146
Table 7.2	Estimated system size based on a 1 st order Shift model	147

CHAPTER 1

INTRODUCTION

1.1 DATA COMPRESSION

With the explosive growth of telecommunications there is an ever-increasing demand for faster and better digital devices. Nowadays, global telecommunication networks and the explosion of Internet usage produce large amounts of data to be transmitted or accessed in the least possible time. Some methods enable these tasks, enhancing the scope and cost-effectiveness of transmission and storage of data.

Data compression is one of the methods that have contributed to make digital devices handle large volumes of data. It allows systems to gain space for storage or increase the bandwidth for data transmission, offering a vehicle for cost reduction and efficient operation. Examples of devices that benefit from data compression are routers, hard disks, and modems.

In data networks, although there are many strategies for optimising traffic, including priority queues, access lists and filters, one of the more effective is to reduce the amount of data by compressing it over the network [Cisco97], reducing significantly the time the data takes to reach its destination point. Data compression is applied to both low-speed links (for example those using modems) and to long-haul links to increase throughput.

In local-area networks (LANs) compression technology can help to reduce transmission bottlenecks if the network transmits data slower than the computers can generate it [Pawlikowski95]. For wireless LANs to be seamlessly and transparently integrated with wired LANs, they must support comparable data rates. Then data compression can be used to inexpensively reduce the amount of data to be transmitted,

thus improving the effective bandwidth of the communication channel and in turn the overall network performance.

The capacity of data storage devices is also further improved with data compression. Magnetic hard disks and tapes, among others, have been using data compression to reduce the amount of data they store. And although holographic data storage and other methods [Ashley00] that do not require traditional compression methods are being developed, magnetic storage devices are far from declining. Instead, magnetic hard disks technology has sped up [Comenford00] requiring each time better, and faster, techniques for data compression.

Most commonly transmitted types of data are significantly compressible [Holtz93], however, encrypted or already compressed data can not be further compressed [Hifn01]; in fact, it might originate data expansion [Bell90]. Compressible data includes text, fax, executable program files, audio, image and video. Audio, images and video are highly compressible and may tolerate some loss of information when compressed, a process known as lossy compression, whereas data that does not tolerate any loss of information is processed by lossless compression techniques [Smith99].

1.2 APPLICATIONS AND IMPLEMENTATIONS

A large number of compression algorithms have been implemented either in software or hardware to meet the demands of a variety of applications and devices, '*reducing storage barriers and breaking system bottlenecks*' [IBM01].

Compression applications may enable and simplify operations as information backup, Web searches and document transfers on the Internet. Most of the implementations of compression algorithms have been in software, mainly for off-line applications such as data storage, where low compression speed and the best possible compression ratios are needed. Software compression is also used in on-line applications. However, the increasing speeds demanded in networks have made it more difficult for this type of compression to deliver the appropriate throughputs.

Throughput demands in compression implementations have varied significantly as technology improves. *'Starting from simple disk file compression some years ago with low speed requirements, data compression chip sets are now spreading to virtually all high-speed networks'* [Holtz93]. Communication systems can use either hardware or software compression, depending on the speed of the design.

The foremost compression applications were for hard disks, which have evolved according to the system requirements. Most utilities designed to relieve cramped hard disks have been based on file compression, although driver level compression is also used [Hovingh01]. File compression software essentially rewrites file data so it tends to take up less space on a hard disk. Driver level compression operates transparently, with compressed files or applications looking and behaving as if they were not compressed. File compression software used in disks exclude certain frequently accessed files from compression to gain speed, whereas driver level compressors compress everything and apparently increase disk savings more than file compression software.

In networks, there are different forms of data compression in use today. For example, there are TCP/IP header compression, link compression and multi-channel payload compression. According to [Mello], header compression shrinks the disproportionately large headers but leaves the data payload uncompressed. It is used to improve throughput of low speed lines. In link compression, the entire frame, both protocol header and payload, gets compressed and encapsulated to ensure error correction and packet sequencing. This method handles large packets and unlike header compression it is protocol independent. Multi-channel payload data compression provides the best overall data compression solution and typically yields the best compression ratios when there are many different sources of data [Mello]. While a number of vendors support multi-channel payload data compression, it is worth noting that implementations vary and the differences can affect overall performance depending on how vendors define data payload compression. Also, IP headers that remain uncompressed by payload compression can be a significant number of bytes long, thus, header compression must work in conjunction with this form of payload compression

for overall compression to be effective, although this may increase processor loading and packet delays.

Data compression implementations can be used in external compression devices or embedded in products as integrated compression hardware or integrated compression software. System architecture, compression algorithms and method of implementation are important issues to consider when evaluating a product's overall data compression performance.

1.3 MOTIVATION

Compression implementations, either software or hardware, favour some applications depending on the type and amount of data to be transmitted or stored as well as the speed requirements. Data compression may have some potential problems if the right method of implementation or compression algorithm is not properly chosen. Some of these problems are data expansion, error propagation and incompatible standards. However, if the method is properly chosen, it multiplies the throughput and saves space without harmful side effects.

Vendors of data communication equipment attempt to present compression devices that offer the highest compression ratios, the simplest implementation and the widest applications. The task of data compression is becoming harder, in order to meet the demanding requirements it is necessary to develop faster and better compression algorithms that adapt to the demands of network and storage applications.

All compression algorithms can be implemented on general-purpose processors. However, in some applications the requirements for throughput, compression ratio and cost may prove difficult to overcome. The advance of semiconductor technology has made possible the integration of complex systems within a single chip. Hardware compression implementations are transparent to the user, they use simple algorithms that balance memory requirements and computational complexity at the expense of compression ratio. A special-purpose compressor offers the possibility of performing a

specific, well-defined compression algorithm as a self-contained system that can be integrated into an existing device or can be developed as a single external compression device. This device may act as a black box that takes in data from a device and sends data out into another device or transmission line.

We exemplify with routers the use of data compressors, but the range of applications is wide and other devices as tape drives, hard disks, printers, among others, also use it.

External data compression devices offer benefits to routers that connect high-speed links. The reason is that data compression is highly processor intensive with multi-channel payload data compression being the extreme case. External data compression devices also become a requirement when there is a concentration of lower speed links. Building a network that includes external devices for data compression is expensive, considering the cost of routers and the compression devices at each side of the link.

Some router vendors use internal data compression implementations based in software for the advantages it may have in the provision of multi-channel compression, lower cost and simplified single device management. However, this design burdens the main processor and consumes memory, which adversely affects router performance and packet latency. An improved internal data compression approach can provide dedicated hardware built into the router for compressing and decompressing data. By using a separate dedicated processor for data compression, all of the other basic functions within the router continue to be processed simultaneously. This parallel processing minimises the packet delay that can occur due to the significant processing required for data compression.

We have emphasised just the use of hardware compression as a means of improving the capacity of high-speed systems. However, another important factor to consider is the type of algorithm being implemented. For a long time, simple algorithms offering a good balance between compression efficiency and space requirements have been implemented in hardware. Recently, a different class of algorithms known as statistical algorithms has shown to be extremely effective, consistently meeting or exceeding the compression ratios of the simple techniques, although they are generally more

complex and time consuming, which has prevented their hardware implementation [Effros00].

To implement a special-purpose compressor architecture it is necessary to select a suitable algorithm for hardware implementation. Once the algorithm has been selected, the number of processing functions that are needed can be identified. These functions must be performed at high speed, and the implementation must be cost effective and better with alternative hardware solutions. The fact that the data compression algorithms perform a well-defined set of operations offers the possibility of exploiting many of its characteristics to achieve more efficient execution using specialised architectures. Then, a carefully selected algorithm together with an architecture that matches it would substantially increase the performance. Although this concept is not new and some manufactures currently offer a number of hardware compression chips, statistical algorithms have not been developed and exploited in current devices and these algorithms may well be the basis of the next generation of data compressors.

1.4 AIMS OF THE RESEARCH

This work concerns practical implementations of lossless data compression algorithms. The aims of the research are to understand how to design high-performance compression algorithms suitable for hardware implementation and to provide hardware support for an efficient compression algorithm.

The area of lossless data compression has evolved to a practical level over recent years. Current hardware implementations of compression algorithms are used in a wide range of applications. However, many of these designs have been concentrated on optimising compression speeds and they usually neglect other important aspects.

This thesis examines the following three areas in particular,

- *Analysing efficient compression algorithms:* good compression techniques generally are complex and use computationally demanding methods. Our goal

is to identify the key computational requirements and other issues that influence their functionality.

- *Simplified algorithmic processes:* our goal is to reduce the complexity of the algorithm and understand how this reduction impacts its performance.
- *Analysis of the interaction and tradeoffs between algorithmic desired characteristics and hardware capabilities to ensure the effective mapping of algorithmic computational requirements into compression architectures.*

From this knowledge we can develop new algorithms which simplify the computational tasks while maintaining the characteristic algorithmic functionality and performance results. Also we can provide hardware support for the algorithm exploiting hardware capabilities.

1.5 STRUCTURE OF THE THESIS

Chapter 1 introduces the area of data compression. It discusses aims and objectives and describes the structure of the thesis.

Chapter 2 provides a review of relevant work in the area of lossless data compression including some compression algorithms and the methods that have been developed to implement them. This chapter also shows comparisons of these algorithms and their implementations and finally argues about the potential benefits of relatively recent, and not practically used, statistical data compression algorithms to improve compression performance of current compression chips.

Chapter 3 introduces the experimental investigations and describes in more detail the research objectives and methodology followed. It also describes the tools used for the development of experiments required in this thesis and the verification method.

Chapter 4 researches the PPMC statistical compression algorithm to identify the main functional requirements of the model and the coder, and other issues that may have an impact on compression. It presents the series of experiments that assist into this investigation including the outcomes of comparing it with current data compression implementations.

Chapter 5 explores the reorganisation of PPMC algorithm, including some alternatives to simplify the PPMC model operation, aimed for hardware implementation. It presents the series of experiments that are important to this investigation.

Chapter 6 describes in detail the issues of designing and implementing a PPMC statistical algorithm, taking into account the knowledge gained from the previous two chapters. Finally, it shows the integration of these issues within the algorithm.

Chapter 7 explains the hardware architecture requirements of the compression algorithm and presents an estimated of performance. Additionally, it analyses the performance of the statistical model and determines how close it corresponds to the expected results.

Chapter 8 concludes the thesis by summarising the main points and discussing whether the objectives have been achieved. It also examines the strengths and shortcomings of the work and, finally, it outlines further research areas that may be of interest.

CHAPTER 2

REVIEW

2.1 OBJECTIVES OF THE CHAPTER

This chapter reviews the relevant work in the area of lossless data compression, including compression technologies and current data compressors. More specifically, the objectives of the chapter are to:

- Briefly review the relevant background on lossless data compression.
- Review software and hardware implementation of lossless compression algorithms.
- Place the work pursued in this thesis in context with related work.

2.2 DATA COMPRESSION

Since this thesis is related to the design of efficient hardware compression algorithms, it seems helpful to start this review by briefly outlining:

- The meaning of data compression
- Its main methods and implementations
- The main issues in its implementation

Data compression represents information with the fewest possible number of bits and is defined as the removal of the redundant information from a piece of data producing an equivalent but shorter message. Decompression is the reverse process, the reinsertion of redundant information. Depending on the application, compression can be lossy or lossless.

Lossy compression is especially useful for analogue data such as images, audio and video where some loss of information may be tolerated. The method removes some of the source information content along with the redundancy. This causes distortion of the data and thus the decompression process can not fully restore the original data. However, if some distortion is tolerated, the systems may achieve higher compression ratios, which is the ratio of the number of compressed bits to the number of original bits before compression, and enable transmission of data within critical times.

Lossless compression does not tolerate any loss of information. It is a process completely reversible and it is used for digital data, as those processed by computer systems like text, object or alphanumerical. The price it pays for not tolerating loss of information is poorer compression ratio.

This thesis is concerned purely with lossless data compression, and from now on, whenever the term *data compression* is used it will refer to *lossless data compression*.

2.3 DATA COMPRESSION TECHNIQUES

This section shows the main compression methods and some of their implementations, explaining their main characteristics and functionality, and what gives them the suitability for hardware or software implementations.

As Figure 2.1 shows, the field of data compression is divided into three classes of compression methods: *ad-hoc*, *dictionary based* and *statistical*. Several *ad-hoc* techniques have been developed over the years and some of them are currently used to assist other compressors. *Dictionary-based methods* are the most widely studied type of algorithms and are used in a broad range of applications due to their simplicity. *Statistical techniques* offer considerable improvements in compression ratio at the expense of space or speed requirements. In this figure, the coloured route is the one followed in this thesis. All the methods will now be explained in more detail; the statistical techniques have been allocated a complete section.

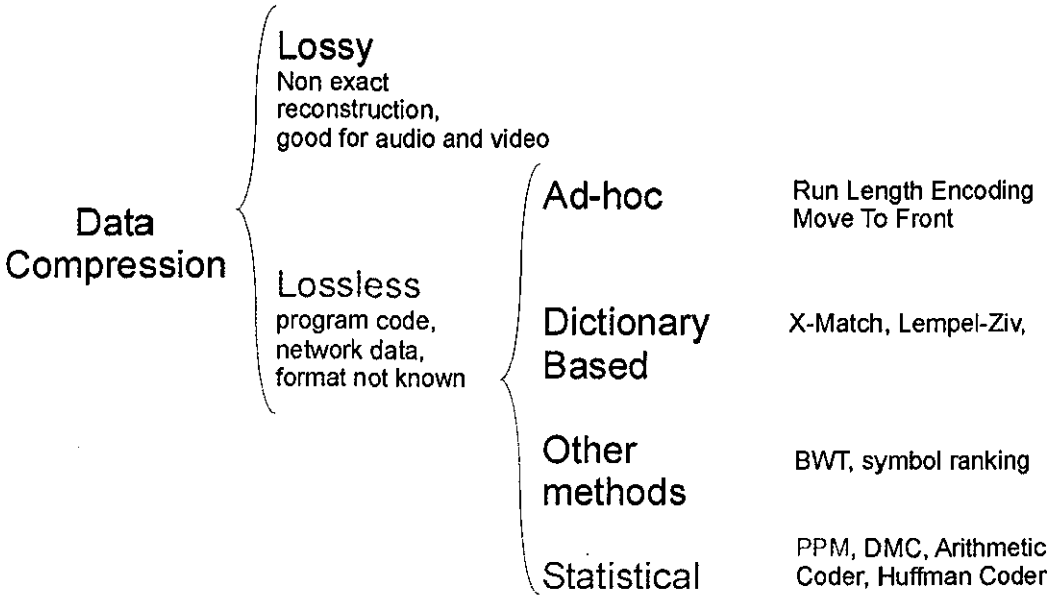


Figure 2.1 Data compression classification

2.3.1 Ad-hoc Methods

Ad-hoc methods exploit particular characteristics of the data. They were developed mainly in the early days of data compression and comprise Run Length Encoding (RLE) [Gollomb66] and more recently Move-to-Front (MTF) [Bentley86, Elias87 and Moffat89] the most successful of this class.

RLE takes advantage of the presence of consecutive identical single symbols often found in data streams. It replaces long runs of repeated symbols with a special token and the length of the run. This method is particularly useful for small alphabets and provides better compression when symbols are correlated with their predecessors.

MTF technique is designed for input streams that can be broken up into words. It assumes a word just occurred is more likely to occur later on again. So, as the inputs are processed, they are placed in the first position of a list. When one of them comes again it is coded with the index of its current location and it is placed on the top of the list. The indexes may be codified with variable code lengths [Bell90] to achieve more compression.

An example of an implementation of both techniques is the X-Match algorithm [Jones00]. It looks for matches or partial matches between an incoming word and the words maintained in a dictionary using the MTF strategy. The strategy considers as the most important property of the scheme the locality of reference where, as [Bentley86] mentions, *‘if a word has been recently used then it will be near the front of the list and therefore have a short decimal encoding’*. X-Match makes use of this property and views as highly probable that the first position in the list will be repeated such that long runs of this location may occur, using then RLE to encode these runs. More information about this algorithm can be found in later sections looking in more detail at its hardware characteristics.

2.3.2 Dictionary-based Methods

In *dictionary-based methods*, groups of consecutive symbols are replaced by a code. These methods are based on breaking the input stream into blocks of symbols and replacing them by indexes of some dictionary. The dictionary holds blocks of data expected to occur frequently and the indexes are chosen in such a way that on average they consume less space than the phrase they represent.

Some authors divide dictionary techniques into static, semi-adaptive and adaptive [Bell90], while others divide them into static or dynamic (adaptive) [Salomon98]. We consider the first classification more complete; it is the one used in this thesis and shown in Figure 2.2.

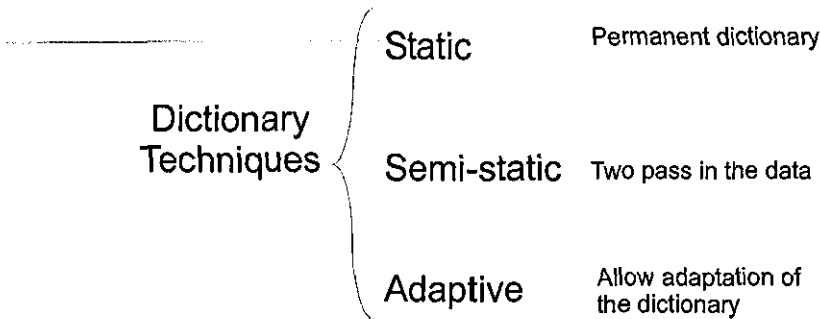


Figure 2.2 Classification of dictionary-based techniques

Static dictionaries are constructed before the compression starts and they are permanent. They are particularly useful when the type of data to compress is known in advance and offers compression with a little effort but does not allow adaptation to unforeseen data.

Semi-adaptive dictionaries are considered as 'two-pass' dictionaries, where the occurrence of the symbols is first calculated and the dictionary initialised, then the dictionary adapts as symbols come in. One example is a scheme proposed by Larsson and Moffat [Larsson99]. They developed a new method for creating the dictionary. The benefit of these dictionaries is the optimisation of the compression performance as the 'statistics' of the data are obtained before encoding. A disadvantage of this scheme is the need of storing a large part of a message in memory. Additionally, requiring two passes in the data makes them not practical for communication applications.

The fact that static dictionaries do not allow adaptation makes them inadequate for universal compression, i.e. compression of any type of data. Also semi-static ones requiring two passes in the data prevents them for real-time applications. Thus, the third type of dictionaries, adaptive, is the most suitable for real-time dictionary-based compression applications.

Adaptive dictionaries hold strings of symbols previously found in the input stream and allow for additions and deletions of strings as new inputs are processed.

Among these adaptive techniques in dictionary based-methods, the *LZ class of algorithms* is the most popular in data compression for providing a satisfactory balance between compression and speed while requiring a modest amount of memory.

The first and most popular dictionary-based algorithm is the LZ77 [Ziv77] developed by Lempel and Ziv. It subdivides a data source into two parts as Figure 2.3 shows. The first part represents n previously encoded symbols, which becomes the dictionary and is usually some thousands of bytes long. The second part represents x symbols to be encoded, it is usually some tens of bytes long [Held96]. They usually are implemented in a sliding window of $n+x$ symbols, which is initially empty. To encode a symbol, the

first n symbols of the window are searched, from right to left, to find the longest match with the dictionary buffer. The coder codifies the offset of the longest match, the length of the match and the first symbol that did not match.

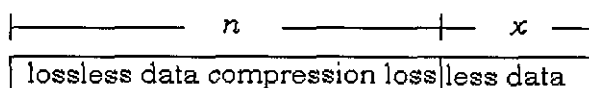


Figure 2.3 Sliding window in LZ77 compression method

In [Craft98], Craft mentions that LZ77 has some advantages particularly for disk storage use. It shows not only better compression on the smaller data sizes desirable for random access applications, but amenability for a fast and simple CMOS hardware implementation based on a CAM [Gajski97] array. This allows the input data-string-matching operations required for the compression algorithm to be performed efficiently at high speeds.

Next we show several implementations of the LZ77 algorithm to demonstrate the compression performance that they can achieve as well as the wide range of possible hardware implementations.

- Ranganathan and Henriques proposed in [Ranganathan93] a VLSI implementation that exploits pipelining and parallelism to obtain high speed and throughput. A dictionary of 256 or at most 512 bytes is recommended for a hardware implementation, since it provides a reasonable choice while providing good compression efficiency. They used a parallel architecture of n processors, where n is the size of the longest possible match. The number of comparisons was reduced from a quadratic order in the sequential algorithm to linear order with this parallel implementation. A prototype CMOS VLSI chip was designed and fabricated using CMOS $2\mu\text{m}$ technology implementing a systolic array of nine processors. Based on the estimates from the prototype design, they estimated the chip could yield a compression rate of 13.3 MB/s operating at 40 MHz.

- In [Bongjin98], Bongjing and Burleson also present a parallel LZ VLSI implementation. The simulation operates at 12.5 MB/s with a clock speed of 100 MHz using 1.2 μm CMOS technology, it uses a 4.1 KB SRAM and 32 processors.

In spite of the years of difference between these two implementations that allow better hardware technologies to be used, Ranganathan's implementation achieves higher throughput with a slower clock cycle than Bongjin's chip. This may well be attributed to the pipelining exploitation in the former implementation.

- In [Kim95], Kim *et al.* present an efficient VLSI architecture for this algorithm but in non-systolic hardware architecture. It is claimed that the design require less area and fewer clock cycles than existing architectures at that time, but the compression performance of the algorithm is not shown explicitly.
- In [Lee95], Lee and Yang also present an implementation of LZ77 algorithm that achieves high throughput by implementing the sliding window in a CAM array. The clock speed is up to 50 MHz and accepts one sample per clock cycle; then it attains a throughput of 50 MB/s. The process is 0.8 μm CMOS technology and the compression ratio is between 0.66 and 0.28, which is a wide range. On average this algorithm gives a compression ratio of 0.5.

The LZ77 method has some inefficiency that Lempel and Ziv almost immediately improved with the LZ78 method [Ziv78], also known as LZ2. It does not use any search buffer, look-ahead buffer, or sliding window, instead, there is an adaptive dictionary of previously seen symbols. It starts empty or with a few symbols and its size is limited by the amount of memory available. It outputs a pointer to the dictionary where the match occurred and the code of a symbol. The dictionary never deletes any entry, which is an advantage over LZ77, since new strings may be matched with previous strings, although the available space for the dictionary fills up soon.

The TAG compressor [Bunton92] implements the LZ78 algorithm; it may compress up to 20 MB/s using a 2 μm CMOS technology. It makes use of RAM and CAM

memories [Gajski97] to store the dictionary and can be scaled to support larger dictionaries.

The algorithms LZ77 and LZ78 are the basis of a number of variants, a clear summary and a short description is presented in [Bell90]. [Salomon98] also describe in detail these variants that improve the compression performance of the original algorithms. Next we describe two of the main variants of these algorithms, namely LZW and LZS.

Probably the most popular variant of the LZ class of algorithms is the LZW, developed by Welch in 1984 [Welch84]. It is a descendant of the LZ78 algorithm and it is by far the most commonly used in practical applications. LZW as well as LZRW1 [Williams91a] compression algorithms find in hash tables a powerful tool for fast search operations mainly in software-based applications. Hash tables are data structures that allow fast insertions, searches and deletions of data.

A good example of an application of the LZW algorithm is the recommendation V.42bis of the International Telegraph and Telephone Consultative Committee (CCITT, now International Telecommunications Union ITU) [V.42bis and Thornborson92] for implementing data compression and increasing the data transmission rates in high-speed modems. Unisys holds the patent of the LZW algorithm although V.42bis modems presently require patent licenses from British Telecom, IBM and Holtz [Holtz93]. The V.42bis standard, as Thornborson foretold, has been widely used for more than ten years and with the increasing number of Internet users the use of modems will continue for some years until the widespread introduction of new technologies.

Another popular variant is the LZS (Lempel-Ziv-Stac) which is the most common type of compression found in networking hardware and software. Stac Electronics developed it as a proprietary version of the LZ77 algorithm. It maintains a history of the last 2 KB of input data as well as other data structures to accelerate the compression operation. Many of the products using this algorithm have a characteristic for tuning the algorithm, trading compression ratio by compression speed. Hi/fn holds the patent of LZS and sold licences to Cisco, U.S. Robotics,

Ascent, IBM and Novell for their products. Also the Microsoft Point-to-Point (MPPC) compression software uses LZS. This algorithm has also been standardised by the ANSI, QIC, the Frame Relay Forum and others [Hifna]. Hardware implementation characteristics of this algorithm are mentioned in sections 2.6 and 2.7.

2.3.3 Other Methodologies

There are data compression algorithms that do not belong to the classification mentioned. These include methods as the Burrows-Wheeler Transform (BWT) [Burrows94] and symbol ranking [Fenwick96b] among others currently entering the data compressors market.

The BWT method transforms a block of data into a format extremely well suited for compression for its later codification. Burrows and Wheeler recommend using MTF technique and an entropy coder. The authors state that their algorithm *'achieves speed comparable to algorithms based on the techniques of Lempel and Ziv, but obtains compression close to the best statistical modelling techniques'*. According to Fenwick, this method is not as good as the best of PPM-style compressors. Additionally, the technique requires a large number of computational operations as showed in [Fenwick96a].

The symbol ranking method uses some symbols seen in the immediate past to prepare a list of symbols likely to occur. The list is arranged according to the probability of occurrence. The position of the current symbol in the list is then encoded. The method uses an LZ77-type dictionary where the searches are done in a similar form to this algorithm. Without giving compression measurements, Salomon states that this method is *'slow but produces excellent compression'* [Salomon98], while Fenwick mention speed as its main characteristic [Fenwick98], estimating that a hardware implementation of 30 MB/s *'should be possible without trouble'*.

A detailed study of these two compression techniques is in [Fenwick96a] including some improvements and comparison with other effective techniques.

2.4 STATISTICAL COMPRESSION

In *statistical* compression each symbol is assigned a code based on their probability of occurrence. Highly probable symbols get short codes and less probable symbols get larger codes. It has been shown that these schemes overcome most practical dictionary-based implementations, as it can be seen in an algorithmic comparison in [Bell90].

Statistical compression methods are divided for their study in two separated stages, modelling and coding, as Rissanen considered appropriated in 1981 [Rissanen81], see Figure 2.4. The model maintains statistics of the source to facilitate efficient compression and the coder maps the statistics into bits. This scheme allows studying either the model or the coder separately and this separation is important because *‘it permits any degree of complexity in the modeller without requiring any change to the coder. In particular, the model structure and probability estimates can change adaptively’* [Howard94].

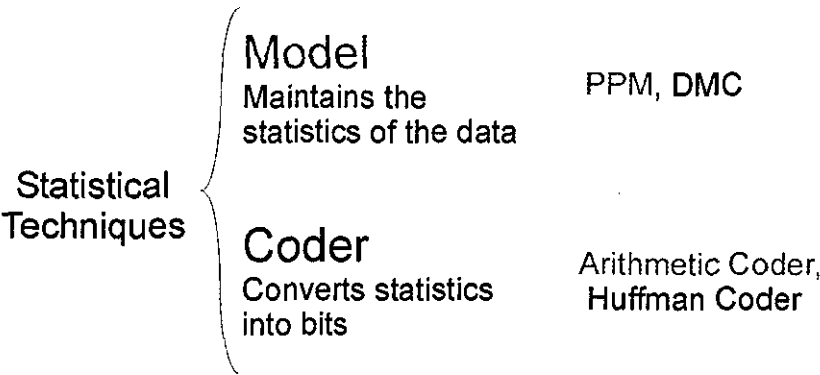


Figure 2.4 Classification of statistical compression techniques

The next section explains the best-known coders and models used for statistical compression in more detail.

2.4.1 Statistical Coders

The two most important statistical coders that have drawn the attention of many researchers in data compression are Huffman and arithmetic coders [Salomon98].

2.4.1.1 Huffman Coder

This method developed by Huffman in 1952 [Huffman52] serves now as the basis for many popular programs used in computers. In its static form, the method consists of building a list of symbols and their probabilities in descending order to construct a tree [Horowitz95] with a symbol at each leaf of the tree. The procedure for building the tree involves the two symbols with the smallest probabilities to build their parent node whose probability is the sum of the individual probabilities. This parent node represents a symbol that replaces the two children symbols in the list. This step is repeated until the list contains only one symbol. The tree is binary where at each level, starting from the root, assigns a 0 to the left branch and 1 to the right one. Then, traversing the tree, the code of a string follows the path in the tree concatenating the 0's and 1's accordingly. This scheme gives short codes to more probable symbols and longer codes to less probable ones.

Since no occurrence frequencies are known in advance in some applications, a semi-adaptive scheme may be followed. However, this mechanism is too slow for real-time applications and in practice an adaptive strategy is used [Salomon98]. The compressor and decompressor start with an empty Huffman tree and as symbols are input the tree is modified to adapt itself to the statistics of the data.

In [Holtz93] Holtz claims that *'although the Huffman method has been known since 1952 there have been few practical applications and it may be used as an addition to other compressors, but it is not commercially viable by itself'*. However, the widespread use of digital libraries, document databases and the Internet as well as digital television has generated new applications for this algorithm. At the moment, it is the base of the most effective compression technique used in information retrieval

systems [Ziviani00] for directly searching the compressed text without decoding the entire text from the beginning. So, contradicting Holtz' view some implementations use just the Huffman coder while others use it as a part of a multi-step compression technique [Salomon98].

- One example of hardware implementation of Huffman coder is found in [Liu94], it presents a dynamic Huffman coder that maintains a tree implemented in CAM and ROM [Gajski97] cells. Simulation results show that the encoder can yield an input throughput of 5MB/s operating at 40 MHz and assumes 0.5 compression ratio that is the average achieved by the algorithm. The chip uses 0.8 μ m CMOS technology and uses about 17.7 K gates.
- Other example is the implementation of [Benschop96] of a hybrid methodology that combines Lempel Ziv and Huffman coding, which is the LZH algorithm used in some software programs. Benschop built a VLSI implementation consisting of a bus interface, the sliding window coder (LZ), buffering and statistics module, Huffman coder and an internal processor that performs the Huffman tree computations and controls the other modules. The sliding window coder operates at a rate of one input character per clock cycle with a clock frequency of 12.5 MHz, that is 12.5 MB/s. This is the same throughput as Bongjin's LZ VLSI implementation and slightly slower than Ranganathan's LZ77 chip.

2.4.1.2 Arithmetic Coder

In [Bell90] the intricate birth and history of the arithmetic coder is reviewed, since the early publications by Abramson [Abramson63], Rissanen and Langdon's description [Rissanen81] oriented towards hardware implementation until the last well-recognised paper by Witten *et al.* [Witten87] which made available a software implementation.

Since its development, the arithmetic coder has been one of the most popular coders. From the theoretical point of view, it is the coder that compresses data closest to the entropy of the source, which produces the best compression than any compressor can

achieve. From the practical point of view, it has been long noted for its complexity and slow performance. However, due to its compression ratios, it has been studied to diminish its complexity and improve its performance speed.

The basic form of the arithmetic coder uses a static model [Salomon98]. It assigns a code to an entire message, instead of assigning individual codes to symbols. A message is represented by an interval of real numbers between 0 and 1. The interval becomes smaller as the message becomes longer and the number of bits to codify this interval increases. Successive symbols of the message will reduce the size of the interval according to the symbol probabilities generated by the model. The more likely symbols reduce the range less than the unlikely symbols, thus, more likely symbols add fewer bits to the code. The adaptive form of this algorithm is used in real-time transmissions.

Recently, the arithmetic coder was considered as the *'method of choice for adaptive coding on multi-symbol alphabets because of its speed, low storage requirements, and effectiveness of compression'* [Moffat98] in a paper that incorporated several improvements over the early version of Witten [Witten97]. One of the software implementations of arithmetic coder in [Moffat98] uses a particular data structure to compute cumulative frequency tables in arithmetic coding. This structure is a binary indexed tree [Fenwick94 and Fenwick95] that provides fast access time either constant or proportional to the logarithm of the table size. Other data structures that have been used to speed up the coding process is a multiple-linked list or multi-list structure used by Howard in the quasi-arithmetic coder [Howard93a], similar to vine pointers mentioned in [Bell90].

Through all the years of study of this coder there have been investigations into hardware implementations of binary and multialphabet arithmetic coding to effectively exploit compression speed. Other forms of diminishing speed created multiplication-free codification [Lei95] and/or division-free [Rissanen89 and Jou99].

Langdon and Rissanen and Pennebaker *et al.* [Langdon82 and Pennebaker88] present methods for coding the binary alphabet. In [Rissanen89 and Chevion91] Rissanen and

Mohjuddin and Chevion *et al.* present methods for non-binary alphabets. While Rissanen and Mohjuddin describe an implementation of arithmetic codes where the proper multiplication is avoided even with non-binary alphabets, and also where no division operation is required, Chevion *et al.* present a simpler and more efficient implementation than the Rissanen method.

The following are some examples of hardware implementations of arithmetic coder:

- In [Jou99], Jou and Chen proposed an implementation of a binary arithmetic coding. Their simulation indicates that their chip compresses at about 1.5 MB/s with a clock rate of 50 MHz.
- In [Kuang98], Kuang *et al.* designed an adaptive division-free binary arithmetic coding and implemented a prototype of a chip using the standard cells of 0.8 μm single-poly double-metal (SPDM) technology. The clock rate is 25 MHz and the compression speed is about 3 Mb/s. The complexity is 54 Kgates and requires 4 SRAM memories of 0.25 KB each.
- A multi-alphabet arithmetic coding presented in [Printsz93] uses a static model and is implemented in a series of FPGA's. It uses 11.5 KB of RAM and has a throughput of 16 MB/s, although no decompression is presented.
- Parallel implementations [Jiang94 and Lee97] have also been proposed and other successful variations named quasi-arithmetic coder [Howard94] and Q-coder [Pennebaker88]. Arps *et al.* implemented in VLSI the method of Pennebaker *et al.* in a chip designed for bilevel images [Arps88]. Although it is targeted for bilevel image data, it serves as a reference. It is a custom chip with embedded SRAM memory with technology 1.5 μm , and complexity of 13 Kgates. The clock speed is 10 MHz and the throughput is between 2.5 and 1.25 MB/s, depending on the data type compressed, whether images or code data. This speed is relatively lower than other arithmetic implementations but definitively can not compete with LZ VLSI implementations.

- In [Lee97], Lee *et al.* present a VLSI implementation of arithmetic coding which divides the input symbols into a number of groups and processes them in parallel. They claim that their implementation improves speed, expandability and latency of the conventional arithmetic coding. However, neither speed nor compression results are shown.

In summary, an endless number of investigations into the arithmetic coder have taken place since its development. Algorithmic modifications have been done to simplify its operational complexity and to improve its compression performance. Arithmetic coder performs better when coupled with a model that effectively estimates the statistics of the source [Salomon98]. It can use any statistical model. However, the best statistics are provided by the class of Markov models, which are explained in the following section.

2.4.2 Statistical or Markov Models

2.4.2.1 Prediction by Partial Matching

Prediction by Partial Matching (PPM) [Cleary84] is one of the most popular models for arithmetic coder, it was developed by Cleary and Witten in 1984. The scheme maintains a statistical model of data, assigning probabilities to the symbols and sending these probabilities to arithmetic coder. The probabilities are assigned according to the most recent symbols seen. Arithmetic coder then maps these probabilities into code bits.

The simplest statistical model counts the number of times each symbol has occurred in the past and assigns a probability to the symbol accordingly. The next model up is *context-based*, where not just the frequency of the symbol is used to predict but the more recently symbols seen. The symbols recently seen are called *context* and the number of them is the *order* of the context.

Compared with arithmetic coder, there has been little research about PPM models. The first model was developed in 1984 [Cleary84] and the next main result came in 1990, Moffat's PPMC implementation [Moffat90]. Table 2.1 shows a history of the development of the main PPM models. It includes different methods for estimating probability information (Methods A, B, C and D), some little modifications and other features related with the order of the model.

Model	Year	Author	Modifications
PPMA PPMB	1984	Cleary J. and Witten I.	Original models
PPMC	1990	Moffat A.	New method for symbol and escape probabilities
PPMD	1993	Howard P.	Similar to PPMC with a slight improvement to probability estimation method
PPMD+	1995	Teahan B.	Some techniques for selecting a particular context to predict the current symbol
PPM*	1995	Cleary J, Teahan W. and Witten I.	Exploits contexts of unbounded length, superior compression to PPMC but consuming considerably greater computational resources
PPMZ	1996	Bloom C.	Uses a local order estimation

Table 2.1 History of PPM class of compression models

There have been a series of improvements over the PPM model, mainly at the probability estimation of the symbols with methods C [Moffat90], D [Howard93b] and PPM* [Teahan95]. PPMC overcomes original methods A and B [Cleary84] by using different ways of predicting the symbols. The PPMD does a small modification to the C method of Moffat and achieves an improvement of 1% over PPMC. The PPM* exploits unbounded length contexts and has shown only a few modifications to the PPM scheme. The PPMZ uses local order estimation to overcome some of the problems of variable order techniques as memory consumption. There are also other methods like P, X and XC [Cleary84] based upon a Poisson process model and perform better than other methods in many cases but have not been given enough attention. From all these methods, the PPMZ is the one that produces the best compression ratios although its space requirements are high [Bloom96c] to be considered for practical implementations, and the PPMC seems to be most suitable option for that.

In [Nelson91], Nelson reviews and includes a software implementation of PPM method coupled with arithmetic coding and discusses how to combine arithmetic coding with several different modelling methods to achieve some impressive compression ratios. Also [Bunton92 and Bunton97] improved the compression performance of any PPM variant, although the improvements are too small relative to their cost to be useful in data compression applications.

In [Effros00], Effros simplified the computational efficiency of PPM by combining it with BWT algorithm [Burrows94 and Nelson96] while using a *prefix tree*, a data structure that according to Bell *et al.* [Bell90] '*is rather complicated and does not appear to be used in practice*'. In [Bloom96c], Bloom made available the software of a PPM model 4th order, and in [Aberg97] Aberg *et al.* present a method for adaptive choice of estimators that provide a simple way of improving PPM.

There have been only software implementations of this model, most of them use tree data structures, either backwards or forwards, binary or with vine pointers, to store the model. But mainly a special type of tree called *trie* is used [Bell90], where the search through the tree is binary and allows fast search operations. Additionally, the branching structure at any level is determined by just part of the data item, not the entire item [Horowitz95]. However, the total number of nodes for a large set of symbols can put them at a disadvantage against other structures. Morimoto *et al.* developed a new double-trie structure that in future software compression applications may speed up the search process [Morimoto94].

According with the documentation of PPMZ software implementation, it compresses at a speed roughly 1 byte per 20,000 CPU cycles with memory requirements about 30 times the file size.

A software implementation of the PPM* model by Itagaki and Yokoo exploits contexts of unlimited length and the space requirements are linear in the string length without depending on the context order [Itagaki00]. It is worth mentioning that, normally, PPM models have an exponential growth of memory as the order of the model increases [Cleary93]. To implement the scheme, Itagaki and Yokoo use a *prefix*

list, dynamic data structure, maintaining a set of contexts in reverse lexicographical order. They claim that although the nodes of the list do not contain statistical information and the structure needs to be traversed whenever statistics are required, the method encodes a string in expected linear time.

Other software implementation that saves memory space is presented in [Hirschberg92]. The PPMC model is stored in self-organising lists and hash tables [Knuth97 and Knuth98]. He implements 2nd and 3rd order models with space requirements showed in Table 2.2 while providing compression ratios close to PPMC with considerable savings in space compared with Moffat’s implementation [Bell90 and Hirschberg92]. These data structures allow them to represent context models of any order in any available amount of memory. Additionally the method executes faster than the PPMC and the *compress* utility of Unix.

Implementation	Order of the model	
	2 nd	3 rd
Hirschberg92	45KB	100 KB
Moffat90	Data not available	500 KB *

Table 2.2 Space requirements for PPMC model. * This figure is in [Bell90 and Hirschberg92].

Previous studies have shown that arithmetic coder provides better results when it is coupled with a PPM-type model, and the higher the order of the model the better the compression performance of the system. Unfortunately, it is practically impossible to implement models without limiting the available space, and this is when some trade-offs among compression and space enter into consideration.

It seems that all the research about PPM style compression models has generated only software simulations and no hardware implementations have been developed.

2.4.2.2 Dynamic Markov Modelling

Some authors mention this scheme under a different type of compression rather than the classical compression classification. However, we consider it as statistical Markov-based methodology. This is a state-modelling technique developed by Cormack and Horspool in 1987 [Cormack87]. The main principle of Dynamic Markov Modelling (DMC) is to maintain a finite number of states that provide symbol probability information. The current state information is used to encode the incoming bit. It counts transitions in each state and when a transition becomes popular enough it is 'cloned' under an intuitive criterion.

The original version of the DMC algorithm is binary and does not compress text well [Salomon98]. However, it is possible to extend the algorithm to handle ASCII characters instead of individual bits, besides implementing more complex states, but the model could grow to consume more memory space, than the binary version. Its overall performance does not improve commercial software algorithms, but yields better compression ratios for binary data (machine code executable files, images and sound) for which it was originally developed. As the PPM model, this technique employs arithmetic coder. It produces compression similar to PPM as stated by Bell in [Bell90] and [Bell97] and although it is said [Salomon98] that it compresses faster than PPM, some results in [Bell97] show the opposite.

There seems to be only software implementations of this algorithm. [Tong96] presents a binary version and [Teuhola93] an application for text. The idea of cloning seems to be complicated, if not impossible, to implement in hardware.

2.4.3 Hybrid Methodologies: Statistical and Dictionary-based

Although data compression has been divided into the three classes of compression methods mentioned in Section 2.3, statistical and dictionary-based classes are the most important. For years, these classes have been studied individually. On the one hand, Markov algorithms yield the best compression but are slow, on the other hand, Ziv and

Lempel algorithms perform at higher speed but offer poorer compression. So, it would be desirable to combine the speed of LZ algorithms with the compression power of the Markov algorithms.

Two approaches, LZIP [Bloom96a] and LZRW4 [Williams91b], combine dictionary and statistical schemes in a single compression method. LZIP is a technique that combines the PPM-type context modelling with LZ77 string matching by finding the most recent occurrence of the lately seen context and comparing the symbols following this context with the current input. The length of the match is coded with either arithmetic or Huffman coding if it is bigger than 0; if not, the literal is coded using other method. Four variants of the algorithm were developed using different coders and context sizes.

The LZRW4 algorithm is an attempt to create a hybrid of the high-speed LZ algorithms and the variable-order Markov algorithms. The algorithm was implemented in an experimental program to measure the compression it would yield. It has a competitive benefit: contexts provide extra compression with little impact on speed, although with a high impact on memory since it uses a hash table of 4096 partitions each containing 32 pointers to the input. The idea is to create a group of contexts to parse phrases from each context. Compression proceeds exactly as with LZW except that the most recent symbol transmitted is used to select one of the 256 contexts. The result would be to take the low-order edge of LZW without losing any of its speed [Williams91b].

2.5 EXISTING DATA COMPRESSION IMPLEMENTATIONS

Data compressors have been implemented in both software and hardware. Among these implementations, some are focused on achieving the best compression ratio, others the fastest compression speed, and others in requiring the less possible space to compress. All these characteristics depend on the application.

2.5.1 Software Implementations

Software implementations of compression algorithms have been numerous due to the large number of applications. They are focused mainly on achieving the best possible compression ratios. Generally such implementations use complex and sophisticated data structures that allow them to store large amounts of data or/and retrieve the information, which at the same time make them slow. These implementations may be used in off-line applications as storage of computer data.

Examples of these implementations are the Unix commands *pack*, which is an old command that uses an adaptive Huffman coding, and *compress* derived from LZW [Welch84] method, also called LZC. Another implementation is ARC, a compression/archival/cataloguing program developed in 1980 that immediately became popular among PC users because it offered good compression and the ability to combine several files into one file called *archive*. The *archivers* are self-extracting; they include a small decompressor in the compressed file, so the file becomes a bit longer, but it can decompress itself. It is derived also from LZC program, but this is an application for file archiving. PKArc is an improved version of ARC; it was developed by Katz who founded the PKWare company, which markets the PKZip, PKUnzip, PKLite and PKArc. The PK programs are faster and more general than ARC and also provide for more user control [Salomon98]. Zip and Gzip, the Gnu's zip compressor, ARJ and LZH algorithms use a variation of LZ77 combined with static Huffman.

LZEXE is yet another software compressor. It is freeware and originally written as a special-purpose utility to compress executable files. It is LZ based, and the main attraction of this program is the facility it provides when a single command can be used to decompress and execute the programs compressed with this tool. It works by using a circular queue and a dictionary tree for finding string matches. An auxiliary algorithm based on the Huffman method then encodes the position and size of the match.

The company HiFn provides software compression with the LZS-221 and MPPC algorithms. Both compressors use the LZS algorithm and are offered also in hardware

chips. The LZS-221 compressor is optimised for network applications as routers, remote access servers and firewalls. The average compression ratio for storage applications is 0.5 and for data communications applications 0.33 is more common [Hifnb]. Compression and decompression speeds are 5 MB/s and 6 MB/s respectively. A relatively new company, ICT (Intelligent Compression Technologies), also offers software compression with the UC-Xpress implementation. As they claim, it *'is a high-performance replacement for Zip coders in real-time compression applications where specialised support for image formats or Microsoft Office formats is not needed'*. It is a variant of the BTW algorithm that on average achieves 15% more compression over the Zip method [UC-ICT]. An additional advantage of this algorithm is that it may adjust its operations in response to the available memory. The algorithm used was developed by Schindler [Schindler97] as a variant of the BTW that improves compression speed by limiting the context size and replacing the MTF coder, suggested by Burrows and Wheeler, by a caching model.

Archivers as e-Space [Wang00] are also offered not only for off-line backup, but online. It dynamically compresses the data according to a policy chosen. *'The software automatically chooses the best compression algorithm for each file and compresses the file by an average of 70 percent', 'compression occurs at a rate of 1 to 3 MB/s'* [Bucholtz00]. The files remain online for immediate access and are transparent to the users and applications. Companies like HP and 3M are using this product in their data centres.

These examples of software compression applications show data storage as their main target. Although software compression is migrating from the off-line applications, it is still not fast enough to meet the requirements of the most demanding high-speed data transmissions.

2.5.2 Hardware Implementations

With the development of better compression algorithms and new VLSI technologies, it is possible to integrate adaptive compression algorithms into single VLSI chips. Fast

software implementations do not meet the requirements of real-time data transmission. Hardware implementations of compression algorithms are better suited for this type of applications, where the compression process is performed on the fly.

These implementations generally use simple compression algorithms to produce fast compression engines. In 1993, Holtz foretold that inexpensive hardware compression chips-sets might soon replace slow and inefficient software compression and become a standard utility in most computers and communication networks. The applications would spread into digital image compression (HDVT), teleconferencing, Wide Area Networks (WAN, ISDN), Digital Audio Tapes and brain-like databases, reaching this market, that in that year was worth \$300 million, \$100 billion by the year 2000 [Holtz93]. Although this figure may have not been reached, the applications of data compression have extended to tapes, hard disk drives, solid state storage (flash), file servers, LAN, WAN, wireless, printers and scanners, medical imaging and military imaging.

Table 2.3 shows a summary of some compression chips that have been presented in the research literature. It comprehends dictionary and statistical methods.

From Table 2.3, it can be seen that considering the time of development and the technology used, Ranganathan's implementation of the LZ77 chip is fast. Huffman implementation was developed about the same time but with better technology and is more than twice as slow as the LZ method. This fact is also derived from the complexity and slowness of Huffman methods against LZ ones and pipelining of Ranganathan's LZ. Arithmetic coder is also a complex method but Printz achieved an implementation of arithmetic coder faster than the LZ77. However, it may be doubtful since no decompressor was presented. Recently, the arithmetic coder hardware implementations show compression throughputs slower than those of LZ and Huffman methods that use even poorer technologies. Also, from the Table, it can be seen that Kuang's implementation uses three times the gate count and is seven times slower than the Huffman chip by Liu *et al.* However, it achieves about 12% better compression ratio.

	Arps88	Bunton92	Printz93	Ranganathan93	Liu94	Lee95	Benschop96	Kuang98	Bongjin98	Jou99	Chen99
Method	Binary arithmetic coding Q-coder	LZ78	Arithmetic multialphabet, division free	LZ77	Huffman	LZ77	LZH (LZ and Huffman)	Arithmetic coding division free	LZ parallel	Binary arithmetic coding	LZSS parallel
Technology	1.5 μ m HCMOS	2 μ m CMOS	FPGA	2 μ m CMOS Systolic array 9 processors	0.8 μ m CMOS	0.8 μ m CMOS	1.2 μ m CMOS	0.8 μ m CMOS	1.2 μ m SPDM, 32 processors	1.2 μ m CMOS	0.6 μ m CMOS
Throughput	/2.5 and 1.5 MB/s	20 MB/s	16 MB/s	13 MB/s	5 MB/s	50 MB/s	12.5 MB/s	0.38 MB/s	12.5 MB/s	1.5 MB/s	91 MB/s
Clock speed	20 MHz & 10 MHz			40 MHz	40 MHz	50 MHz	12.5 MHz	25 MHz	100 MHz	50 MHz	91 MHz
Complexity	13 Kgates	Additional RAM and CAM memories	Additional 11.5 K RAM		17.7 Kgates	Additional 2K CAM memory		54 Kgates Additional 4 SRAM 0.25 Kb each	Additional 4.1 KB SRAM	Additional 3.4 KB memory	90 Kgates
Compression ratio					0.5	range 0.66 and 0.28 (ave. 0.5)	about 0.36	0.57	0.5 assumed		

Table 2.3 Main characteristics of some compression hardware implementations in research

Most of the hardware implementations of arithmetic coding that have been commercialised have been mainly for image compressors, which is out of the scope of this thesis. For universal data compression, most of the current available hardware implementations (Table 2.4) are based on LZ class of compression schemes. Such is the case of LZS chips from HiFn Corporation and ALDC from IBM [Craft98]. However, there are other products, such as X-Match [Jones00], that using different and simple algorithms provide the best compression speeds. Also DCP816 uses a different algorithm, a genetic compression algorithm, and provides moderated compression ratios [DCP] although its compression speed is poor, just 210 KB/s.

The figures in Table 2.4 are taken from the documentation provided by the companies, where the compression ratio is the average achieved for many data types.

The compression chips from HiFn, the LZS family, use proprietary variations of LZ algorithm. Hardware solutions include multiple compression dictionaries maintained so that when a certain type of data is detected, the best dictionary can immediately be used without first building it and sending it to the receiver. This is used mainly in ISDN routers, which offers up to 5-to-1 data-compression ratios. Cisco is one of the companies that employ this algorithm in routers.

Company	Hi/fn	IBM	AHA	Loughborough University
Model	9600	ALDC-1-40S-M	AHA3580	X-Match
Method	LZS	ALDC	ALDC	X-Match
Technology	0.5 μ m CMOS migrating to 0.35 μ m CMOS	IBM CMOS 0.8 μ m	0.5 μ m CMOS	0.18 μ m CMOS
Throughput	80 MB/s	40 MB/s	80 MB/s	100 MB/s
Clock speed	40 MHz	40 MHz	80 MHz	25 MHz
Compression ratio	0.5	0.5	0.5	0.51

Table 2.4 Main characteristics of commercial compression hardware implementations

IBM developed some compression chips, the ALDC (Adaptive Lossless Data Compression) series, now discontinued although they still integrate data compression to some of their devices, such as the AS/400 Integrated Hardware Disk Compression. AHA (Advanced Hardware Architectures) distributes versions of the ALDC compression chips. ALDC is a CAM-based (512) variant of the LZ77 compression algorithm. IBM compression chips perform at 20 and 40 MB/s, while AHA's performs up to 80 MB/s.

DCP816 it is a genetic compression algorithm designed explicitly for WAN data communications equipment such as bridges, routers, and point-to-point compressors. The company claims to achieve significantly higher compression ratios, typically 20% to 80% better than LZS. It has a small, custom-designed RISC processor with onboard ROM, and an integrated DRAM controller for the attached dictionary memory [DCP].

In [Jones00], Jones describes the X-Match algorithm, it is the leader compressor engine achieving speeds of 100 MB/s, over twice the speed of other commercial devices. It compresses 4 bytes at a time and permits partial matching where at least 2 out of 4 bytes may match. It also has a growing dictionary that makes the codes of the match location be codified with less bits when the dictionary is small, which is particularly useful at the start of the compression. The dictionary is implemented in a CAM array, which allows it to attain high speeds and be an attractive option for network applications. It is worth mentioning that a recent version of X-Match [Nunez01] by Nunez *et al.* achieves 200 MB/s with compression ratios of 0.58.

2.6 COMPARISON OF DATA COMPRESSION IMPLEMENTATIONS

A wide range of compression algorithms and implementations has been presented. There is not a single algorithm that may be considered the best for any type of data and application.

All compressors are useful for specific types of data or applications. As mentioned, the best compression ratios can be achieved mainly with software implementations that

are used in off-line applications as storage of computer data. Fast compressors are hardware based and are usually employed for real-time applications where the compression is performed on the fly.

Table 2.3 and Table 2.4 show the main characteristics of commercial and prototype lossless data compression chips. It would be difficult to make a fair comparison of these characteristics since they have been developed under different conditions and with different technologies. However, they are good examples of hardware implementations of lossless data compression algorithms.

Two characteristics that data compressor developers use to measure the efficiency of their algorithms and/or implementations are compression ratios and speed. In [Bell90] there is a comparison of several compression algorithms in terms of compression ratio, speed and memory requirements. From this comparison, several important issues are obtained. For example, dictionary-based algorithms are the simplest and thus fastest algorithms, while statistical methods achieve the best compression ratios. Statistical coders may be coupled with any model to estimate the probability of the incoming data, which will dominate the compression performance. Sophisticated models such as finite-context Markov type achieve better compression, but the time consumption for updating the model is enormous. This is why the arithmetic coder has been studied in great detail and implemented in hardware for research and commercial products but has not been attractive to be coupled with Markov models to improve its compression.

As far as we are concerned, the literature shows that statistical models have been implemented only in software and they have been left as a research work. [Bell90] states that these models *'are rather complicated and do not appear to be used in practice'*. Also, it seems that no commercial companies offer compression products based on these systems.

In compression chips, the highest speeds are achieved with simple methods taking advantage of hardware structures that allow fast search operations. Such structures are CAMs used in the fastest chips [Lee95, Jones00 and Nunez01] that deliver up to 50 MB/s, 100 MB/s and 200 MB/s respectively.

Then, the new generation of data compressors could be a combination of statistical models with efficient hardware structures that will keep the system as simple as possible.

2.7 SUMMARY

We have briefly reviewed the background needed to understand this thesis and have reported the current state of data compression together with the technological issues that influence the hardware implementation of compression algorithms.

It has been shown that dictionary methods meet the requirements for practical implementations due to their simplicity and relatively low memory requirements. The number of compression chips that use this type of methods (Table 2.3 and Table 2.4) confirm this fact.

As mentioned in Section 2.6, statistical algorithms such as arithmetic coders coupled with PPM Markov models, achieve better compression than dictionary-based algorithms, although it is recognised that their complexity have prevented their practical use. In [Bell90], this fact is verified when several algorithms were analysed and its complexity was measured, choosing the PPMC as the best performer of the statistical and dictionary-based compressors. It is also mentioned that, with the same amount of memory, the PPMC compresses always much better than other algorithms (dictionary based). With small memory they achieve similar performance, but as more memory is allowed, PPMC is always better.

Also, it has been said that *'in practice, the extra compression obtained by Markov methods is usually not worth the decrease in speed. However, the field is by no means stable and it is possible that faster Markov techniques will appear'* [Williams91]. This statement leads us to think that a hardware implementation of this type of algorithms may be a good solution to make fast Markov methods. PPMC seems to be the best performer algorithm among the Markov statistical compressors and the most suitable for simplification, although in the literature there is not a detailed study that analyses

the factors that may influence its performance. An analysis of this nature should provide the knowledge necessary to simplify it and make it suitable for hardware implementation. This thesis attempts such an analysis.

CHAPTER 3

OVERVIEW OF INVESTIGATIONS

3.1 OBJECTIVES OF THE CHAPTER

This chapter gives an overview of the investigations contained in this thesis. More specifically this chapter includes:

- A brief description of research topics.
- An introduction to experimental investigations outlining the questions to be addressed in each investigation.
- An overview of the methodology followed in the investigations.

3.2 IDENTIFICATION OF RESEARCH TOPICS

Chapter 2 discussed how lossless data compression techniques have been developed to fully exploit capabilities and reduce costs of data transmission and storage systems. We also discussed the speed advantages and the requirements of implementing compression algorithms in hardware, rather than in software, to suit the most demanding applications in data communications. Most of these hardware implementations use LZ compression schemes due to their relative efficiencies in memory and computational complexity [Bell90] while achieving acceptable compression ratios. The compression ratios that these methods and their descendants attain have been overcome by the PPM class of statistical methods; however, there seems to have been relatively little work in the implementation of PPM algorithms in the belief that they are impractical for being too slow and resource hungry.

Some PPM algorithms are the best compression performers. This thesis will look into the simplification of these algorithms for its hardware implementation. However, the time and resources for this task are limited, making possible only to look at certain algorithm of this class. Based on our literature review, PPMC [Moffat90] is the most promising algorithm for hardware simplification and the one that *'has been carefully tuned to improve compression and increase execution speed'* [Bell90] in software, so we adopt it as our research vehicle. The coder coupled with this model has been widely studied, so, we will focus mainly on the *simplification of the PPMC model for its hardware implementation*.

Commercial chips and some of the research compression chips yield the best throughput in hardware technology. We identify some hardware structures that help these chips to achieve such performance additional to simple algorithms. However, the simplification of PPMC algorithm using simple structures has not been studied. We propose an investigation into *the reorganisation of the PPMC algorithm* to identify and select efficient and effective hardware structures.

This involves a study of the interaction among the statistical PPMC compression algorithm, the simplified updating process and the simple data structures, which lead to simple *hardware implementations*, to provide a clear understanding of their relation.

3.3 INTRODUCTION TO INVESTIGATIONS

This section presents a brief introduction to the proposed investigations and the strategies to follow to meet the objectives. We will assess each strategy for its suitability for hardware implementation and its cost-performance tradeoffs.

3.3.1 PPMC Algorithmic Compression Investigation

This section addresses the main characteristics of the PPMC algorithm (model plus coder). Firstly, we review how the model works and how it is coupled with the

arithmetic coder to provide a clear understanding of the system. Later, the main characteristics and their impact in compression performance are studied in detail. An analysis and software simulation of the PPMC algorithm should help to answer some questions arisen in this investigation as:

- Which are the key computational requirements of the PPMC algorithm?
- What other issues may influence the performance of the PPMC?
- What is the impact on compression ratio of these issues?
- Can the complexity of the coder and model be simplified?

The results provide the information about the functionality and operational requirements of the algorithm that may help in the next investigations for the hardware implementation. The next chapter discusses and evaluates this investigation.

3.3.2 Reorganisation of the PPMC Algorithm

This investigation explores how to simplify the PPMC algorithm, including coding operations and the exploration of different strategies for model updating to reduce the complexity of the PPMC algorithm and speed up the compression process.

Among the questions to answer with this investigation are:

- Are there any methods to simplify the PPMC algorithm?
- What are the design issues involved with these methods?
- What is the impact on compression that these methods may have?

The results of this investigation also provide information that helps into the implementation of a statistical compressor in hardware.

3.3.3 Shift Model Implementation and Performance

This investigation aims to understand the tradeoffs between algorithmic characteristics and hardware architectures by modelling in software a statistical data compressor to be implemented later in hardware. Among the questions to answer in this investigation are:

- Is it feasible to implement in hardware statistical compression?
- Which could be the cost of this implementation?
- Which is the performance impact of efficient hardware architectures in compression?

This investigation provides a framework for characterising the performance impact of the statistical algorithm in hardware and is shown in Chapter 6. This understanding is provided by an *algorithm implementation*, that we have called Shift model, taking care of its main functional requirements. Chapter 7 looks into the hardware modelling implementation in SystemC of Shift model.

3.4 TOOLS AND VERIFICATION

This section describes the tools used to carry out the experiments and the method of verification. The experiments comprise software and hardware simulations and a verification stage where the results from software and hardware simulations are cross-checked.

The data sets to use along the simulations are the popular Canterbury [Arnold97] Corpus, Memory and Thesis Data to have a wide range of file types. A detailed description of the files comprised in these sets is in Appendix A.

- *Canterbury Corpus* is a data set introduced in 1997 as an alternative to a previous corpus (Calgary Corpus [Bell90]) for evaluating lossless data compression methods. The corpus consist of 11 files, which range in size from 3K to 1,029K,

from C and LISP source code, html files, technical writings and text files. Table A-1 and Table A-2 list the files in the corpus, their size and their category. The average size of a file is 255,564 symbols (characters) and the total number of symbols is 2,811,210.

- *Memory Data Set* is a selection data files of about 80 MB contained in memory and includes code and data from the SunOS operating system and eight real applications and utility programs. The set contains nine files from the SunOS operating system, Netscape, Emacs, Textedit, Ghostview, Xman, Matlab, Vlabplus and Logsyn, they are listed in Table A-3. For experimentation purposes, we have shortened this data set to 9 MB of data, 1 MB from each file.
- *Thesis Data Set* is a collection of 65 files, which range in size from 3K to 450K, from audio, images, object and text files. This set was obtained from [Gooch96]. Tables A-4, A-5, A-6 and A-7 list the audio, object, image and text files respectively. All the tables show the corresponding files, its category and size. The total number of characters is 8,045,584.

The software simulations are carried out in a PC using Microsoft Visual C++ 6.0 professional. They explore the behaviour of a compression model, firstly analysing only the model and later adapting it to a coder to simulate the whole compression system.

Later, a hardware simulation of the whole system is developed to demonstrate the suitability of the algorithm for hardware implementation. It was developed in SystemC from the Open SystemC Initiative. SystemC is a C++ class library and a methodology to effectively create cycle-accurate models of software algorithms, hardware architecture and interfaces of SoC(System on a Chip) and system-level designs [SystemC00]. It exploits concurrency and allows simulating high-level functional models, in a similar way to VHDL (VHSIC Hardware Description Language) that is the most commonly used language to simulate and model hardware systems. As VHDL, SystemC allows a structured hierarchical design methodology. It is hardware-

oriented with C++ flexibility and allows cycle-accurate modeling and high-speed simulations.

To verify the functionality of the hardware support of the algorithm, the compressed files output cross-checked with the compressed files output from the software simulation.

CHAPTER 4

PPMC ALGORITHMIC INVESTIGATION

4.1 OBJECTIVES OF THE CHAPTER

This chapter looks into the main functional requirements of the PPMC algorithm for data compression. Specifically, the objectives of this chapter are to:

- Provide a detailed understanding of the PPMC compression model and its interaction with the arithmetic coder.
- Identify the computational requirements of the PPMC algorithm.
- Observe the impact these computational requirements have on compression.
- Identify other issues that may affect compression performance.

Statistical compression algorithms have been studied for several years, overcoming compression performance of dictionary-based algorithms. However, not much has been done to integrate them into the set of practical data compression techniques due to their high complexity and slow execution [Hirschberg92]. In this chapter we study the requirements of one of these algorithms, the PPMC, and analyse the feasibility of speeding it up and simplifying its complexity in order to generate a practical statistical algorithm.

To achieve the objectives of this chapter, it seems helpful to start this review by outlining:

- How the PPMC algorithm works (PPMC model and arithmetic coder).
- What are the key computational demands of the algorithm?
- Important design issues when implementing this algorithm.

4.2 THE PPMC MODEL

The PPMC model is the first software implementation of the PPM class of compression algorithms, developed by Moffat in 1990 [Moffat90]. The scheme is based on a system that maintains a dictionary containing a statistical model of the data, assigning probabilities to the symbols and sending these probabilities to an arithmetic coder.

The statistical model in its simplest form counts the number of times each symbol has occurred in the past and assigns a probability of occurrence to the symbols accordingly. A more sophisticated model is *context based*, where not just the frequency of the symbol is used to predict but also the particular sequence of symbols that immediately preceded that symbol. The preceding symbols are called *context* and the number of them is the *order* of the context.

A PPMC model of order o reads a symbol s and considers the previous o symbols as the current context. Then it searches in the dictionary for the symbol s preceded by the context of order o . If the symbol is found, its probability is sent to the coder. If the symbol is not found, the model faces the *zero-frequency problem* [Cleary95 and Witten91] to estimate the probability of a novel event. PPMC deals with this problem by 'escaping' to the next lower order $o-1$ transmitting a 'Escape' code. After that, the process continues until the symbol is found or the model reaches the order 0. If the symbol is not found in order 0, then a final Escape is transmitted and the symbol s is predicted by order '-1', where all symbols have the same probability of occurrence. The model is then updated adding s to the corresponding contexts. Next, s becomes part of the o^{th} order context used to predict the next symbol.

PPMC 'computes' *symbol and escape probabilities* using the method C (from where the model takes its name) with the following formulas:

$$p(s | context) = \frac{f_s}{t + k} \quad \text{and} \quad p(esc | context) = \frac{k}{t + k} \quad (4.1)$$

where $p(s|context)$ is the probability that symbol s will occur given that $context$ has occurred; f_s is the frequency count of symbol s ; k is the number of different symbols seen in the current context, and t is the sum of the frequency counts of all symbols in the current context.

The PPMC algorithm is a version of PPM models that has been '*carefully tuned to improve compression and increase execution speed*' [Bell90]. To achieve this objective, unlike PPMA and PPMB [Cleary84] that take into account and update all context levels when a symbol is predicted, PPMC uses *lazy exclusions* [Bell90] by only taking into account frequency counts in context levels at or above the context in which a symbol was predicted. Then, when updating the model, just these frequency counts are updated.

Table 4.1 shows a 2nd order PPMC model at some stage in the compression process. Table 4.2 shows the same model after symbol ' u ' came in with previous context ' th '. In both tables, an 'empty' context means that there is no context to follow; the counts simply represent the frequency of the symbols. Frequency counts of 0 indicate that the symbol has not been seen in the corresponding context. The 'total' is the sum of the frequency counts. Order -1 is a special case that has and thus predicts all possible symbols of 8 bits with the same probability, so the total in this order is 256.

For example, in English text if the stream ' th ' occurred it is more probable the next symbol would be ' e ' rather than ' u '. Then, according to Table 4.1, and using the formulas in (4.1), if the current context is ' th ' and the incoming symbol were ' u ', the model would search for ' thu '. Since it has not occurred, it escapes from 2nd order with probability:

$$p('esc'|'th') = \frac{6}{(95 + 6)} = 0.059$$

Then, the context ' th ' drops the ' t ' and with the 1st order context ' h ' looks for the symbol. As it is found, symbol ' u ' is predicted by order 1 with probability:

$$p('u'|'h') = \frac{2}{(200 + 7)} = 0.009$$

Later, during the updating process of the model, the frequency counts and the *total* are augmented by 1. When the incoming symbol has not been seen before, it is added to the dictionary and a frequency count of 1 is assigned to it. Table 4.2 shows the model after the updating process, indicating in bold letters the counts updated.

Order	2	1	0	-1	
Context	'th'	'h'	empty	empty	
Symbols	'a'	8	33	226	1
	'e'	51	110	362	1
	'i'	22	24	188	1
	'o'	7	16	248	1
	sp	6	14	781	1
	'.'	1	1	16	1
	'u'	0	2	84	1
total	95	200	1,905		

Table 4.1 Example of the PPMC model *

Order	2	1	0	-1	
Context	'th'	'h'	empty	empty	
Symbols	'a'	8	33	226	1
	'e'	51	110	362	1
	'i'	22	24	188	1
	'o'	7	16	248	1
	sp	6	14	781	1
	'.'	1	1	16	1
	'u'	1	3	84	1
total	96	201	1905		

Table 4.2 Example of the PPMC model, updated after symbol 'u' followed context 'th' *

* The frequency counts were obtained from a piece of English text of the file *alice29.txt*, part of Canterbury Corpus [Arnold97]

Note that from Table 4.2:

$$p('i'|'th') = \frac{22}{(96 + 7)} = 0.213$$

and

$$p('i'|'h') = \frac{24}{(201 + 7)} = 0.115$$

The entropy formula (4.2) quantifies the information content, where E_l is the entropy and p_l is the probability of the l^{th} symbol.

$$E_l = -\log_2 p_l \tag{4.2}$$

According to this formula, symbol 'i' would be codified with 2.22 and 3.11 bits in 2nd and 1st orders respectively:

$$E_{(i|h)} = -\log_2 p_{(i|h)} = -\log_2(0.213) = 2.22$$

and

$$E_{(i|h)} = -\log_2 p_{(i|h)} = -\log_2(0.115) = 3.11$$

Generally and as shown in this specific example, the higher the context the higher the probability of occurrence and the fewer the bits needed to codify the symbol. Thus, the higher the model the better the compression.

Although PPMC considers two different formulas to compute the probability of Escape and other symbols, it is really a matter of implementation. The probability of Escape may be computed when required with the second formula in (4.1), or if Escape can be stored in the dictionary as any other symbol, its probability may be computed using the first formula in (4.1). In this latter case, the Escape frequency count must be k . In both cases the Escape probability must be the same.

In reality, the arithmetic coder uses the probabilities but in the form of cumulative frequencies (provided by the model) to encode the symbols. And any practical implementation of the model considers restrictions of the maximum frequency counts the model can handle. The model avoids overflow of the counts by halving all of them once a certain threshold has been reached. This technique is called *count scaling* and is explained in more detail in [Bell90].

4.3 THE ARITHMETIC CODER

In Chapter 2, the review of the arithmetic coder just included a semi-adaptive model. However, this type of model is not used in real-time compression but an adaptive model is used. This section shows how arithmetic coder works when coupled with the latter model.

In adaptive coding, the frequency information of the current symbol is generated from the occurrence of symbols previously coded. Arithmetic coder continually receives from the model the symbol frequencies together with a total count so that they can be normalised into estimated probabilities. These frequencies must be in cumulative form, as Table 4.3 shows, to simplify operations, otherwise, the direct calculation of such frequencies can be very time consuming. The table illustrates the index, i , of the symbols, the symbols, the frequencies and the cumulative frequencies.

	Frequencies				Cumulative frequencies			
	Context							
S_i	'th'	'h'	empty	emp	'th'	'h'	empty	empty
'a'	8	33	226	1	0	0	0	0
'e'	51	110	362	1	8	33	226	1
'i'	22	24	188	1	59	143	588	2
'o'	7	16	248	1	81	167	776	3
sp	6	14	781	1	88	183	1,024	4
'.'	1	1	16	1	94	197	1,805	5
'u'	1	3	84	1	95	198	1,821	6
total					96	201	1,905	

Table 4.3 Example of the PPMC model with cumulative frequencies

The coder represents a message with an interval of integer numbers between 0 and N where N depends on the precision desired in the implementation. The interval is subdivided in proportion to the specified probabilities. The algorithm for encoding taken from [Howard94] and adapted here for N , works as follows:

1. Start with a 'current' interval [$low, high$) initialised to $[0, N)$,
2. For each event in the file, perform these steps:
 - (a) Subdivide the current interval in proportion to the specified probabilities provided by the model. The size of the symbol's subinterval is proportional to the estimated probability that the symbol will be the next one to occur.
 - (b) Select the subinterval corresponding to the event that actually occurs next, and make it the new current interval.
3. Expand the interval following a normalisation procedure and output bits as soon as they are known.

Arithmetic coder uses the normalisation procedures to avoid the code range narrows such that the top bits of *low* and *high* become the same. Then, '*any high-order bits that are the same are transmitted immediately*' [Bell90]. We have also adapted the normalisation procedure developed by Witten *et al.* [Witten87] and used by Howard and Vitter in [Howard94] to the interval $[0, N)$. This procedure must be added immediately after the selection of the subinterval corresponding to an input event. Thus step 2(b) in the algorithm above is followed by:

2. (c) Repeatedly execute the following steps in sequence until the loop is explicitly halted:
 1. If the new subinterval is not entirely within the intervals $[0, N/2)$, $[N/4, 3N/4)$, or $[N/2, N)$ exit the loop and return.
 2. If the new subinterval lies entirely within $[0, N/2)$, output 0 and any following 1's left over from previous events; then double the size of the subinterval by linearly expanding $[0, N/2)$ to $[0, N)$.
 3. If the new subinterval lies entirely within $[N/2, N)$, output 1 and any following 0's left over from previous events; double the size of the subinterval by linearly expanding $[N/2, N)$ to $[0, N)$.
 4. If the new subinterval lies entirely within $[N/4, 3N/4)$, keep track of this fact for future output by incrementing the follow count; then double the size of the subinterval by linearly expanding $[N/4, 3N/4)$ to $[0, N)$.

The operations performed by arithmetic coder to compute the new interval (subdivision of the current interval and the selection of the corresponding subinterval, steps 2(a) and 2(b) in the encoding algorithm) are as in (4.3), where CF_{s_i} is the cumulative frequency of the i^{th} symbol; CF_{s_0} is the overall cumulative frequency, and $CF_{s_{i-1}}$ is the next cumulative frequency, *i.e.*, of symbol $i-1$.

$$\begin{aligned}
range &= high - low + 1; \\
high &= low + range * \frac{CF_{S_{i-1}}}{CF_{S_0}} - 1; \\
low &= low + range * \frac{CF_{S_i}}{CF_{S_0}};
\end{aligned} \tag{4.3}$$

It has been mentioned in the literature [Howard94] that '*the main usefulness of arithmetic coding is in obtaining maximum compression in conjunction with an adaptive model*'. In the last two sections we have described the PPMC model and arithmetic coder separately, so, in the next section we will show how they work together to obtain this maximum compression.

4.4 THE PPMC ALGORITHM: PPMC MODEL + ARITHMETIC CODER

In this section we describe how the PPMC model and arithmetic coder interact to form a compression system. To do so, this section explains how the formulas of the model (4.1) are related to the ones used by the coder (4.3) with a simple example.

Figure 4.1 shows an example of the PPMC model information and corresponding range in arithmetic coder. Figure 4.1a shows the model storing symbols (S_i), frequencies (f_{S_i}) and cumulative frequencies (CF_{S_i}). Figure 4.1b represents the arithmetic coder subintervals according to the information in the model. Escape (Esc) has been stored as any other symbol, taking as its frequency the number of times that the symbol has escaped and it is the same as the number of different symbols seen in the current context, k , from formulae (4.1). The information for this table is the same as Table 4.3 for the 2nd order context 'th'. The index i has been placed strategically in reverse order to simplify the calculations.

To relate the formulas used by the model (4.1) with our example, we consider:

$$\begin{aligned}
 k &= f(esc) \\
 t &= \sum_{i=k}^k f_{s_i} \\
 T &= t + k = CF_{s_0}
 \end{aligned}
 \quad (4.4)$$

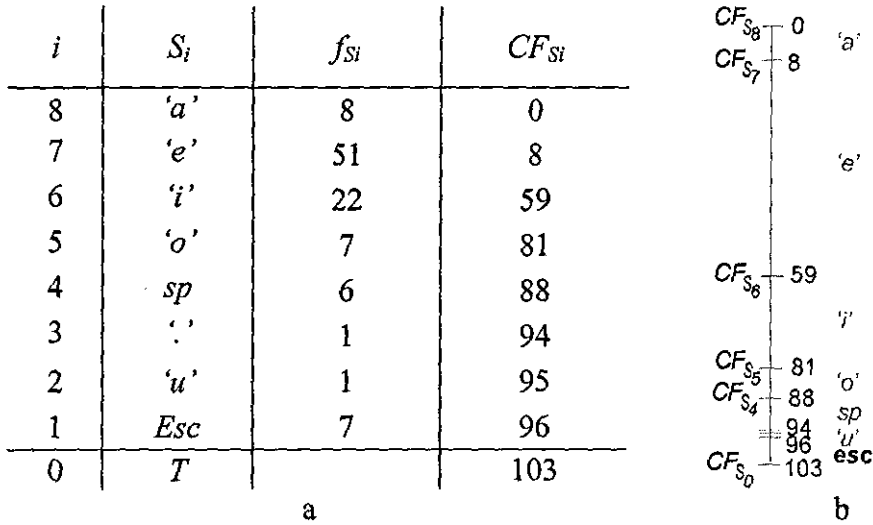


Figure 4.1 Example of the PPMC model information and corresponding range in arithmetic coder

Then, the frequency of any symbol may be computed as the difference of two cumulative frequencies, as shown in formula (4.5).

$$f_{S_i} = CF_{S_{i+1}} - CF_{S_i} \quad (4.5)$$

and the symbol probabilities may be computed as:

$$p(s_i | context) = \frac{CF_{S_{i+1}} - CF_{S_i}}{CF_{S_0}} \quad (4.6)$$

By considering Escape as any other symbol, the two formulas in (4.1) are no longer necessary. The formula in (4.6) is enough to compute both symbol and Escape probabilities, as mentioned in Section 4.2. For example, if the next symbol to occur were 'o', then its probability may be computed as:

$$p('o') = \frac{88 - 81}{103} = 0.067$$

This probability is mapped to the arithmetic coding range as follows:

$$range = 103 - 0 = 103$$

$$high = 0 + 103 * \frac{88}{103} = 88$$

$$low = 0 + 103 * \frac{81}{103} = 81$$

Looking at Figure 4.1b, the values 88 and 81 are the *high* and *low* bounds respectively in the interval for symbol 'o'. After the occurrence of symbol 'o', f_{s_i} increases to 8 and CF_{s_i} and CF_{s_0} increase by 1 and the new current interval in arithmetic coder is [81,88). Then the normalisation procedure is executed and bits are output if required. Thus, the PPMC model must provide the arithmetic coder with CF_{s_i} and $CF_{s_{i-1}}$ as well as the total CF_{s_0} for this to subdivide the range using the formulas (4.3).

4.5 KEY COMPUTATIONAL REQUIREMENTS OF THE PPMC MODEL

Analysing the PPMC algorithm and its interaction with the arithmetic coder helps to identify the main computational requirements and some issues that affect its compression performance.

As compression techniques are used in combination with storage structures, it is helpful to understand how the model is stored to determine the type of operations performed when executing the algorithm. So, we will firstly show the data structure commonly used in software implementations of PPM models.

Later, we review and show the issues and requirements that affect the compression performance. To do so, the algorithm is divided into processes that assist in this analysis to determine the number and type of operations the algorithm requires. After

that, we will use the operations performed in the data structure to describe the processes.

4.5.1 Software Implementation of the PPMC Model

PPM software implementations [Bloom96c, Cleary93, and Howard93b] store data in an efficient tree-class structure, which grows dynamically fulfilling the requirements of data storage. This structure commonly called '*trie*' is a lexicographical tree, particularly useful for processing strings of variable length and specially suited for fast searching operations performed when looking for contexts [Horowitz95]. Its branching structure at any level is determined by just part of the string, not by the entire string. Figure 4.2 shows the structure of a trie in its simplest form. The numbers indicate the size of the arrays, where d is the number of bits required to represent a symbol.

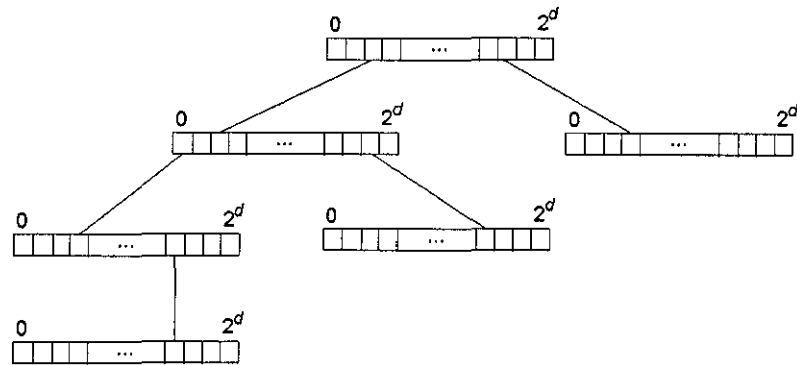


Figure 4.2 Structure of a lexicographical tree

Due to the nature of the algorithm, the strings (context plus symbol) must be stored in reverse order, such that when a symbol is dropped from the context, just a leaf node is ignored. Then, the first level of the tree, level 0, has pointers corresponding to the last symbol of each string. The depth of the trie is the maximum number of symbols allowed in a string minus one.

The structure itself has nothing to do with the compression results, unless its space is limited, allowing the model to gather less statistical information of the data. This fact may lead to poor compression performance.

The operations required to manipulate data within the structure play an important role in the compression performance of the model. The operations we refer to are searching and updating which we describe in the following sections as processes.

4.5.2 Searching Process

This process inputs the current symbol and context so the model looks for them in the dictionary and outputs the symbol cumulative frequencies.

When a trie data structure stores the modelling information of the data, the searching process is done in lexicographical form. To search for a string, the corresponding symbols of the string are taken one by one following the path of the pointers that correspond to them until the information of the string is reached.

Considering the trie in Figure 4.2; in the best case the searching time is $O(l)$ where l is the number of levels in the trie, and in the worst case, it is $O(l!)$. This worst case is when the symbols are predicted in order -1 , so the model looks in the trie for the highest-order context that may predict the symbol, which consumes $O(l)$ time. If the symbol is not found, another search is performed having dropped one symbol from the context. This second search requires $O(l-1)$ time and so on until the symbol is predicted in order -1 .

From this analysis we conclude that as the number of levels is directly related to the order of the model, the higher the order, the slower the search operation.

4.5.3 Updating Process

Updating the model requires adding positions into the dictionary and updating the counts corresponding to the current symbol and context. When the model does not keep counts in cumulative form then they must be computed on the fly, which may be very time consuming, depending on the implementation. As the order of the model

increases this process becomes more complex since in the worst case, when a symbol is predicted in order -1 , the contexts of all orders in the model must be updated. This may be illustrated with the next example.

Table 4.3 showed the model at a certain moment in the compression process. If the letter 'm' were following the context 'th', as it has not occurred before, after the search operation it would be necessary to add the symbol to the model. Then, Table 4.4 reflects with bold letters the counts updated in the model, 'm' is added to the model and because it is a new symbol, its frequency counts are set to 1 in all the orders and cumulative frequencies are also updated.

		Frequencies				Cumulative frequencies			
		Context							
<i>i</i>	<i>S_i</i>	'th'	'h'	empty	empty	'th'	'h'	empty	empty
8	'a'	8	33	226	1	0	0	0	0
7	'e'	51	110	362	1	8	33	226	1
6	'i'	22	24	188	1	59	143	588	2
5	'o'	7	16	248	1	81	167	776	3
4	<i>sp</i>	6	14	781	1	88	183	1,024	4
3	'.'	1	1	16	1	94	197	1,805	5
2	'u'	1	3	84	1	95	198	1,821	6
1	'm'	1	1	1	1	96	201	1,905	7
total						97	202	1,906	

Table 4.4 Example of the PPMC model, updated after symbol 'm' followed context 'th'

The time spent in inserting a new entry in a dictionary depends on the data structure used. A table may require $O(1)$, while a trie may take $O(o)$ where o is the order of the model. Updating cumulative frequency counts in a table may take $O(q)$ where q is the size of the alphabet. However, if efficient structures [Fenwick94 and Fenwick95] are considered the time may be reduced to $O(\log_2 q)$. The type of operations required for updating are mainly add operations.

From this analysis we conclude that the updating process of the model may be as simple or as complex as the data structure that stores the modelling information and the form in which the information is stored. Thus, simple data structures and simple data may lead to simple updating processes.

4.5.4 Arithmetic Coding Operations

Recalling that the basic formulae used by the arithmetic coder (4.3) to subdivide the current interval according to the probability information are:

$$\begin{aligned} range &= high - low + 1; \\ high &= low + range * \frac{CF_{s_{i+1}}}{CF_{s_0}} - 1; \\ low &= low + range * \frac{CF_{s_i}}{CF_{s_0}}; \end{aligned}$$

and that they include adds, multiplications and divisions, it is expected that arithmetic coder executes slow due that the last two operations are the most expensive in terms of time and complexity. These formulae require 5 adds, 2 multiplications and 2 divisions to codify each input symbol, without considering any renormalisation procedure.

The research into the simplification of the coding process has been intense, developing multiplication or division free arithmetic coding versions [Rissanen89, Chevion91 and Lei95]. Some of these alternatives for speeding up the arithmetic coder have included the replacement of multiplications by additions and shifts, ignoring low order bits and replacing arithmetic operations by lookup tables. Naturally, these alternatives introduce errors, which cause the code length [Howard94] increases.

Then, as simpler operations are performed faster, arithmetic coder may benefit considerably by performing simple operations but care must be taken if compression performance is to be maintained.

4.5.5 Other Issues

PPMC software implementations restrict the space to store the model; they also limit the maximum number of bits used for frequency counts. A method for overcoming this problem in the PPMC models is to *scale counts*, a practical issue when implementing

these models referred in the literature, that consist on halving the frequency counts of symbols that share the same context once a threshold has been reached. In addition to solving the problem, this method improves slightly the compression ratio of the model [Bell90]. However, this process requires many operations to halve all the frequency counts in a context. Thus, in the worst case, when all possible symbols are seen in the context, they must halve their frequencies, requiring $O(q)$ operations, where q is the size of the alphabet. This process is not executed per every input symbol, but each time a certain threshold is reached. In models with higher order other than 0^{th} , the number of contexts increases with the order of the model, thus the likelihood of scaling counts also increases.

Another process that requires attention for consuming a considerable amount of time is the *discarding policy*. During the compression process, the tree growth proceeds at full speed while memory is available. Once the memory is exhausted, an efficient strategy for reclaiming space from the *trie* has to be implemented. The software implementation by [Moffat90] discards the entire *trie* when the space allocation has been filled. To avoid inefficient coding at this stage, the model keeps the last 2,048 symbols transmitted and rebuilds the trie from this information.

From here we conclude that other processes such as scaling counts and discarding policies that are irrelevant to the algorithm and required by its implementation may have a big impact in its compression performance due to the number and type of operations they require.

4.5.6 Discussion

‘The main problem in any practical implementation of PPM models is to maintain a data structure while all contexts (orders 0 through o) of every symbol read from the input stream are stored and can be located fast’ [Salomon98]. This statement indicates why the trie data structures have been successful while implementing the PPM models in software: they solve the storing problem while allow fast search operations.

However, data structures suited for software are not necessarily well suited for hardware.

As the order of the model increases, so does the complexity of the model and the processing time, because, as the Table 4.5 shows, the number of possible contexts grows exponentially with the order of the model.

Order of the model	Number of possible contexts
0	256
1	$(256)^2 = 65,536$
2	$(256)^3 = 16,777,216$
3	$(256)^4 = 4,294,947,296$

Table 4.5 Growth of the possible number of tokens as the order of the model increases

Then, the space requirements also increase and even with the current technology it is not possible to store high-order models other than probably 3rd.

Generally, the data structures are not allowed to grow freely when they are implemented; space constraints are always imposed by the capabilities of storage devices. So, it is required to investigate measures to limit the data space and the sensitivity of the models to such bounded spaces.

According to the key computational requirements discussed in this section, we can conclude that the most expensive operations in the PPMC compression system are the searching and updating processes as well as the maintenance of the cumulative frequencies in the model showed in Sections 4.5.2 and 4.5.3. And expensive are the multiplication and division operations in the coder in terms of complexity and compression speed mentioned in Section 4.5.4. Probably other issues related with the implementation are also expensive but it is difficult to know to what extent without practical experiments and this is what next section explores.

4.6 PPMC COMPRESSION PERFORMANCE

The model analysed to identify the key computational requirements is the PPMC model of the literature review [Moffat90] since it is the only document dealing purely with implementation issues of PPM type of models that we are aware of. Some books [Bell90 and Salomon98] briefly explain these key requirements. However, there are other issues that seem to have an impact on compression and have not been considered specifically for this model in the literature, particularly if data structures other than tries or trees are used. So, this section intends to explore the impact of such issues when data structures better suited for hardware implementations are used.

Such issues are the order of the model, the block size of the data to compress, the dictionary size and the discarding policy. To observe to what extent and how these issues affect the performance of the system, we require additional knowledge that may be gained through experimentation. To have a benchmark to compare our results, the PPMC model is simulated and its performance compared against some commercial compression chips based on LZ algorithms.

A common assumption to the experiments of this section is that the compression system consists of model and coder. The model uses a matrix data structure; it is chosen for the simplicity to perform operations in hardware and its closeness to any hardware memory device. The 'perfect' coder computes the number of bits required to codify the probability using the entropy formula (4.2)

$$E_l = -\log_2(p_l)$$

where E_l is measured in bits and p_l is the probability of the symbols.

This coder is chosen to simplify the experimentation process in this early stage. Furthermore, it allows focusing purely on the model. So, for simulation purposes of these experiments, cumulative frequency counts are not required.

4.6.1 PPMC Compression Performance - Experiment

According to the literature review, PPMC performs well compared with commercial chips. However, it is not mentioned how well it performs. To show this and to set a benchmark for further experiments, it is necessary to gain some knowledge from experimentation. To do this experiment as fair as possible, the PPMC model is set to similar circumstances to the commercial chips.

Assumptions

Order of the model:	3 rd order PPMC model as it may be a balance between space requirements and compression performance.
Dictionary size:	4,096 positions to provide the results as fairly as possible to the rest of the algorithms and to guarantee that the dictionary does not fill up considering the chosen block size.
Block size:	Since commercial algorithms allow compressing data per blocks, we compress data with the PPMC model also per blocks. The size chosen is 4 KB for being representative of the block of data found in many data networks.
Data set:	Canterbury Corpus for being a data set collected to evaluate lossless data compression methods while containing a mixture of data types.
Discarding policy:	Not required

The compression systems used in the experiment include dictionary-based hardware implementations, chips from IBM, AHA and Hifn, which use proprietary LZ variants, and X-Match. The results from the commercial chips were obtained executing demo versions provided by the companies.

Method

A simulation of the PPMC algorithm measures the compression ratio under the assumed conditions. Executing the demo software for the corresponding compression chips also measures their compression ratio. The review of the literature for

compression speed of the chips and compression performance of the PPMC model under different circumstances help to conclude this experiment. The compression ratio is measured as the average of the ratio of output bits and input bits per each block of data.

Results

Figure 4.3 shows the compression ratios that PPMC and the commercial chips provide. The X-axis shows the data compressors tested and the Y-axis the compression ratios. From these results, it can be seen that the PPMC model delivers the best compression ratios. The DCLZ, ALDC and LZS chips are better performers than X-Match, they deliver compression ratios about 23% better, but on average 17.5% worse than PPMC.

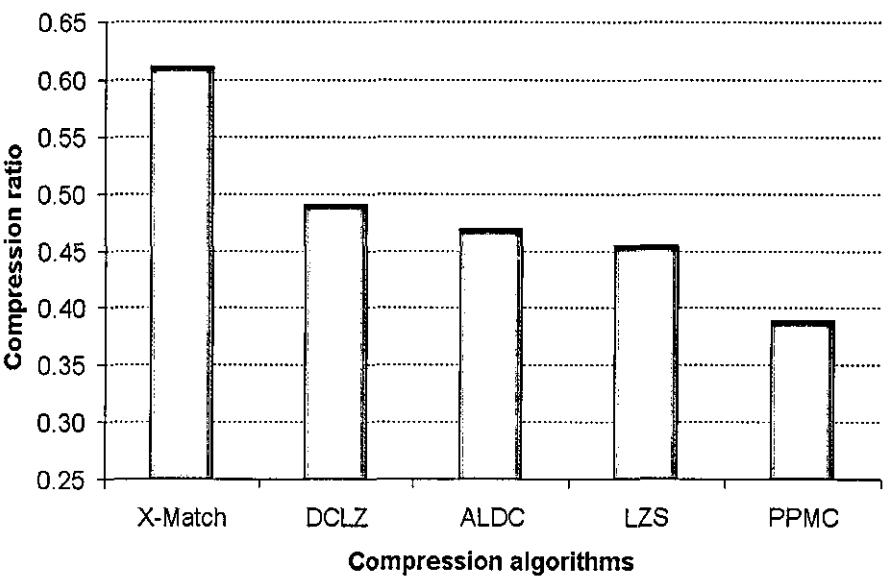


Figure 4.3 Compression comparison of several data compressors and PPMC

From the compression speeds of these chips reviewed in Chapter 2, X-Match is the fastest algorithm and the LZ type is the next fastest. PPMC has not been implemented in hardware so its compression speed can not be fairly compared with these chips.

Evidence [Moffat90] shows compression speeds of 4 KB/s while the fastest of the hardware chips compresses in excess of 100 MB/s.

Compression results of the 3rd order PPMC model from [Moffat90] are about 0.34 for a set of test files, using update exclusion and scaling counts to a maximum precision of about 8 bits. However, Figure 4.3 shows about 0.39 for a different set of files. This clearly indicates that there are some other issues affecting the compression performance; one of them may well be the type of data.

Some other facts in this experiment that may lead to further explorations are:

- The space limitation to store the model, while Moffat uses between 56 KB and 448 KB, our implementation of the model requires about 25 KB.
- Block size of 4,096 symbols are compressed in this experiment while Moffat's software implementation compresses entire files, ranging from 16,384 to 139,521 symbols.

Conclusions

From this experiment we conclude that:

- The PPMC algorithm provides compression ratios superior to other common compression engines.
- PPMC seems to be a promising algorithm that if implemented properly in hardware may reach the compression speeds of those of LZ class, but requires hardware experimentation.
- Space limitations in the dictionary, block size and the type of data, among other issues, clearly affect compression performance of the PPMC algorithm.

Other issues that may impact the compression performance of the PPMC algorithm are considered in the following sections.

4.6.2 Order of the Model - Experiment

In Chapter 2 we mentioned that the higher the order the better the performance of a PPMC model, but it is also true that the compression speed diminishes and the space requirements increase considerably. So it is necessary to find a balance between these requirements and the compression results. The following experiment may achieve this:

Assumptions

Order of the model:	0 th , 1 st , 2 nd and 3 rd order models. Higher order models were not simulated since they do not represent a viable possibility for a hardware implementation due to the high space requirements they may have, see Table 4.5.
Dictionary size:	8,192 positions, equivalent to no space restrictions for the block size used
Block size:	4 KB for being representative of the block of data found in many data networks
Data set:	Canterbury Corpus
Discarding policy:	Least Recently Used for being the most popular policy used

Method

Simulations measuring the compression ratios measured as the average of the ratios of output bits and input bits per block and per file.

Results and Analysis

Table 4.6 shows the compression ratios obtained with PPMC models of different orders.

Order of the model			
0 th	1 st	2 nd	3 rd
0.556	0.467	0.405	0.388

Table 4.6 Performance of different order models

These figures clearly support the fact that the higher the order the better the compression. Although higher-order models were not implemented, there are studies in the literature that reveal that for text there is a little improvement in compression performance with models of order higher than 5 [Cleary93].

Conclusions

- It is true that the higher the order the better the compression, although the improvement in compression decreases as the order of the model increases.
- Higher-order models gather more accurate statistics of the source thus producing better compression ratios. However, the space requirements grow significantly.
- The space requirements are directly related to the order of the model since the number of possible contexts increases exponentially with it. This may well be an issue to consider in a hardware implementation, as the limitations in space are more severe.

4.6.3 Block Size - Experiment

Software compression applications generally compress entire files of data. However, there are certain applications such as networks that require transmitting packets, frames or headers that are a few bytes long, so relatively small blocks of data are compressed. As data compression applications may require compressing blocks of data of different sizes, some commercial compression chips have the option to compress blocks of any size, having a limitation of a few gigabytes. Naturally, compressing blocks of data rather than the entire file may affect compression performance.

To assess the impact of block size on the performance of the PPMC model, and the size of the block that delivers the best compression ratios, further knowledge is required. The following experiment that tests the model compressing blocks of different sizes should provide information to achieve this:

Assumptions

Order of the model:	3 rd order PPMC model
Dictionary size:	Same as the block size to guarantee the dictionary does not fill up considering the chosen block size
Block size:	256, 512, 1,024, 2,048, 4,096, 8,192, 16,384, 32,768, 65,536 bytes. These sizes are chosen as a representative set of the practical block sizes commonly used
Data set:	Canterbury Corpus
Discarding policy:	Not required

Method

A simulation of the PPMC algorithm measures the compression ratio, obtained with different block sizes under the assumed conditions. The compression ratios are measured as the average of the ratios of output bits and input bits per every block and per file.

Results

Figure 4.4 shows the compression results. It can be seen how PPMC delivers poorer compression ratios with smaller blocks of data. Larger block sizes may provide an improvement in compression ratio of about 40% when compared with the smallest block.

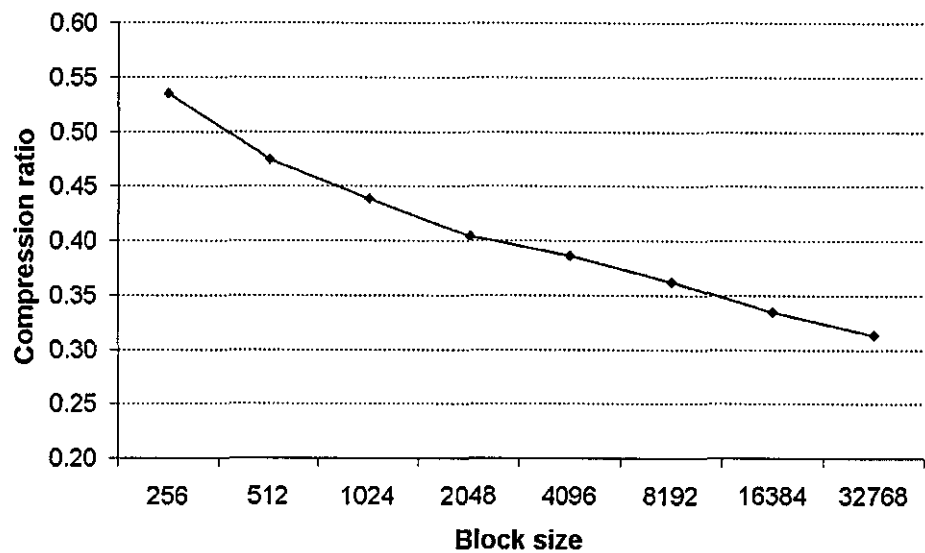


Figure 4.4 Impact of block size on PPMC compression performance

In this graph a different gradient can be seen after 4,096 symbols. This may be explained by the size of the files. Canterbury Corpus has files of different sizes, the smallest being of 3,721 bytes. That means that it is not possible to compress a block of 4,096 bytes or bigger from a file of 3,721 bytes. Thus, the compression ratios of the graph are for blocks of the sizes indicated if the files are at least of the size of the block. If not, the whole file (smaller than the block) is compressed and the result is used to obtain the average compression ratio of this block.

Conclusions

From this experiment we conclude that:

- Block size plays an important role in the compression performance of the PPMC model. The larger the size of the block the better the compression ratio because more accurate statistics of the model are gathered.
- It is expected that large block sizes require more space to store the model than small blocks. So, care must be taken when choosing the block size due to the implications it may have in space requirements or compression results.

Some networking applications have fixed transmission requirements and block sizes can not be changed. However, the results of this experiment serve as reference when designing a model for this type of application.

4.6.4 Dictionary Size - Experiment

When implementing a compression algorithm, there are space limitations that restrict the size of the dictionary. Some compression chips limit the dictionaries to a few thousand bytes [Hifna]. Thus, it seems helpful to assess how, and to what extent, the dictionary size affects compression ratio. This knowledge may be gained with the following experiment that determines the impact that dictionary size has on the compression results, and which size of a dictionary may provide a good balance between compression and space requirements:

Assumptions

Order of the model:	3 rd order PPMC model
Dictionary size:	256, 512, 1,024, 2,048, 4,096, 8,192, 16,384, 32,768 positions
Block size:	32,768 to guarantee this parameter does not influence compression results
Data set:	Canterbury Corpus
Discarding policy:	Least Recently Used

Method

Simulations of the model with different dictionary sizes measure the compression ratios. These ratios are measured as the average of the ratios of output bits and input bits per every block and per file.

Results and Analysis

Figure 4.5 shows the compression results of the PPMC model using different dictionary sizes. From this figure it is clear that the larger the dictionary the better the compression. However, the improvement in compression becomes smaller as the dictionary size increases. For example, it can be seen that the improvement in compression ratio from a dictionary of 8,192 positions to one of 4,096 is not considerable, mainly taking into account that the space requirement duplicates.

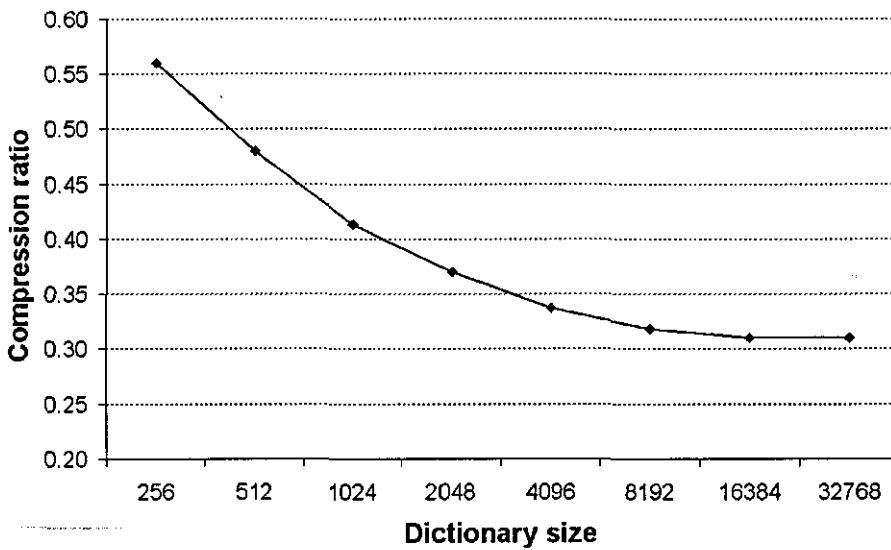


Figure 4.5 Compression ratios for different sets of data using the PPMC model with different dictionary sizes

Conclusions

- The larger the size of the dictionary, the better is the compression ratio, as it gathers more accurate statistics of the data.

- The improvement in compression ratio with larger dictionaries does not justify the increase in space requirements, mainly with larger dictionaries.

4.6.5 Discarding Policy - Experiment

The limitations in space for storing the modelling information imposed by storage devices in practical implementations cause developers to adopt some measures to continue adapting the model to the source of data once the space allocated to the model has been occupied. Such measures, called *reclaiming or discarding policies*, allow the model to reclaim space while continue adaptation.

There are several discarding policies to consider, including *least recently used* (LRU) position [Bell90, Williams93], *least frequently used* (LFU) position, *climb* policy [Williams93], to reset randomly any entry or to reset the entire dictionary. We have analysed them, identifying and summarising their characteristics together with advantages and disadvantages in Table 4.7.

Discarding policy	Functionality	Advantages	Disadvantages
Least Recently Used (LRU)	Removes the 'oldest' entry	Discard less probable symbols/contexts	Requires to maintain sorted the positions in the dictionary that have been used
Least Frequently Used (LFU)	Removes the entry that has been used less times	Do not move/add counts	Requires to maintain sorted the frequency with which the positions in the dictionary have been used
Climb	Moves only one position to the front	Easy to maintain	Need to 'move' data and frequency counts
Reset randomly	Frees any entry randomly	Do not need to manage the positions to free	Requires generate random numbers Need to update many counts
Reset the entire dictionary	Resets all the information of the dictionary	Very easy to maintain	May harm compression results of the model

Table 4.7 Discarding policies

The information of Table 4.7 together with the following description gives a wider understanding of these policies:

- In LRU policy, when a new phrase is input, it becomes the most recently used, and it is discarded only when it is the oldest one to have appeared in the input [Bell90]. Figure 4.6 shows how a window has to be maintained to keep this policy working.
- LFU policy is similar to LRU, but the window must maintain the least *frequently* used phrases, i.e. counting the number of times each phrase is used. The phrase to be discarded is the one with fewer occurrences.
- Climb policy works by moving up one position the phrase that has just been used, so that the phrase to discard is always the one at the bottom. Figure 4.6 illustrates this policy.

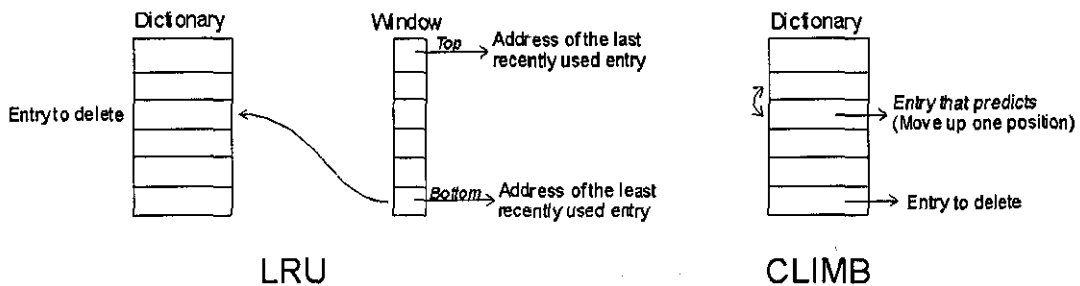


Figure 4.6 LRU and Climb discard policies

- Resetting randomly any phrase requires generating the random positions of the phrases to be discarded. No further window maintenance or counting is required.
- Resetting or discarding the entire dictionary or structure is used as a simple and fast mean of limiting and reusing space. When the space allocated to the dictionary has been filled, the entire model is discarded. This has been demonstrated to be successful in some implementations [Moffat90].

Comments about these policies have been found in the literature, LRU policy *'ensures that the memory available is well utilised, and it adapts well to changes in subject or*

style' [Bell90]. Climb policy can be 'performed in constant time whereas frequency ordering can degenerate to linear time in the number of symbols' [Williams93].

It seems difficult to define which of these alternatives should be used in a practical implementation of the PPMC model without further analysis that considers the data structure that could be used, since the data structure may effect the type and number of operations required in the maintenance of the policy. So, next, this analysis is performed:

Assumptions

- Order of the model: 3rd order PPMC model
- Dictionary size: Small enough to require the reuse of positions
- Block size: Does not apply
- Data set: Does not apply
- Discarding policy: LRU, LFU, Climb, Reset, being analysed

Figure 4.7 shows a diagram of the data structure used in this analysis. The frequency counts for the contexts of order 0 (column D) are 256 indicating that this is the maximum number of symbols in this context order.

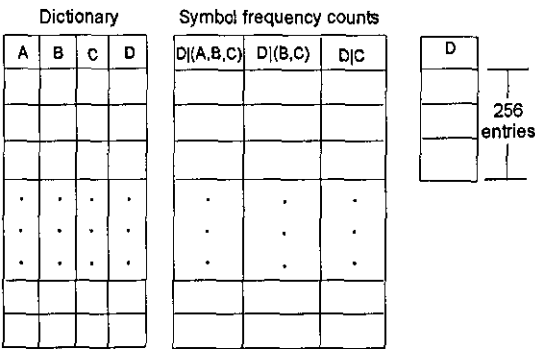


Figure 4.7 Data structure for 3rd order PPMC model

Method

Logical analysis of the discarding policies under the assumed conditions.

Results

Although maintaining a window sorted with the entries in the dictionary that was used, LRU policy seems to be effective. According to the locality of reference principle stating that a symbol that just occurred is more probable to occur in the near future, the symbols pointed by the top positions of the window (see Figure 4.7) are more likely to occur again soon. Thus, deleting the symbols of an entry that has not been used for a long period of time may not harm the compression ratios of the model. Further analysis of this policy indicates:

- When the arithmetic coder is coupled with the model, frequency counts may be stored in cumulative form. When freeing the least recently used entry, cumulative frequencies of all the symbols that share the same context need to be updated.
- When an entry is deleted, it is necessary to identify the positions that share the same context such that after the deletion it is possible to update the cumulative frequency counts. This requires an increasing number of operations to be performed including searching the contexts similar to the context just deleted and updating their frequency counts.
- A further problem is that any entry involves contexts of 3rd, 2nd, 1st and 0th orders, with the entry having different frequency counts in each order. Then each time an entry is deleted, in the worst case most of the cumulative frequencies within the structure must be updated.

Climb policy seems a simple option: it is less complex [Williams93] than LRU and LFU and does not need to maintain any array (window) with the oldest or least frequently used positions. However, it may also cause inaccuracies in the model; any symbol once frequent and has not come for a short period of time, may be deleted. Then when it comes again, the model must treat it as if it has never occurred.

The LFU discarding policy requires a large amount of operations to maintain sorted the positions of entries frequently used. When reusing a position, if cumulative frequency counts are stored, similar contexts to the one to delete need to be updated. This fact may slow down considerably the compression process due to the number of operations to be performed per every input symbol and may prevent the policy from being practical.

A common characteristic of these three strategies is that their complexity not just rests in the maintenance of the window or the moving up of the data, but in the modification of the frequencies and cumulative frequencies associated with the entries being moved or updated.

Resetting random positions in the dictionary on the one hand can lead to inaccurate statistics of the symbols in the model since the more frequent symbols may be deleted. On the other hand, although it seems very simple to maintain, it is not. When a position is deleted, similar context must be identified and updated, what is complex.

Resetting the entire dictionary seems the fastest and easiest policy but also the one that affects more the compression results, as mentioned in [Moffat90] in his statement '*to avoid very inefficient coding while trie was being rebuilt*'. It may bring severe imprecision to the model since all the information that has been gathered about the source of data is deleted and the model must start rebuilding the statistical information again. However, this is probably the most simple strategy and the less expensive to implement in terms of space and time requirements.

Conclusions

From this analysis we can conclude that:

- The complexity of implementation of discarding policies as LFU or LRU in a single dictionary seems to outweigh any benefit of additional compression.
- The higher the order of the model, the more complex to maintain the policy if just one dictionary contains context of all orders.
- The number of operations required would be significantly diminished by keeping contexts of the same order in separated dictionaries due to the simplification in cumulative frequency counts updating of all other contexts involved with each entry. Having separated dictionaries would guarantee contexts of a single order to be updated at a time.

This latter solution has the advantage of resetting just part of the model, freeing space and allowing the adaptation of the model to continue. The cost could be some degradation in compression.

If even this arrangement is considered to be complex or expensive in terms of the number of operations, there is a further consideration. As different dictionaries are maintained, one complete dictionary may be reset when one of them has consumed its space allocated, leaving other dictionaries with statistical information of the model.

4.6.6 Summary

This chapter has provided a detailed understanding of the PPMC model and has shown how the model interacts with the arithmetic coder giving an overview of the entire compression system. We have learnt that PPMC is a complex algorithm to be implemented in hardware due to the number and complexity of the operations required to execute and maintain it. A closer look of the algorithm has helped us to identify the

main computational requirements of the model and coder, as well as other issues that impact on the compression performance.

We have learnt that the searching and updating processes, the data structure utilised and the coding operations, may impact on compression, but it is difficult to know to what extent without further experimentation. Other issues (size of the block being compressed, order of the model, dictionary sizes and discarding policies), that are not directly linked to the algorithm but to the implementation or external issues, also effect compression, our experiments have given a general understanding of them.

Thus, further experimentation to reduce the complexity of the model is required and this is what the next chapter explores. Efficient hardware structures to store the dictionary, simpler coding and faster searching and updating processes may be the key to a simple hardware implementation of the PPMC model.

CHAPTER 5

REORGANISATION OF THE PPMC ALGORITHM

5.1 OBJECTIVES OF THE CHAPTER

This chapter investigates the reorganisation of the PPMC algorithm, including model and coder. The arithmetic coder is simplified and to adapt the model to that simplification, different methods for model updating are explored. This is to reduce the complexity of the PPMC algorithm and speed up the compression process. Specifically, the objectives of this chapter are to:

- Investigate how the arithmetic coder could be simplified and what could be the effects on the model.
- Investigate methods capable of fast and efficient model management for PPMC using simple hardware structures.
- Identify the performance impact these methods have on compression.
- Detect key design issues of the PPMC model.

5.2 ALGORITHMIC REDESIGN

Recalling that the entire compression system (PPMC model plus arithmetic coder) is complex compared with other compressors, attempts to simplify the arithmetic coding operations have included the substitution of multiplication and divide operations for add/subtractions that are simpler and may be performed faster. Some authors [Moffat94 and Witten87] have suggested scaling frequency counts up to a power of two, so that divide operations may be substituted by simple shifts. However, there are not explicit implementations of this proposal that mention the operational impact on the model, how the model should behave under these circumstances, what is the

interface to the model, and whether or not an approximation in the model updating could provide further simplifications in the system performance.

In this chapter we explore the reorganisation of the PPMC algorithm, the modification of arithmetic coding operations, shown in the following section, and how, and to what extent, the model could be affected by such modification.

5.3 SIMPLIFICATION OF ARITHMETIC CODING OPERATIONS

In this section we look into the simplification of arithmetic coding operations, recalling from Chapter 4 the basic formulae (4.3) used by the arithmetic coder to subdivide the intervals:

$$\begin{aligned} range &= high - low + 1; \\ high &= low + range * \frac{CF_{s_{i+1}}}{CF_{s_0}} - 1; \\ low &= low + range * \frac{CF_{s_i}}{CF_{s_0}}; \end{aligned}$$

The division operations were identified in Chapter 4 as the most complex. So, if they could be substituted by simpler operations the encoding process could be speeded. These simpler operations, as mentioned in the last section, could be the use of lookup tables or the approximation of multiplications by shifts and adds. Such modifications in the coding operations do not affect the functionality of the model, although the performance of the entire system may be slightly degraded due to the approximations that this implies.

We think that the division operation could be substituted by shifts while the denominator is kept constant and to a power of two. Then, assuming that such a denominator is constant, we will call it 'Fixed Number of Tokens' or *FNT*. So, the basic operations for the arithmetic coder must be:

$$\begin{aligned}
range &= high - low + 1; \\
high &= low + (range * CF_{s_{i-1}}) \gg FNT - 1; \\
low &= low + (range * CF_{s_i}) \gg FNT;
\end{aligned} \tag{5.1}$$

For the arithmetic coder to execute the formulae in (5.1), it is required that the value of *FNT* is known in advance. As the coder requires the probability information to be sent by the model, with this modification, the *FNT* value must also be provided. These modifications in the arithmetic coder require the model to be reorganised as well. A modification of this type brings consequences in the model since the meaning of this denominator has been modified. It no longer indicates the sum of the frequency counts as symbols are seen, but it is now assumed that the value of this sum is always the same.

Thus, in the next section we explore how the PPMC algorithm could be reorganised such that the precision of the model can be kept and the compression performance remains unchanged.

5.4 REORGANISATION OF THE PPMC MODEL

In this section we explore how the PPMC model should behave when the denominator in the arithmetic coder is constant and how it may update frequency counts in the simplest possible way.

We continue to refer to the constant denominator as ‘Fixed Number of Tokens’ (*FNT*) and the frequencies of the symbols are going to be ‘measured’ by *tokens*. For example, when the model updates, it ‘redistributes the *FNT*’ among other symbols. From now on we use this terminology to differentiate this modification from the original basic formulae.

As the main effect in the model comes when updating or distributing this *FNT* value, we will focus on the model updating. We shall consider and study several methods for model updating to obtain a clear understanding of the updating process under the

circumstances mentioned and how the process could be simplified. Among the questions we need to answer are:

- Which is the best value for the total number of tokens, FNT ?
- Which are the possible strategies or methods for model updating?
- Are these methods suitable for practical implementation?
- Which are the best methods in terms of compression ratio, complexity and speed?

There are some considerations and assumptions to be made when studying the different alternatives, as explained in Chapter 4. However, many of these must be put aside while the core of the alternatives is analysed. So, we will make use of the simplest assumptions and data structures in order to be completely focused on the model updating.

The simplest data structure to consider is an array as shown in Figure 5.1, and it is used here just for a 0th order model. In this way, it is easy to visualise the steps to follow when updating frequency counts. The ideal updating procedure is the one that requires the simplest and lower possible number of operations. The 0th order model maintains one extra position for the 'termination' symbol which indicates when there are no more data to compress.

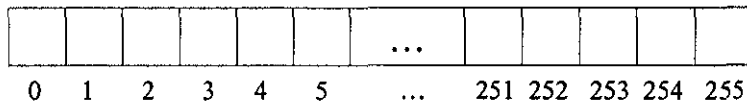


Figure 5.1 Array for frequency counts

One may consider methods that assign the value of FNT to the first symbol to occur, and during the compression process, redistributing the tokens among the symbols such that few and simple operations are required for updating. This is probably the most attractive characteristic of having FNT ; the model updating could be kept simple while the division operation in the arithmetic coder could be avoided.

Thus, the benefits from these methods, if they are successful, are that it could be possible to reorganise the PPMC algorithm by:

- 1) Substituting divide by shift operations in arithmetic coding
- 2) Preserving the compression performance of the model by rearranging its maintenance, e.g. redistributing frequency counts (tokens).

The risks of keeping constant the denominator in arithmetic coding, i.e. using *FNT*, may be that the model could not reflect the statistics of the data or that no approximations for model updated could be found. Then, the only solution should be to have more complex operations in the model.

The methods to consider may include the assignment of *FNT* to the first symbol to occur or to Escape if it is considered as such. Also, the symbols could be kept sorted according to their frequency counts (number of tokens) and so a variety of approximations for the model emerge to be explored. For example, the redistribution of tokens may be dependent or independent of the symbol frequency or the position of the symbol (in case no sorting is done). The possible methods are illustrated in Figure 5.2.

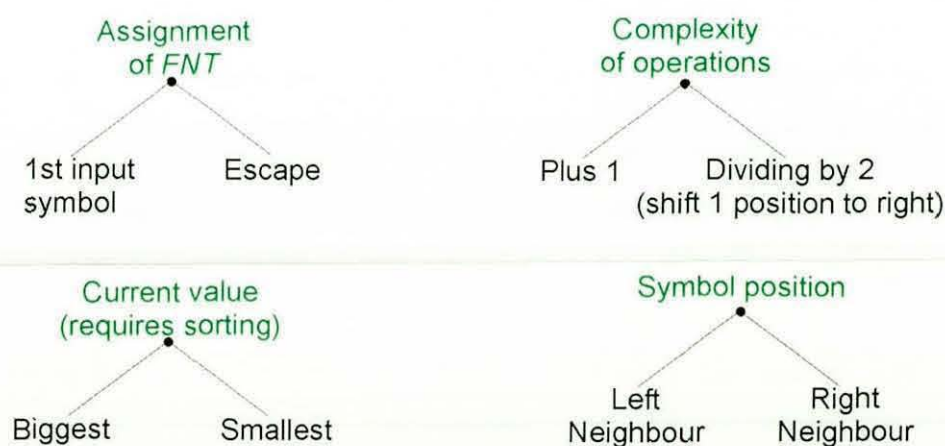


Figure 5.2 Different alternatives for approximating the model updating

There could be a combination of the alternatives shown in Figure 5.2, for example, when the model updates counts it may be adding one token to a symbol (*plus one* strategy) or dividing by two the frequency count (*dividing by two* strategy). Either of them can be combined with a method that takes into account the current value of the symbol frequency or the symbol position as shown in Table 5.1.

	Plus One				Dividing by Two			
	FS	EF	L	LR	FS	EF	L	LR
Biggest	X	X			X	X		
Smallest	X	X			X	X		
Neighbour			X	X			X	X

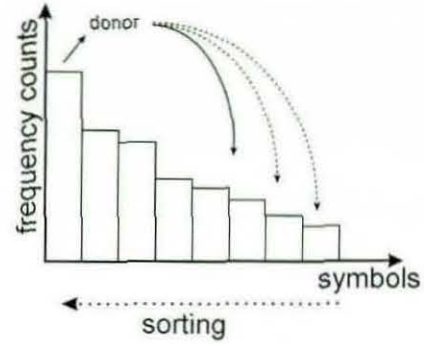
Table 5.1 Methods for model updating

In Table 5.1 *FS* indicates ‘First Symbol’, *EF* stands for ‘Escape First’, *L* for ‘Left’ and *LR* for ‘Left-Right’. The first two refer to the assignment of *FNT* and the last to the symbol position as Figure 5.2 shows. Next in Figure 5.3 we show the combination of methods studied, for each of them both strategies ‘*plus one*’ and ‘*dividing by two*’ apply. Biggest and smallest methods require sorting all symbols except ‘Escape’ that is placed always at the first position of the array when the *EF* option is considered. *DIS* conditions the donor symbol to be ‘Different from Incoming Symbol’.

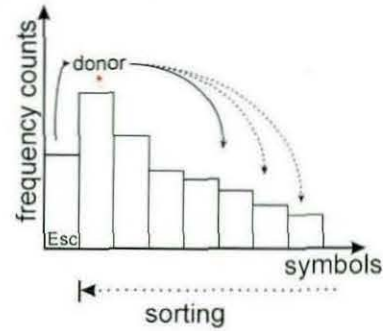
Next we show the algorithms for each method that we investigate. In all of them, a limit in the number of tokens of the donor symbol has been imposed to guarantee the donation or redistribution, e.g. a symbol with one token can not donates tokens, otherwise, the *zero-frequency problem* may arise. The way in which the symbols are redistributed gives the name to the method as it describes the operations required to update the frequency counts. The number of tokens the Escape or new symbol get depends on the strategy chosen (*plus one* or *dividing by two*). After donating tokens, the donor diminishes the number of tokens donated.

Figure 5.3a Biggest FS

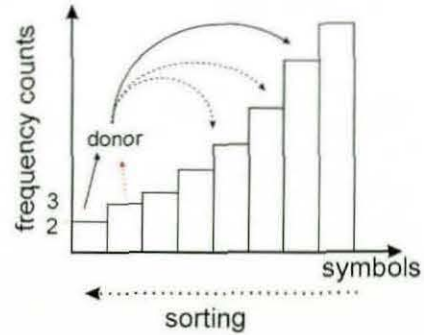
- a) Assign *FNT* to *FS*
- b) As symbols are input:
 - If the symbol is new
 - Escape gets tokens from symbol with Biggest frequency count
 - Else
 - Go to step c)
- c) Biggest symbol donates tokens to the incoming one
- d) Perform sorting procedure
- e) Go to step b)

Figure 5.3b Biggest EF

- a) Assign *FNT* to *EF*
- b) As symbols are input:
 - If freq. count of escape = 1
 - donor = Biggest
 - Else
 - donor = Escape
- Incoming symbol gets tokens from donor
- Perform sorting procedure
- Go to step b)

Figure 5.3c Smallest FS

- a) Assign *FNT* to *FS*
- b) As symbols are input:
 - If symbol is new
 - donor = Smallest ≥ 3
 - Escape gets tokens from donor
 - Else
 - donor = Smallest ≥ 2
- Incoming symbol gets tokens from donor
- Perform sorting procedure
- Go to step b)

Figure 5.3d Smallest EF

- a) Assign *FNT* to *EF*
- b) As symbols are input:
 - If freq. count of Escape = 1
 - donor = Smallest ≥ 2
 - Else
 - donor = Escape
- Incoming symbol gets tokens from donor
- Perform sorting procedure
- Go to step b)

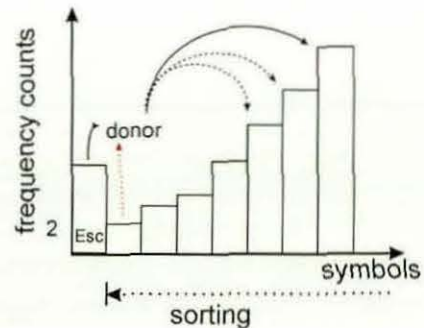
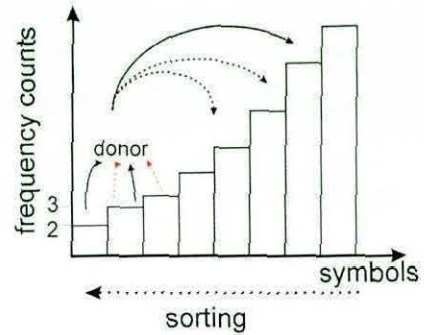
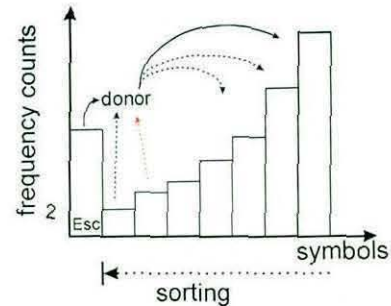


Figure 5.3e Smallest FS DIS

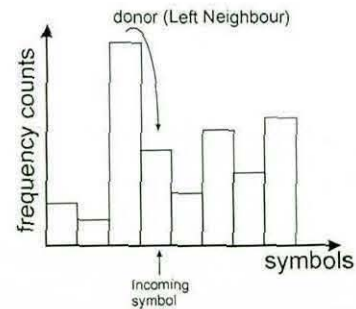
- a) Assign *FNT* to *FS*
- b) As symbols are input:
 - If symbol is new
 - donor = (Smallest ≥ 3) *DIS*
 - Escape gets tokens from donor
 - Else
 - donor = (Smallest ≥ 2) *DIS*
 - Incoming symbol gets tokens from donor
 - Perform sorting procedure
 - Go to step b)

Figure 5.3f Smallest EF DIS

- a) Assign *FNT* to *FS*
- b) As symbols are input:
 - If Escape = 1
 - donor = (Smallest ≥ 2) *DIS*
 - Else
 - donor = Escape
 - Incoming symbol gets tokens from donor
 - Perform sorting procedure
 - Go to step b)

Figure 5.3g L Neighbour

- a) Assign *FNT* to *FS*
- b) As symbols are input:
 - If Left Neighbour ≥ 2
 - donor = Left Neighbour
 - Incoming symbol gets tokens from donor
 - Else
 - There is not redistribution of tokens
 - Go to step b)

Figure 5.3h LR Neighbour

- a) Assign *FNT* to *FS*
- b) As symbols are input:
 - If Left Neighbour ≥ 2
 - donor = Left Neighbour
 - Else
 - If Right Neighbour ≥ 2
 - donor = Right Neighbour
 - Else
 - There is no redistribution of tokens
 - Incoming symbol gets tokens from donor
 - Go to step b)

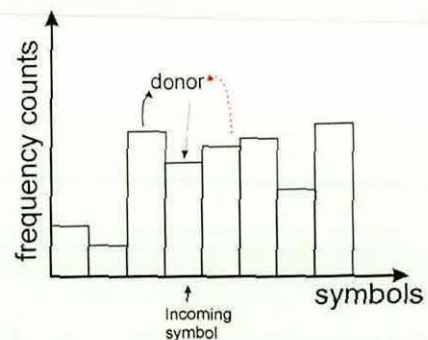


Figure 5.3 Algorithms and examples of methods for model updating in PPMC

All the simulations in this chapter assume that the model is coupled with a ‘perfect’ coder to compute the number of output bits; the coder is in reality the entropy formula explained in section 4.2. Also, the escape symbol (Chapter 4, section 4.2) is considered as any other symbol, so avoiding the need for computing escape frequency counts every time a symbol is not found. Next we show the common assumptions for the methods simulated:

Assumptions

Order of the model:	0 th order model. A low order model allows us to focus on the model updating without involving the order of the model
Dictionary size:	256 positions which is the maximum number for symbols of 8 bits, plus one position for the ‘end of file’ symbol
Block size:	4,096 bytes, this size representing a typical packet size found in many computers and telecommunication systems
Data set:	Canterbury Corpus
Discarding policy:	Not required

As *FNT* is the most important parameter at this moment, it is required to determine its best value, i.e. the value that provides the better compression ratios. So, we now describe in some detail one of our methods and once the value is found other methods consider it.

5.4.1 ‘Biggest *plus one* EF’ Method for Model Updating

Following the algorithm of this method, *FNT* is assigned to the first symbol that occurs, Escape. As symbols come in, the symbol with the biggest frequency count donates tokens to the incoming one. Figure 5.4 shows an example of this method.

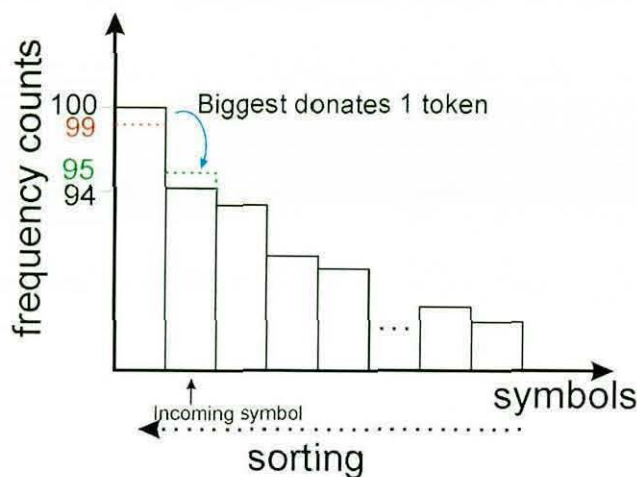


Figure 5.4 Example of 'Biggest *Plus One* EF' method

In the figure, it can be seen how the symbol with the biggest frequency count donates one token to the current or incoming symbol and later they are sorted again. The value of the biggest frequency count was 100 before the incoming symbol arrived, and the next biggest frequency count was 94. As the incoming symbol was the second biggest, the new frequency counts of them are 99 and 95 tokens respectively. This process continues until Escape has just one token left. Then the symbol with the biggest frequency count continues donating tokens. The operations involved are one addition and one subtraction, and later the sorting algorithm is executed. This sorting algorithm will be discussed later.

Assumptions

- Number of tokens: 256, 512, 1,024, 2,048, all of these are powers of two that later on permit the simplification of the divide operations
- Other assumptions have been mentioned before.

Method

The simulation of this method helps us to identify the most appropriate number of tokens, apart from providing the results for this method. The value that helps the model to provide the best compression ratio is later used to simulate the other methods to understand how they perform.

Results

Figure 5.5 shows the compression ratios that the 'Biggest *plus one* EF' method provides with different numbers of tokens.

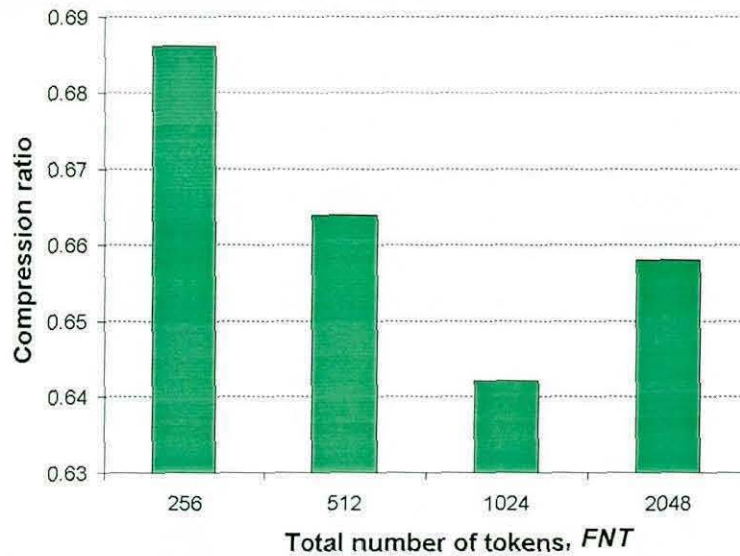


Figure 5.5 Compression ratios obtained with 'Biggest *Plus One* EF' method

From the results shown in Figure 5.5 we observe that:

- 256 tokens do not give flexibility to the model to represent the real statistics of the data. In the worst case, where all symbols occur, just one token could be assigned to each symbol, which means an equal probability distribution for all the symbols, as it occurs in a -1 order model.
- A similar situation occurs with 512 tokens, although not as severe as with 256. An improvement in compression ratio of about 3.2% is observed.
- 1,024 tokens provide compression ratios about 6.5% better than with 256 tokens. This indicates that the model have more flexibility to represent the statistics of the data.

- 2,048 tokens degrade the compression ratio considerably, about 2.4% with respect to 1,024 tokens. This is caused by the high probability assigned to the first symbol and the slowness to reduce it or to share the tokens with other symbols.

Conclusions

It is clear that the value of *FNT* has an impact on compression ratio. On the one hand, big values make some symbols have higher probabilities that do not reflect in reality the occurrence of the symbols in the input stream. On the other hand, small values of *FNT* do not allow the model to make a proper redistribution of tokens and this fact is reflected in the statistics of the data. In both cases the model can not gather appropriately the probability information of the source of data and thus poor compression ratios are obtained. It is best to find a balance between compression and *FNT* value and it is provided by 1,024 tokens, then, for other methods 1,024 tokens will be used.

5.4.2 Other Methods - Experiment

The previous experiment gave an insight of the compression ratios that can be expected from a single method. However, we have a variety of them and they need to be explored and simulated. This is the purpose of this section: to find the compression ratios these methods provide and to compare them with the compression results of PPMC model. This comparison will serve to decide whether or not is worth reorganising the PPMC model in this way.

Assumptions

Number of tokens: 1,024, as this number showed in the previous experiment that it gives enough flexibility for the model to redistribute tokens

Other assumptions are as mentioned before.

Methodology

The simulation of all the methods provides compression results to evaluate their performance. An analysis of the behaviour of the methods will later reflect why such results were obtained. Compression results and the analysis should help to evaluate whether or not these methods are suitable for fast and simple model updating. If the compression results are satisfactory then we may select the method that is closer to our expectations for further experimentation.

Results and analysis

Figure 5.6 shows the compression performance of all the methods simulated. It includes the compression ratios obtained with all the variants of the two strategies mentioned and PPMC to serve as a reference.

From Figure 5.6 we observe that:

- The *plus one* strategy is considerably better than *dividing by two*, except for the first two methods. This is because in general, the *plus one* strategy allows slow adaptation of the model, and this is similar to how it is done in PPMC model, while *dividing by two* provokes abrupt changes to the frequencies of the symbols, which results in imprecise symbol probabilities and thus poorer compression ratios.
- The methods where the smallest symbols donate tokens are the worst performers because sometimes there is not proper redistribution of tokens and in many cases there is not adaptation at all, depending on the method.

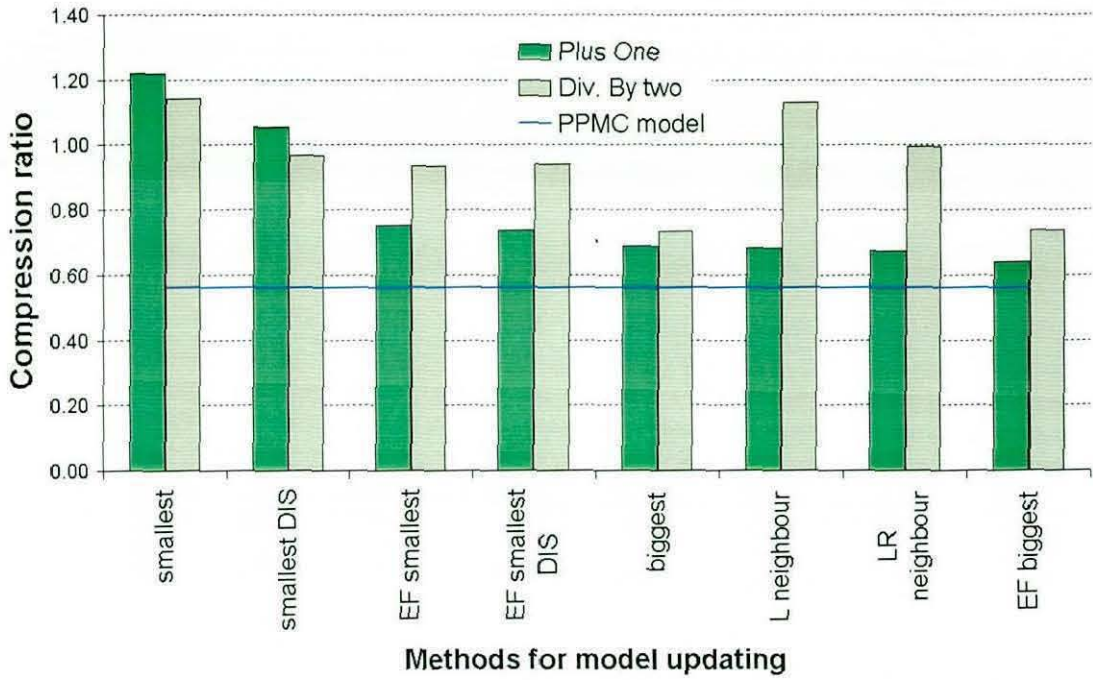


Figure 5.6 Compression ratios obtained with methods to approximate the updating process of the PPMC model

- The two neighbour methods perform similarly to the biggest ones when they follow the *plus one* strategy, not so when they follow *dividing by two*, where the difference about strategies is considerably larger. This is because the speed of adaptation, additional to the fact that neighbour and biggest methods generally do not affect symbol frequencies, allows continually the adaptation of the models.

To compare and analyse in more detail the different methods, the compression process may be divided in two states, shown in Figure 5.7. The first state is transitory where the model is gathering the statistics of the model (1 and 2 in the figure). The second one is the steady state of the process, where the model has learned the statistics of the data: the area marked 3 in the figure.

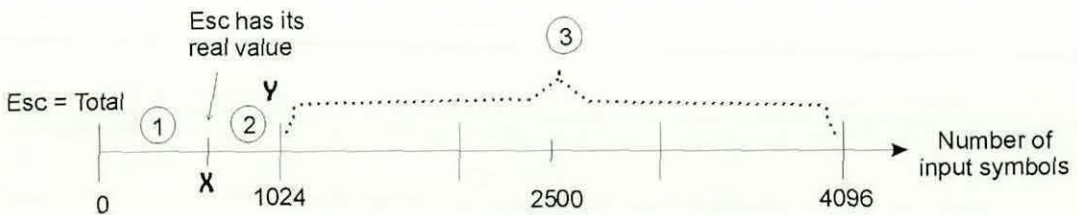


Figure 5.7 Stages of the compression process

Stages of the compression process

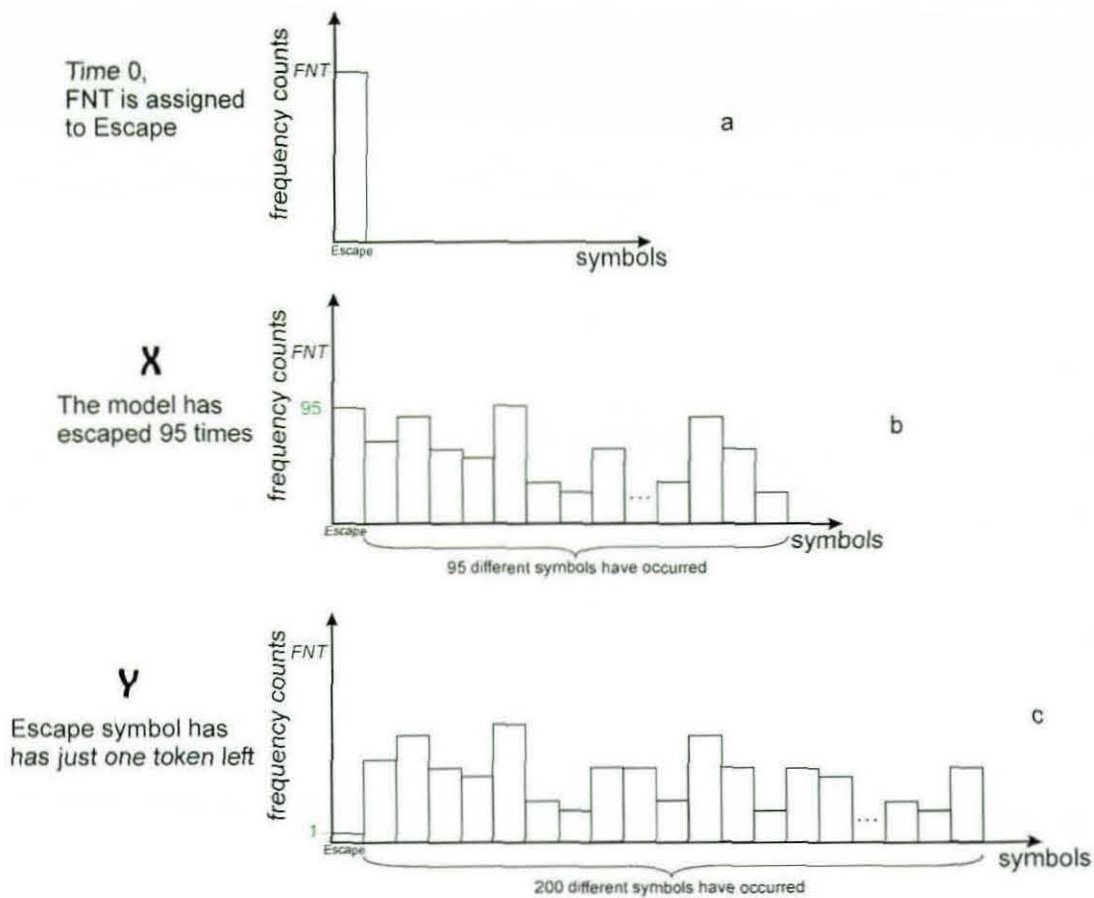


Figure 5.8 Detail of the stages of the compression process

Taking as an example the *escape first methods* (Figure 5.3b, Figure 5.3d, and Figure 5.3f) where *FNT* value is assigned to the Escape symbol at the beginning of the compression process, and looking at the Figure 5.7 and Figure 5.8, we observed that:

- As symbols come in, the escape donates tokens and eventually it reaches its real value at point **X** (the real value is the number of times the model has escaped to a lower order).
- After point **X**, escape continues donating tokens until it has only one token left, at this time it has reached point **Y**.
- When the model is point **Y**, the compression ratio does not suffer if escape has just one token left since at this moment it is expected that most of the possible symbols

have occurred. In steady state, there is a redistribution of tokens among the incoming symbols, where the key symbol (smallest, biggest or neighbour) always donates them.

The first three methods, with the smallest symbol donating tokens, do not compress at all because a symbol with a number of tokens smaller than two can not donate tokens, and the model has not the flexibility to redistribute them. Thus, during the transitory state the model does not learn the statistics of the data correctly.

With the *plus one* strategy, the two ‘biggest’ methods perform well because the symbols are sorted according to the number of tokens they have, and the more frequent symbols are located at the beginning of the array, so tokens can be constantly redistributed. That means that the model adapts as symbols are input.

‘Biggest plus one escape first’ performs closest to PPMC. The biggest symbol has always tokens to donate so there is always adaptation. When escape donates tokens first, the method improves compression ratio about 7%.

In addition to the compression results, the number of operations performed by the methods and their resource requirements must be considered. The simplest strategies are the neighbour ones, as they do not require sorting while ‘smallest EF DIS’ method is the more complex due to the number of operations it requires for every symbol, e.g. sorting, but most importantly the operations to choose the donor symbol.

Conclusion

- One of the goals of model reorganisation is to find simple methods that allow accurate modelling. However, it was seen that not all the alternatives are simple, neither do they generate acceptable compression ratios. So, additionally to the compression ratios generated, complexity is also to be considered, and among the methods, ‘biggest plus one EF’ is the alternative that represents the best balance.

- When the model is flexible enough to distribute the tokens, the *dividing by two* strategy is better in a transitory state because it quickly adapts the model to the statistics of the data. Then, in the steady state it is required that the model adapts slower, thus making the *plus one* strategy better for this state.

As these two strategies provide an appropriate adaptation, each one for each state of the compression process, it may seem natural to think that a combination of them may offer better results. This is what the next experiment shows.

5.4.3 Combining Strategies – a New Method

The previous experiment showed that *plus one* strategy gives a slow adaptation to the model, while *dividing by two* is faster to adapt. So, as the analysis of the stages in the compression processes showed, it is desired a fast adaptation in transitory state and a slow adaptation in steady state. This experiment shows results of a combination of strategies to find out if this new method can provide closer compression ratios to PPMC model. Also we test a different combination that would adapt faster than *plus one* but slower than *dividing by two* in transitory state and later in steady state *plus five* is considered.

Assumptions

Order of the model:	0 th order model. A low order model allows us to focus on the model updating without involving the order of the model
Dictionary size:	256 positions which is the maximum number for symbols of 8 bits
Block size:	4,096 bytes, this size representing a typical packet size found in many computers and telecommunication systems
Number of tokens	1,024
Data set:	Canterbury Corpus
Discarding policy:	Not required

Methodology

The method to consider is the ‘Biggest *EF*’ using both strategies, *dividing by two* in transitory state and *plus one* in steady state and *plus eight* and *plus five* respectively. That means that the redistribution of tokens while Escape has not finished with the *FNT* tokens is done dividing by two or diminishing by eight its frequency count. Once Escape has one token left, the symbol with the biggest frequency count donates one or five tokens every time it is required. This simulation must provide compression results to evaluate the performance of this combination of strategies.

Results

Figure 5.9 shows the compression ratios obtained with the two ‘Combined strategies’ described above. The ‘Biggest *plus one, EF*’ is also shown as it was the best method that we obtained with the previous experiment and it serves as a reference along with the PPMC model.

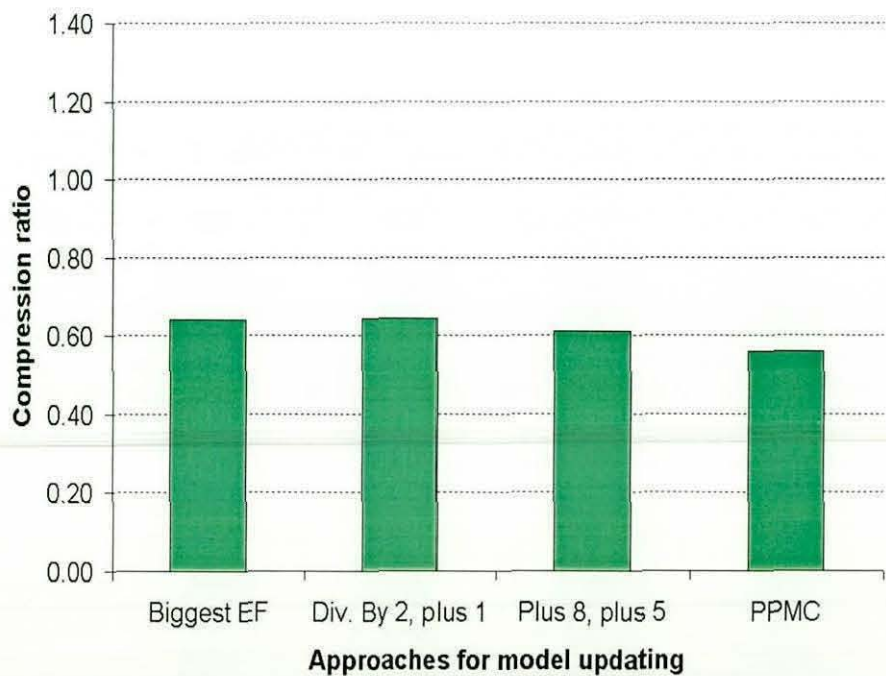


Figure 5.9 Compression ratio on combined strategies

In the figure, the Y-axis shows the compression ratio of the methods, keeping the scale used for Figure 5.6 to appreciate the difference in compression as with the other methods. There is a slight degradation with the first combination of strategies of about 1.04% with respect to 'Biggest *plus one* EF' method. Also there is a slight improvement in compression with the second combination of about 3.94% with respect to 'Biggest *plus one* EF' method. However, it still is 8.75% worst than the PPMC model.

Conclusion

Any improvement in compression ratio helps to close the gap between the methods and PPMC model, however, we consider that a method with a maximum degradation of about 5% is good enough to be taken into account for further investigation. Unfortunately it was not the case of any of our methods.

5.4.4 Effect of Sorting - Experiment

We have mentioned that it is better for the methods to keep symbols sorted according to the number of tokens they have (Section 5.2). There are several sorting algorithms, its selection depends on factors such as the number of symbols involved, the knowledge of the orderliness of the symbols, or the cost of comparing vs. the cost of moving symbols, etc. We are interested in the fastest algorithm that is probably *quicksort* [Knuth97] where several groups of comparisons are required.

We thought about other algorithm that sorts partially the symbols. It compares frequency counts of the symbols in alternate form, e.g. compare symbols 1-2, 3-4, etc. and the next time compare 2-3, 4-5, etc. In this way, when implemented in hardware the comparisons may be done in parallel and thus this 'partial' sorting algorithm would require $O(1)$ time.

Table 5.2 shows a comparison of the complexity of perfect and partial sorting, considering the quicksort algorithm for the perfect sorting. It is worth noting that the figure of the complexity of the quicksort algorithm is the one of its software implementation, which is purely sequential. However, a hardware implementation may be able to add some degree of parallelism at the expense of hardware complexity.

	Perfect (Quicksort)	Partial (Parallel)
Number of comparators	1	$n - 1$
Number of comparisons	$O(n \log_2(n))$	$n/2$
Running time	$O(n \log_2(n))$	$O(1)$
Other operations	Adds and subtracts to maintain indexes and compute the algorithm	Just permutations

Table 5.2 Complexity of perfect and partial sorting

It is difficult to know if the compression ratio of the PPMC model is affected by using partial sorting without experimentation. So, next, we experiment and compare compression results with both types of sorting.

Assumptions

Order of the model:	0 th order model for being the simplest form of evaluating this experiment.
Dictionary size:	257 positions, 256 for the symbols plus the 'end of file' condition
Block size:	4,096 bytes, this size representing a typical packet size found in many computers and telecommunication systems
Number of tokens:	1,024, as this number showed in the previous experiment that it gives flexibility for the model to distribute tokens
Data set:	Canterbury Corpus
Discarding policy:	Not required

Method

The simulation of both sorting strategies, perfect and partial, must provide the knowledge to evaluate the impact of these strategies on compression. The only strategy tested is 'biggest *plus one EF*' as similar results are expected for the other strategies.

Results

Compression results are showed in Figure 5.10.

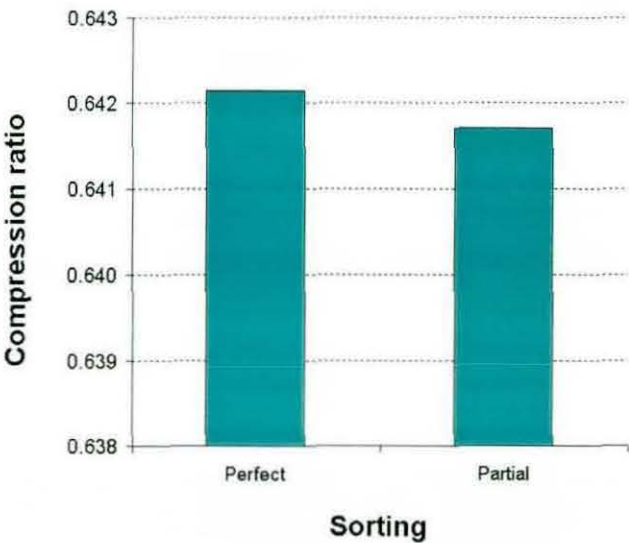


Figure 5.10 Perfect and partial sorting in 'Biggest plus one EF' method

Sorting the symbols may require a great number of operations to be performed by the model, which may slow down substantially the compression process and make it impractical for implementations. However, with partial sorting the time is not a problem and as Figure 5.10 shows, neither it is the compression ratio. It shows how partial sorting achieves about 0.06% better compression ratio and it is an acceptable sorting method that we may use.

Conclusions

From this experiment we conclude that:

- Considerably compression speed can be gained by partially sorting the symbols rather than using a perfect sorting algorithm.
- The cost of implementing a partial sorting strategy is the increase in hardware logic compared with the perfect sorting.

5.4.5 Conclusions

The statistical nature of the PPMC model requires a careful study when looking for an approximation to the model updating process. Experimental results have shown significant degradation of the compression ratio due to small changes in the updating procedure of the model.

From all these strategies and methods we have learnt that the updating process may be simplified considerably at the cost of some degradation in compression ratio. Results of the experiments showed that the degradation was about 15% for the best case, ‘biggest *plus one EF*’ (Section 5.4.2), compared with the PPMC model and a combination of the strategies is still 8.75% worse (Section 5.4.3). Most of these strategies do not assign symbol probabilities according to the method C and that is why they do not provide compression ratios similar to PPMC algorithm.

A general review of our strategies and the functionality of PPMC lead us to understand that there is no form of approximating the model updating strategy close enough to the PPMC model unless the probabilities of the symbols are assigned proportionally to the symbol occurrence. Then, a strategy that, having a fixed number of tokens, redistributes them in such way that follows the behaviour of the PPMC model, may be more accurate. In the next chapter we explore this strategy.

5.5 SUMMARY

In this chapter we looked into the reorganisation of the PPMC algorithm, including the simplification of arithmetic coding and the implications in the model. Also the exploration of some methods that substitute complex operations used commonly by the PPMC compression system. The researched methods are capable of fast model updating for the PPMC model that may also help to speed up the compression process.

We identified the performance impact that these methods have on compression, from which we deduced that a slight modification in the updating process may bring significant degradation in compression ratio if the model is not studied carefully. Also we have detected some of the key design issues in the implementation of the PPMC model.

We have learnt that if the proposed modifications in the arithmetic coder are done as in Section 5.3, then, the coding process must perform faster. However, to ensure that the PPMC model continues providing accurate information, further study in the model is required to guarantee proportional frequency counts to the PPMC model.

CHAPTER 6

SHIFT ALGORITHM

6.1 REVIEW OF OBJECTIVES

This chapter explores the interaction between software algorithms and efficient hardware structures, looking into the implementation of a statistical model. More specifically, the objectives of the chapter are to:

- Give an overview on the issues involved in optimising the PPMC model for its hardware implementation.
- Show the integration of these issues in a single algorithm and observe its performance.

This investigation will address the following questions:

- Is the model suitable for hardware implementation?
- How the model performs against PPMC software implementation?
- What are the main design issues of this algorithm?
- How these design issues influence compression performance?

6.2 OPTIMISATION OF THE PPMC MODEL FOR HARDWARE IMPLEMENTATION

In Chapter 5, the simplification of the arithmetic coder was studied. In this chapter we will focus mainly on the model, keeping in mind the interaction between the model and coder. Chapters 4 and 5 analysed issues related to the PPMC compression algorithm. The knowledge gained from these chapters is applied here to design a

PPMC algorithm, focusing on the model, to be implemented in hardware. From now on we will call our method the Shift algorithm to distinguish it from the PPMC algorithm.

6.2.1 Order of the Model - Experiment

The order of the model was studied in Chapter 4. We learnt that a higher order model provides more accurate statistics, therefore the better the model is. However, its space requirements are directly related to the order of the model, so, a good balance between space and compression is required to make the model suitable for practical implementation.

To select the order of the model we take into account that there is a gain of about 16% in compression ratio when using the 2nd order model rather than the 1st one and about 6% when using the 3rd rather than the 2nd (Chapter 4, section 4.6.2). We may predict that a 4th order model would provide a further improvement in compression. However, it may be not worth given the increase in space requirements. Thus, a 3rd order model is a good option. However we can simplify the experiments by using a 2nd order model when necessary.

As part of a study of the order of the model, we considered the percentage of predictions made by each context order within the model. Also, the experiment involves the use of the Memory Data Set (Appendix A) to observe whether or not the type of data effect these results. This information can be used later to take some decisions for the design.

This experiment helps to identify how much weight each context order has in the model and if the type of data influence that result. This fact may help later on to determine the size of the dictionary.

Assumptions

Order of the model:	1 st , 2 nd and 3 rd order PPMC model
Dictionary size:	2,048 positions for 3 rd , 2 nd and 1 st order contexts 257 and 256 positions for 0 th and -1 st orders respectively
Block size:	4,096 bytes, due to this size representing a typical packet size found in many computers and telecommunication systems
Data set:	Canterbury Corpus and Memory Data to observe if the percentage of predictions in each order context could be also related to the type of data
Discarding policy:	LRU

Method

This involves the simulation of the models under the assumed conditions and monitoring the order of the context where the predictions are made. The results will be obtained as the percentages of predictions made by each context order. For example, a 3rd order model has contexts of 3rd, 2nd, 1st, 0th and -1st order. So, we measure how many of the symbols are predicted in each context order.

Results and Discussion

The percentage of predictions for Canterbury Corpus is showed in Table 6.1. The first row shows results for a 3rd order model and indicates the percentage of the order of the contexts that compose it. The second and third rows show results for 2nd and 1st order models respectively.

Order of the model	Order of the contexts				
	3 rd	2 nd	1 st	0 th	-1 st
3 rd	58.75	13.34	16.12	8.74	3.05
2 nd		74.12	15.04	7.91	2.93
1 st			89.32	7.80	2.88

Table 6.1 Percentage predictions for each order of the model with Canterbury Corpus

More than 50% of the predictions are made by the 3rd order contexts and surprisingly, more symbols are predicted by the 1st order contexts than by the 2nd order ones. Table 6.2 shows the percentage of predictions for Memory Data. It can be seen that the predictions made by 0th order contexts, in any order of the model, is considerably higher.

Order of the model	Order of the contexts				
	3 rd	2 nd	1 st	0 th	-1 st
3 rd	48.35	11.14	15.66	20.12	4.74
2 nd		60.70	15.61	19.42	4.26
1 st			77.15	18.94	3.92

Table 6.2 Percentage of predictions for each order of the model with Memory Data

By comparing the previous two tables, it may be observed that the percentage of symbols predicted by the contexts of different orders varies according to the data type. This is because more redundant data has strings of symbols that are expected to occur more often and thus the highest order contexts would predict more symbols than with a less redundant type of data, where the number of predictions is distributed among the different context orders.

Conclusions

- The results in this experiment show how, as the order of the model increases, the percentage of predictions given in the highest order context is redistributed among the two highest order contexts in the next higher order model.
- The data type being compressed influences also the percentage of symbols predicted in each contexts and thus compression ratios.
- The statistics of the model gathered by the highest order contexts are more accurate; for this reason, although in higher order models the percentage of predictions in the highest context seems to diminish, the compression ratios improve.

6.2.2 Dictionary Size - Experiment

Practical implementations of any data structure have a limitation in size, particularly taking into account that storing space is expensive in digital technology. Furthermore, space restrictions may guarantee that the system fits in a digital device as FPGA or ASIC. Naturally, severe restrictions in space lead to compression degradation and care must be taken to identify a good trade-off between space and compression.

From Chapter 4 we learnt that the larger the dictionary the better the compression. However, this knowledge is not sufficient to decide the size of the dictionary to be implemented, which guarantees a good trade-off between space requirements and compression performance.

The decision about the size of the dictionary may be linked to parameters such as the block size or the order of the model; but in order to study one of the parameters, the others must remain fixed. However, one of the parameters that we considered important in this experiment is the type of data (Appendix A), as it would effect the size of the dictionaries. So, next, we show the assumptions of this experiment.

Assumptions

Order of the model:	2 nd order PPMC model
Dictionary size:	Without space restrictions to avoid this parameter effecting compression and to effectively verify the number of positions used
Block size:	4,096 bytes, due to this size representing a typical packet size found in many computers and telecommunication systems
Data set:	Memory Data, Thesis Data and Canterbury Corpus as different types of data may require different dictionary sizes
Discarding policy:	Not required

Method

As a measure to observe the appropriate dictionary size, we carried out an experiment to determine the number of locations required for the model when compressing different types of data. The results are the average number of positions used by all blocks compressed in the entire set of files.

Results

Table 6.3 shows how for almost all types of data, the average number of positions used in the dictionary of order 1 is one third of the one used in order 2. This means that if contexts of different order could be stored in separated dictionaries, the dictionary storing contexts of order 1 could be three times smaller than the one of order 2 and this could represent great savings in storage space.

Dictionary Order	Canterbury Corpus		Memory Data		Thesis Data	
	1 st	2 nd	1 st	2 nd	1 st	2 nd
Average	685	1,947	1,250	3,446	2,157	6,551

Table 6.3 Average number of positions used in the dictionaries

The Canterbury Corpus contains files of different types (Appendix A). However, a big percentage (57.8%) of it is text. Text has a high degree of redundancy as it can be compressed more than other types of data (see compression ratios achieved in text files in [Bell90] Appendix B). So, because text is more redundant, the same entry in a dictionary is expected to be used more times, and thus, fewer positions in the dictionary are used.

For the Thesis Data Set, the image files increase considerably the average number of positions required. This set contains text, object, audio and image files (Appendix A). Here, while the text files required about the same dictionary size as Canterbury Corpus, the images required up to 4,400 positions for order 1 and 12,000 for order 2, and this increased considerably the average.

Conclusions

If separated dictionaries are used, each keeping contexts of certain order, we may conclude that:

- If the average length of the dictionaries is restricted to 2K positions and 4K positions for 1st and 2nd order contexts respectively, compression ratios would not be affected.
- If the nature of the data to compress is text biased, 1K and 2K dictionary sizes for 1st and 2nd order contexts respectively should be enough space to guarantee good performance.

6.2.3 Multi-dictionary Model - Experiment

Multi-dictionary model strategy was mentioned in Chapter 4 as an alternative for simplifying the complexity of LRU discarding policy. A different policy, *resetting the entire dictionary*, was shown to be the best choice as the simplest form of reclaiming space once the dictionary becomes full and before continuing the model adaptation; however it affects considerably the compression performance.

We consider that the Shift model may benefit from the combination of both strategies by using multiple dictionaries, one for storing contexts of each order of the model so when one of the dictionaries (except for orders -1^{st} and 0^{th}) becomes full, it can be reset. In this way, a 3rd order model will have 5 dictionaries for -1^{st} , 0^{th} , 1st, 2nd and 3rd order models respectively.

This combined discarding policy has not been tested, so there is not knowledge about the performance of the model under these conditions. The following experiment that implements this policy should provide this knowledge:

Assumptions

Order of the model:	3 rd order PPMC model
Dictionary size:	2,048 positions for 3 rd , 2 nd and 1 st order models 257 and 256 positions for 0 th and -1 st orders respectively
Block size:	4,096 bytes, due to this size representing a typical packet size found in many computers and telecommunication systems
Data set:	Canterbury Corpus
Discarding policy:	LRU and resetting the dictionaries that fill up

Method

The Shift model is simulated under the assumptions mentioned and considering one dictionary to store contexts of the same order. When the dictionaries fill up the space assigned to them they are reset, except for orders -1 and 0. To compare the results, a model with separated dictionaries but using the LRU policy was simulated.

Results and discussion

Table 6.4 shows the compression ratios obtained with the models. Storing data in separated dictionaries and resetting one of the dictionaries does not harm compression ratios, just a minimum 1% degradation is observed compared with the LRU policy.

PPMC single dictionary	LRU policy	Resetting an entire dictionary
0.389	0.388	0.393

Table 6.4 Compression performance of the compression model

It should be worthy mentioned that from past experiments (Chapter 4, Section 4.6.1) we know that implementing the model with a single dictionary provides compression ratios of 0.389, while Table 6.4 shows 0.388 for separated dictionaries with LRU discarding policy. This slight improvement in compression may be due to more space

is assigned to contexts of order 1 and 2 and there are more predictions in this order rather than in 0th and -1st orders. So, compression ratio does not seem to be a problem when using one or more dictionaries to store the model. However, the space requirements may vary considerably. Next, we estimate the requirements in every case, showing the results in Table 6.5 and Table 6.6.

Table 6.5 shows the space requirements for the model that stores the data in a single dictionary and uses an LRU discarding policy. The extra requirements for maintaining the policy are to be considered, at least 2,048*9 (length of the dictionary * width) bits are required.

Dictionary order	Space requirements (bits)		
	Symbols	Frequencies	Cumulative frequencies
3 rd , 2 nd , 1 st	65,536	73,728	73,728
0 th	2,313	3,084	3,084
-1	2,048	2,048	2,048
Total	227,617 bits		

Table 6.5 Space requirements for a single dictionary

Table 6.6 shows the space requirements for the model that stores the data in separated dictionaries. No further space is required for discarding policies apart from a few bits to keep the number of positions being used in each dictionary.

Dictionary order	Space requirements (bits)		
	Symbols	Frequencies	Cumulative frequencies
3 rd	65,536	24,576	24,576
2 nd	49,152	24,576	24,576
1 st	32,768	24,576	24,576
0 th	2,313	3,084	3,084
-1 st	2,048	2,048	2,048
Total	309,537 bits		

Table 6.6 Space requirements for separated dictionaries

From these estimates, we observe up to 21% of extra space required when the model is stored in separated dictionaries.

Conclusions

- Independently of the discarding policy used, separating the modelling information in several dictionaries does not harm compression ratios, as this experiment showed.
- Resetting part of the model when one of the dictionaries has filled up provides compression ratios close to the LRU policy and simplifies considerably the complexity of the model at the cost of higher space requirements. This may be the equivalent to the discarding policy used in [Moffat90], where the trie is deleted and 2,048 nodes of the trie are kept to reinitialise the model.

6.2.4 Multi-dictionary Model Experiment – Resizing the Dictionary

From the two previous experiments we know the percentage of predictions that each order context has and that the model could use separated dictionaries for every order of the contexts without really harming compression ratios. Taking into account both results, it seems helpful to resize the dictionaries according to the percentage of symbols predicted in each context. This could result in considerable savings in space. However we do not have this knowledge but the following experiment can provide it.

Assumptions

Order of the model:	3 rd order PPMC model
Dictionary size:	2,048, 512 and 1,024 positions for 3 rd , 2 nd and 1 st order contexts respectively 257 and 256 positions for 0 th and -1 st orders respectively
Block size:	4,096 bytes, due to this size representing a typical packet size found in many computers and telecommunication systems
Data set:	Canterbury Corpus
Discarding policy:	LRU

The dictionary sizes are considered always in powers of two as normally the commercial memories come in those sizes.

Method

Simulation of the model separating contexts of the same order in different dictionaries and having different space limitations proportionally to the percentage of predictions made by each context order.

Results and Discussion

The compression ratio obtained was 0.397, just about 1% of degradation compared with the model that uses multiple dictionaries but has 2,048 positions for each of them. In this case, the dictionary of 2nd order contexts saves 75% of the positions and the dictionary of 1st order contexts halves its size.

Then, there are significant savings in space requirements, about 59% as shown in Table 6.7 and compared with Table 6.6.

Dictionary order	Space requirements		
	Symbols	Frequencies	Cumulative frequencies
3 rd	65,536	24,576	24,576
2 nd	12,288	6,144	6,144
1 st	16,384	12,288	12,288
0 th	2,313	3,084	3,084
-1 st	2,048	2,048	2,048
Total	194,849 bits		

Table 6.7 Space requirements resizing the dictionaries

Conclusions

- It is possible to change the dictionary sizes according to the percentage of predictions given in each context order. This gives a slight degradation in compression ratio but saves considerable amounts of space.
- A good measure to minimise space requirements in this model is to further study the order of the model to implement, and the weight that the contexts of each order have in the predictions. If possible, the study of the type of data also helps to define well-balanced dictionaries in terms of size, as this experiment confirms.

6.2.5 Constant Total Frequency Counts - Analysis

From Chapter 4 we learnt that one of the most expensive operations in the compression system (PPMC model plus arithmetic coder) in terms of complexity and time is the division operation the coder performs. Because of that, some strategies have been developed to remove division from the arithmetic coder operations.

The PPMC model uses an ever-increasing total frequency count as all symbols increase their count when they come in. To obtain the probability of occurrence of each symbol, the arithmetic coder divides the symbol frequency by the total frequency count. This division operation in the worst case, when a symbol is predicted in a -1st

order model, is performed $o+2$ times, where o is the order of the model. Furthermore, because arithmetic coding inputs cumulative frequency counts, two division operations are performed at a time, see the following formulae taken from Chapter 4.

$$\begin{aligned} range &= high - low + 1; \\ high &= low + range * \frac{CF_{S_{i-1}}}{CF_{S_0}} - 1; \\ low &= low + range * \frac{CF_{S_i}}{CF_{S_0}}; \end{aligned}$$

Generally, at the beginning of the compression process it is expected that most of the symbols are predicted in order -1 as the model starts empty and begins to learn the statistics of the data, and in this state the compression process is slower.

In a software implementation of the PPMC model, entire files are compressed. If the files are too small, by the time the model has learnt the statistics of the data the entire file has been compressed. This means that, according to Table 6.1, about 60% of the time the arithmetic coder performs just two division operations (both at once) per symbol but 30% of the time it performs from four to six division operations. This latter figure may rise with different type of data, as with Memory Data that is 50%. These figures are obtained considering that if a symbol is predicted in order 3, the arithmetic coder just performs its operations once for these symbols. However, if a symbol is predicted in 1st order, the model escapes from 3rd order, from 2nd order and finally the symbol is predicted in 1st order, so arithmetic coder perform its operations three times (two divisions at a time) for this symbol.

A division operation in hardware is expensive in terms of complexity. The number of cycles required per division multiplied by the number of divisions required per symbol makes the compression process too slow. This is why avoiding the division operation in the arithmetic coder may improve considerably the compression time.

One of the strategies that may simplify the type of operations performed by the coder, and thus can speed up the whole system, is to consider constant the total number of

symbols seen, as mentioned in Chapter 5. Since it is a contradiction to call ‘constant’ the total number of symbols seen because this amount is ever increasing, we adopt a different terminology. In this, we have called these constant counts *fixed number of tokens (FNT)*. Keeping constant the total frequency counts to a power of two benefits the system by allowing the replacement of ‘divide’ operations in arithmetic coding by simple ‘shifts’ which are faster to execute. The arithmetic coder then needs to shift to the right the cumulative frequency counts by $\log_2 FNT$ positions to substitute the division operation. From now on, where we refer to *shift* operations they will be considered *to the right*.

The new formulae to be used in arithmetic coding would be:

$$\begin{aligned} range &= high - low + 1; \\ high &= low + (range * CF_{s_{i-1}}) \gg FNT - 1; \\ low &= low + (range * CF_{s_i}) \gg FNT; \end{aligned}$$

as shown in Chapter 5, formulae 5.1. The saving in computation costs comes from the type of operations performed. A shift operation is simpler and may be done faster than a division. However, this strategy affects the method for updating the frequency counts in the modelling unit. We learnt from Chapter 4 that to obtain efficient compression ratios having a fixed number of tokens, the number of tokens assigned to the symbols must change proportionally to their occurrence in the input stream. So, updating frequency counts involves a redistribution of tokens. These points are covered in detail in the following two sections.

6.2.6 Number of Tokens - Experiment

In this section we study the *FNT* parameter to determine the value that helps the model to provide the best compression ratios.

Model updating differs now from the PPMC model, as can be seen in Table 6.8. As *FNT* is a constant value, frequency counts no longer increase by one. Instead of that,

all symbols donate tokens to the incoming one to redistribute tokens proportionally to the probability of occurrence of the symbols. This donation is performed through a division operation where, if the dividend is kept to a power of two, the operation may be substituted by simple shifts of m positions, with $m = \log_2(\text{dividend})$. The result of this operation is the number of tokens they donate, TIS . In higher order models other than 0th only symbols sharing the current context may donate tokens.

	PPMC model	Shift model
Updating frequency counts	$Nf_{s_i} = f_{s_i} + 1$ $Nf_{Escape} = f_{Escape} + 1$	$SD_{S_j} = f_{s_j} >> m$ $Nf_{s_j} = f_{s_j} - SD_{S_j}$ $TIS = \sum_{i=1}^k SD_{S_j}$ $Nf_{s_c} = f_{s_c} + TIS$
arithmetic coding operations	$high = low + range * \frac{CF_{S_{i+1}}}{CF_{S_0}} - 1;$ $low = low + range * \frac{CF_{S_i}}{CF_{S_0}};$	$high = low + (range * CF_{S_{i+1}}) >> FNT - 1;$ $low = low + (range * CF_{S_i}) >> FNT;$

Table 6.8 Formulae required for the PPMC and Shift algorithms

In Table 6.8, SD_{S_j} represents the tokens donated by symbol j , Nf_{s_j} indicates new frequency count of symbol i . The '>>' symbol indicates shift operation to the right and in the case of the arithmetic coder, the range is obtained by subtracting low to $high$ values for both models. k is the number of different symbols seen, and c is the index of the current symbol. The Shift algorithm using the formulae in Table 6.8 would be:

1. Initialise the model with the Escape symbol having its frequency count set to FNT
2. Repeat for every incoming symbol:
 3. Search symbol in all context orders
 4. From context order where the symbol is found (except -1st) and above do:
 5. All symbols donate tokens – obtaining Nf_{s_j} , for $j = 1$ to k
 6. Compute TIS adding all the donations from other symbols
 7. Compute frequency count of current symbol adding TIS
 8. If symbol is new verify if there is space in every dictionary
 9. If there is space,

10. Add new symbol, frequency and cumulative frequency counts
11. Else
12. Reset the dictionary where there is not space
13. Add new symbol, frequency and cumulative frequency counts
14. Update cumulative frequency counts of all symbols with same context
15. Perform arithmetic coding operations
16. Go to step 2

The following is an experiment to determine the best value of *FNT* parameter:

Assumptions

Order of the model:	0 th , 1 st and 2 nd PPMC order models
Dictionary size:	4,096 positions for 1 st and 2 nd orders to simulate no space restrictions, 257 and 256 positions for 0 th and -1 st orders respectively
Block size:	4,096 bytes, due to this size representing a typical packet size found in many computers and telecommunication systems
Data set:	Canterbury Corpus
Discarding policy:	Not required
Number of tokens (<i>FNT</i>)	Varied, from 512 to 262,144
Positions to shift <i>FNT</i> , <i>m</i>	Varied from 1 to 9

Methodology

We set a value to *m* and *FNT* and execute the simulation. Then, we vary *FNT* to find the best value that generates the best compression ratios. All context levels have the same value of *FNT* and *m*. The first symbol to occur is Escape and to this symbol is assigned *FNT* value. As symbols are entered, the tokens are redistributed.

Results and discussion

Figure 6.1 shows the compression performance obtained with the 0th order Shift algorithm. As can be seen, compression ratios similar to PPMC are obtained with large values of *FNT*, for example when *FNT* is between 32,768 and 262,144. These values together with the number of positions to shift, *m*, guarantee that the model adapts as in PPMC, i.e. reflecting the statistics of the data. Smaller values of *FNT* do not give enough flexibility to model to redistribute tokens and thus to adapt correctly to the changes in the input.

Naturally, the bigger the count in *FNT*, the higher the number of bits required to represent them. Thus, for the 0th order model, a *FNT* value of 65,536 offers the best balance between hardware requirements and compression results.

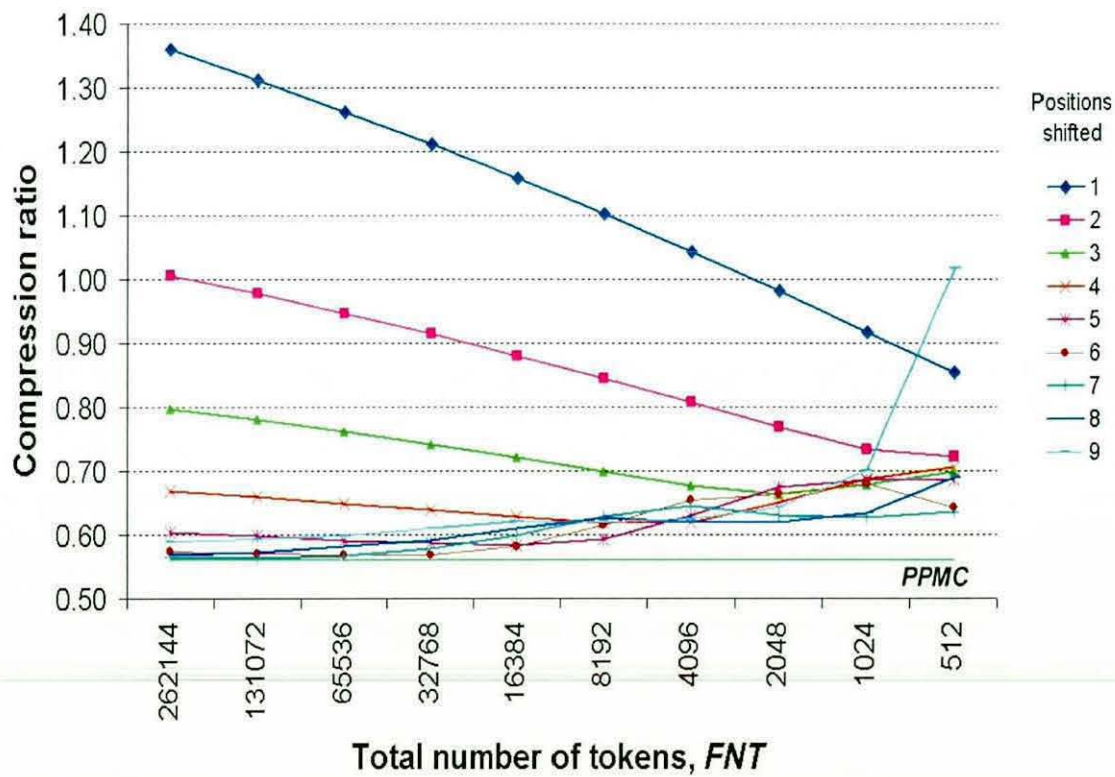


Figure 6.1 Compression ratio as *FNT* changes for 0th order model

Figure 6.2 shows the compression performance obtained with the 1st order Shift algorithm. As can be seen, the closest compression ratio to PPMC is 0.468 and it is obtained with 4,096 tokens for *FNT*, while shifting four positions. That means,

contexts of order 0 and 1 have a maximum number of tokens of 4,096. As symbols come in, the updating process redistributes frequency counts corresponding to the current context by shifting four positions and later updating the cumulative frequencies.

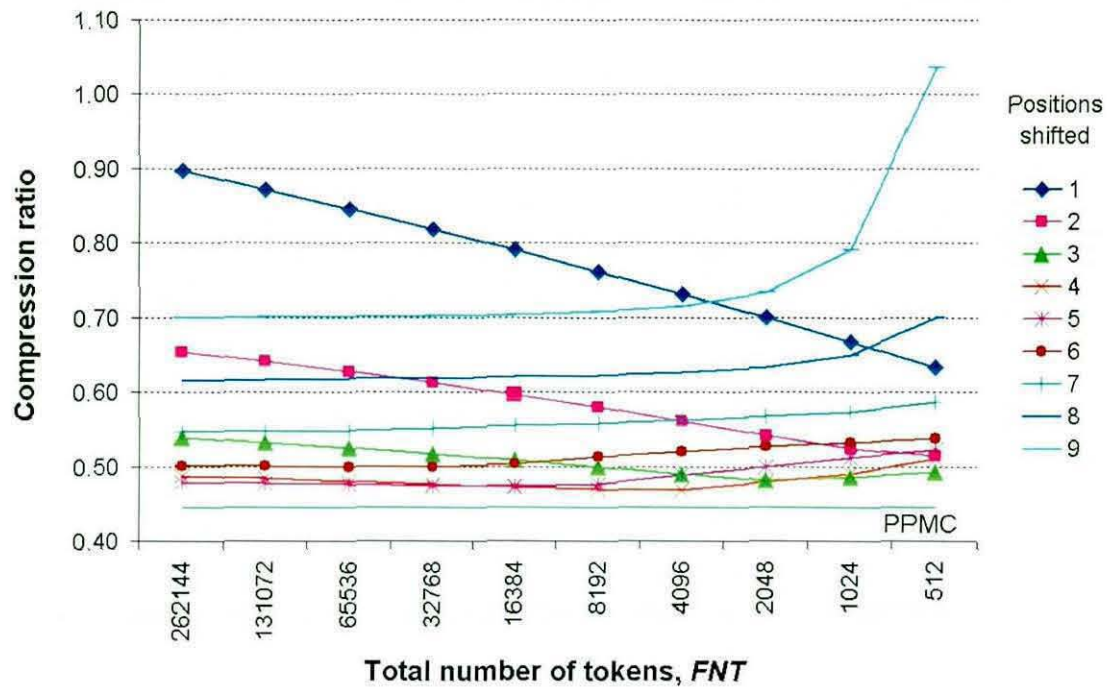


Figure 6.2 Compression ratio as *FNT* changes for 1st order model

Figure 6.3 shows the compression performance of the 2nd order model. The closest compression ratio to the PPMC algorithm is achieved with a value of *FNT* of 2,048 as Figure 6.3 shows, providing a compression ratio of 0.427.

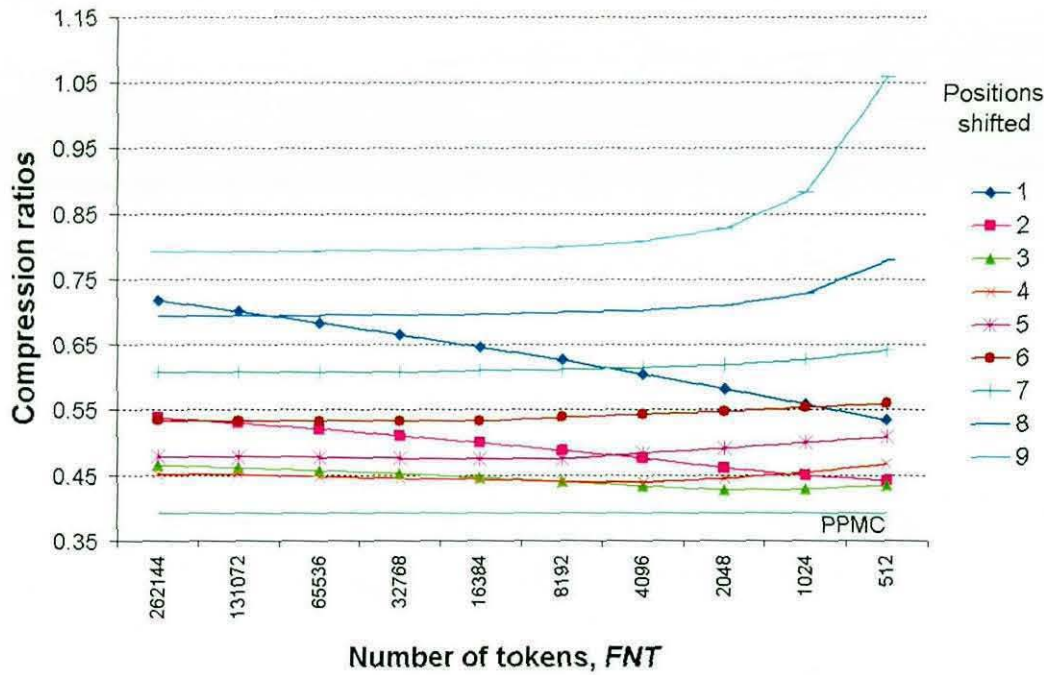


Figure 6.3 Compression ratio as *FNT* changes for 2nd order model

From these experiments it can be noticed that as the order of the model increases, the difference in compression ratio with respect to the PPMC algorithm also increases. This is because the speed of adaptation is affected by the order of the model. As the order of the model increases, the adaptation of the model becomes slower due to the higher number of contexts present. Slow adaptation leads to poorer compression performance because the model does not really gather the statistics of the data according to their occurrence in the input stream.

The best choice of values for *FNT* parameter that we found in this experiment is shown in Table 6.9.

Order of the model	<i>FNT</i>
0 th	262,144
1 st	4,096
2 nd	2,048

Table 6.9 Best choice of values for *FNT* and positions to shift

Although this experiment is about *FNT* values, it is almost impossible to decide the best choice without considering also the positions to shift, as these two parameters are closely linked. However, m is further analysed in the next section.

Conclusions

- The value of *FNT* is a very important parameter to consider, as it effects compression performance. As the order of the model increases, its value must diminish to ensure better adaptation of the model that leads to better compression performance.
- The *FNT* parameter can not fully determine the success of the algorithm as it is closely linked with m . It can be thought that these two values measure to a certain degree the speed of adaptation of the model in addition to the order of the model.
- The fact that a slight degradation in compression ratio compared with the PPMC model is observed when the order of the model increases, leads us to think that probably there is one more parameter to consider that may help to better adjust symbol probabilities.

6.2.7 Positions to Shift Frequencies - Experiment

In this section we study the impact that the number of positions to shift the frequency counts has on the compression ratio. Although from the previous experiment we learnt that this parameter is not independent of the *FNT* value, it has been analysed as independent as possible.

Assumptions

Order of the model:	0 th , 1 st and 2 nd PPMC order models
Dictionary size:	4,096 positions for 1 st and 2 nd orders to simulate no space restrictions, 257 and 256 positions for 0 th and -1 st orders respectively
Block size:	4,096 bytes, due to this size representing a typical packet size found in many computers and telecommunication systems
Data set:	Canterbury Corpus
Discarding policy:	Not required
Number of tokens (<i>FNT</i>)	Varied, from 512 to 262,144
Positions to shift <i>FNT</i>	Varied from 1 to 9

Methodology

For this experiment we simulated the model, fixing *FNT* and varying *m*, to find the value of *m* that helps the model to generate the best compression ratios. We have been able to rearrange the data used to generate Figure 6.1, Figure 6.2 and Figure 6.3, inverting the axis, to identify the number of positions to shift frequency counts.

The first symbol to occur is Escape and *FNT* value is assigned to it. As symbols are entered, the tokens are redistributed. All context levels (dictionaries) have the same value of *FNT* and *m*.

Results and Discussion

Figure 6.4 shows the results of varying *FNT* and *m* in a 0th order model. When *m* is small, the model does not reflect symbol probabilities according to their occurrences in the input stream due to symbols donating fewer tokens than they should and this fact leads to poor compression.

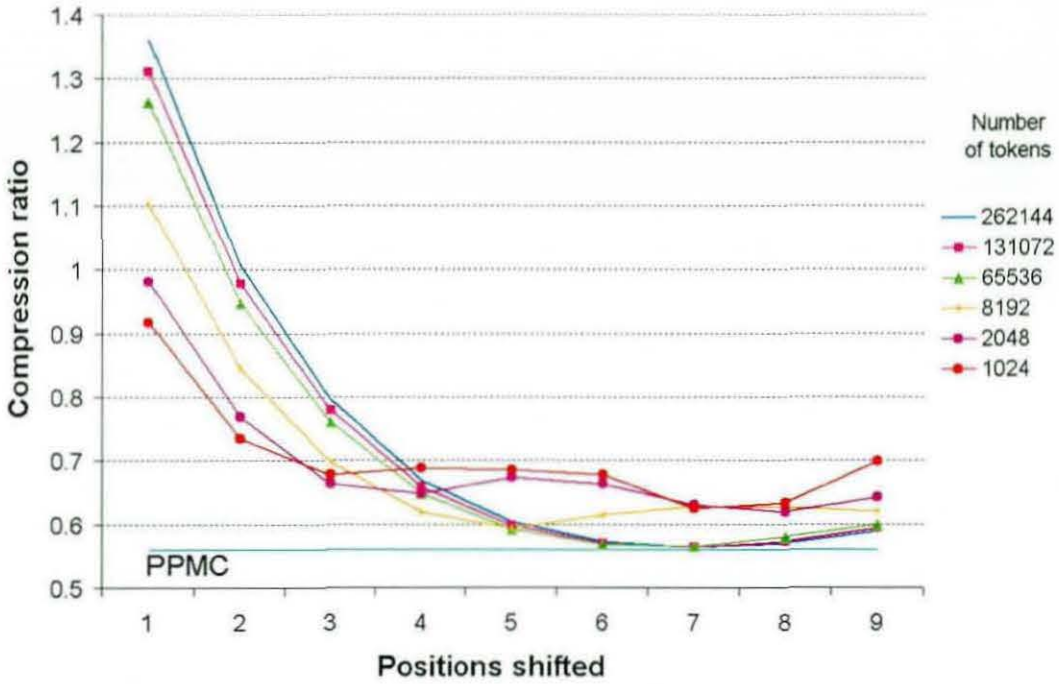


Figure 6.4 0th order model compression results as m and FNT varies.

From this figure we observe that:

- In general, small FNT values do not generate proper adaptation of the model since there are not enough tokens to reflect the occurrences of the symbols.
- Very good compression ratios can be observed, close to PPMC, shifting the frequency counts up to seven positions. Clearly, the value of m that yields the best compression ratios is seven.

Figure 6.5 shows the simulation results of the 1st order model as the number of positions to shift the frequency counts varies. From this figure it can be seen that when shifting between three and six positions, the frequency counts provide good compression ratios with the best results for m equal to four.

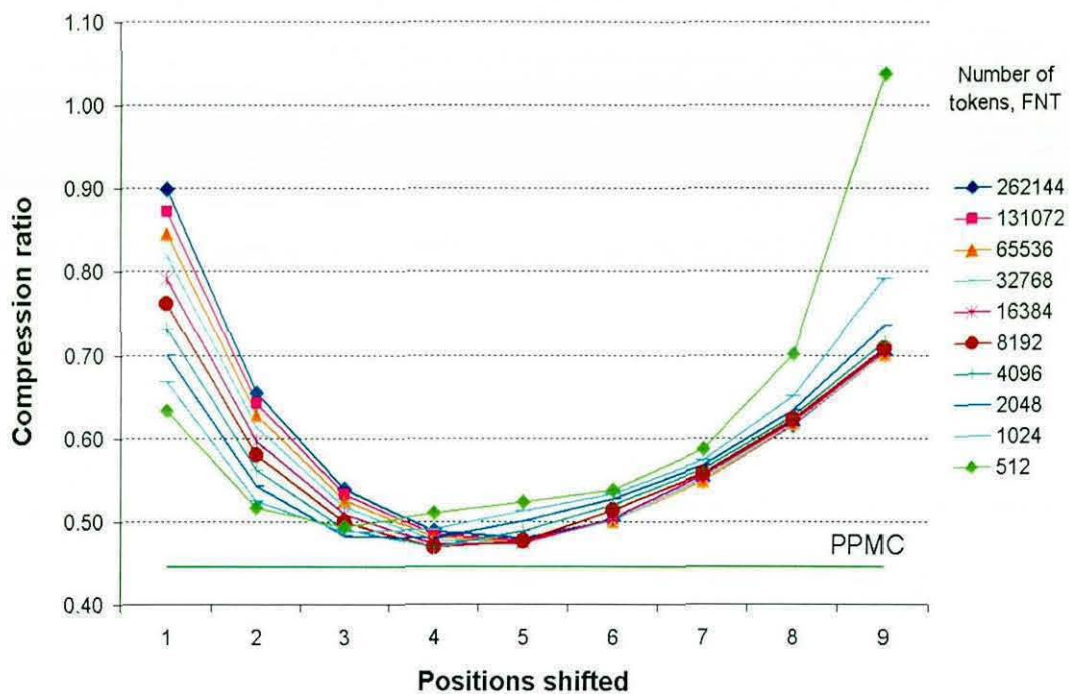


Figure 6.5 Compression ratios with 1st order model

From the figure we observe that:

- In the 1st order model, small values of m generate poor compression ratios but not as much as in the 0th order model. However, large values degrade compression more. This difference in compression is due to the high number of possible contexts in the 1st order model, while in the 0th order there is just one possible 'context' (empty); in the 1st order model there are many contexts and their probability of occurrence is lower. So, each of these contexts have less opportunity to redistribute the tokens assigned to the Escape symbol. If m has a large value, the Escape remains with high probability, while the symbols have lower probabilities than the one they should have and this causes poorer compression ratios.
- When m has a large value, Escape may deplete its tokens when new symbols are still coming, so the model continues escaping with low probabilities in Escape symbols, which requires more bits to represent these low probabilities and the compression ratio worsens.

If the previous explanation of the changes in compression is accurate, the 2nd order model should maintain this tendency, i.e. large values of m should generate even poorer compression ratios, and the results confirm this. Figure 6.6 shows the compression ratios obtained with the 2nd order model. In general, the better compression ratios are obtained when shifting frequency counts three or four positions. When comparing the changes in this model against 0th and 1st order models, it can be seen how the compression ratios with small values (1 and 2) of m are slightly better; however, for values of m bigger than 4 the compression ratios degrade considerably.

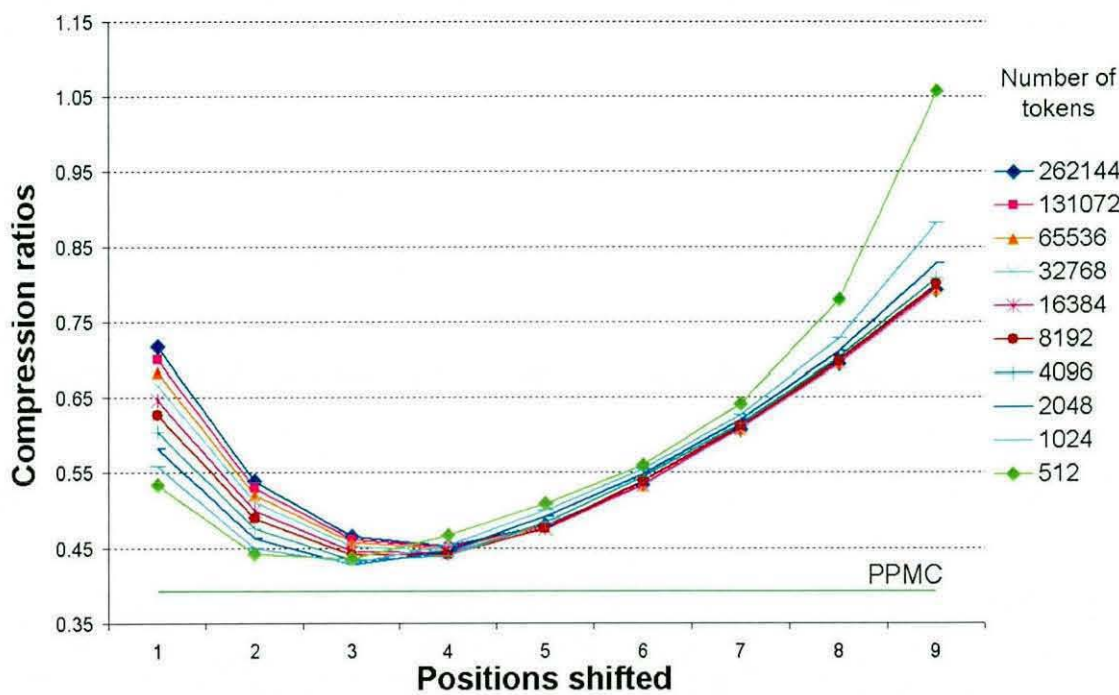


Figure 6.6 Compression ratios with 2nd order model

From this experiment, we summarise the best choices for the number of positions to shift to the right, m , the frequency counts as in Table 6.10.

Order of the model	m
0 th	7
1 st	4
2 nd	3

Table 6.10 Best choices for m

Conclusions

- This experiment showed that m , the number of positions shifted, has a significant impact on compression ratio. This value must be chosen carefully if the best compression ratio is to be obtained. From the last section we learn that the best choices of FNT depends on m and here we have seen also that the best choice of m depends on the value of FNT , but also on the order of the model.
- Parameter m measures the speed of adaptation in symbol probabilities. Large values of m make symbols donate fewer tokens to the incoming symbol and small values of m make symbols donate more tokens.

To summarise, we have learnt that the higher the order of the model the more contexts may occur and the lower probability of occurrence the contexts may have. Thus, bigger changes in frequency counts are needed to adapt faster the model to the statistics of the data and so the m parameter must be smaller as the order of the model increases.

6.2.8 Frequencies and Positions Shifted - Experiment

From the previous two sections, we found that FNT and m must differ according to the order of the model to obtain the best compression ratios. This led us to think that probably varying the values of the FNT and m among the different context levels could improve compression ratios. In this section we study both parameters, differing among the context orders.

We learnt that as the order of the model increased, the value of FNT and m decreased. We will simulate a model with different values among the orders of the contexts. Contexts of lower order will have larger values of FNT and m , and contexts of higher orders will have smaller values.

Assumptions

Order of the model:	0 th , 1 st and 2 nd PPMC order models
Dictionary size:	4,096 positions for 1 st and 2 nd orders to simulate no space restrictions, 257 and 256 positions for 0 th and -1 st orders respectively
Block size:	4,096 bytes, due to this size representing a typical packet size found in many computers and telecommunication systems
Data set:	Canterbury Corpus, Memory and Thesis Data Sets were compressed
Discarding policy:	Not required
Number of tokens (<i>FNT</i>)	Varied, from 512 to 262,144
Positions to shift <i>FNT</i>	Varied from 1 to 9

Methodology

Firstly, we study a 0th order Shift algorithm, we simulate the algorithm similarly to those in the two previous sections. Then, once knowing the best *FNT* value for the 0th order model, we simulate the 1st order Shift model. The 0th order contexts use the value found in the first simulation and the 1st order contexts vary the value of this parameter to find the one that gives the best compression ratio. We follow the same procedure for the 2nd order model and later modify the value of *FNT*.

When a frequency count is going to be updated, other symbols donate tokens to the incoming one. Tokens donated are added together and assigned to the corresponding frequency count. In this way, it is guaranteed that the tokens are not lost and the *FNT* value remains constant along the compression process.

Results and analysis

Sections 6.2.5 and 6.2.6 are the bases of Table 6.11 that shows the *FNT* and *m* parameters that exhibit the best compression ratios for the model; they are shown for

models of orders 0th to 2nd. In order -1 both parameters are not relevant since the symbols will have the same probabilities. Thus the same results are obtained independently of the values chosen for those parameters.

		Context Order		
		0 th	1 st	2 nd
FNT	Order 0 th	65,536		
	Order 1 st	65,536	16,384	
	Order 2 nd	32,768	8,192	4,096
m	Order 0 th	7		
	Order 1 st	7	5	
	Order 2 nd	6	4	3

Table 6.11 Fixed number of tokens, *FNT*, and number of positions to shift, *m*

For example, Table 6.11 indicates that in a 2nd order model, the *FNT* value for contexts of the 0th order is 32,768 and their frequencies are shifted seven positions when updated. The *FNT* value for contexts of the 1st order is 8,192 and frequencies are shifted four positions while 2nd order contexts have a *FNT* of 4,096 and shift just three positions to the right.

At the beginning of the compression process the *FNT* value is assigned to the escape symbol. As symbols come in, this value is redistributed among the symbols. The adaptation speed of the Escape is different for every context order. Higher order contexts adapt slower than lower order ones because the number of possible contexts is higher; so, when a symbol is predicted by the highest order context, among all the contexts only one is updated. The higher the order of the model, the more possible contexts, so, it is more probable that Escape will have a high probability at all times.

As explained in Chapter 5, Escape has a large number of tokens in a transitory state, i.e. at the beginning of the compression process, which represents a high probability of occurrence. This high probability is mapped into few bits by the coder. While in this state, if tokens are redistributed according to the strategy of this section, other symbols get fewer tokens than the number that would reflect their real occurrence in the input

stream. Then, in this state it is expected that the model escapes more frequently, which is when the Escape has higher probabilities. In a steady state, Escape symbol has the lowest possible probability while more probable symbols have divided among them the tokens corresponding to Escape, assigning them higher probabilities than they deserve. In this way, Shift model benefits in both states of the compression process.

Conclusions

- In higher-order models when *FNT* and *m* values differ among the order of the contexts improves compression. This fact is related to the possible number of contexts and the chances of every of these contexts to predict a symbol.
- The relation that explains the previous improvement is that, as the number of different contexts increases, they have a lower chance of predicting the incoming symbol, so large *FNT* values adapt slowly. In this case, a close behaviour to PPMC is achieved with small *FNT* values.
- Using each context order different *FNT* and *m* parameters, the arithmetic coder must input the order of the context as well as symbol frequency counts. This is to ensure the proper calculation of symbol probabilities and so the correct functionality of the arithmetic coder.

6.2.9 Parallel Frequency Updating - Discussion

To update frequency counts, the *tokens* must be redistributed among the symbols according to the symbol occurrence in the input stream. The shift operation (equivalent to the division) allows the redistribution of *tokens* according to this occurrence. All symbols in the current context must donate tokens to the incoming symbol. Their frequency counts are shifted *m* positions to the right (equivalent to divide by 2^m) and the results of these operations are added together. This sum is then assigned to the incoming symbol.

When implementing the Shift model in hardware, the redistribution of tokens may be done in serial or parallel form. The former requires one shifter to shift all the corresponding frequency counts (Figure 6.7a). This process is time consuming since in the worst case all symbols may have occurred and all of them donate tokens. The second form requires more operators but all shift operations may be done in parallel (Figure 6.7b).

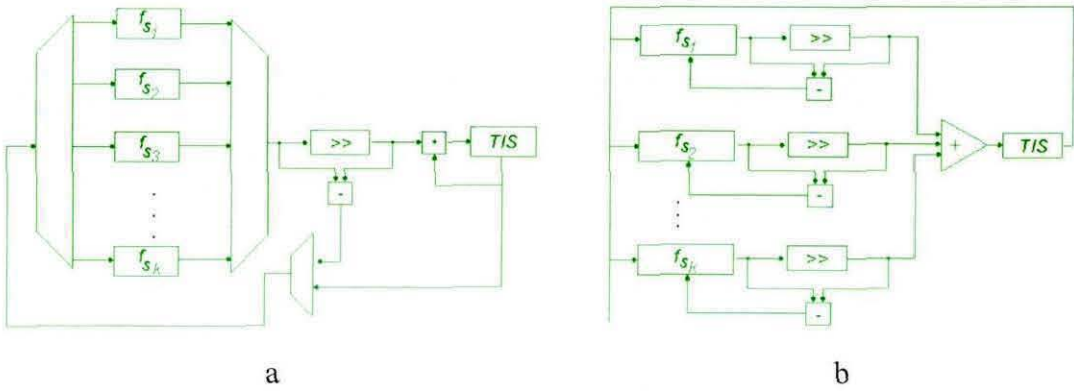


Figure 6.7 Architecture for frequency updating

Shift model is focused on increasing compression speed, thus the second form for redistributing *tokens* (Figure 6.7b) is suggested.

6.2.10 An Approximation for Updating Symbol Frequencies - Experiment

As mentioned above, the model adds together the tokens for the incoming symbol (TIS). The adding operation when implemented in hardware would require either an adder tree or a single adder. The first case is costly in terms of logic and requires $O(\log_2 q)$ time, where q is the size of the alphabet. The second case requires $O(k)$ time where k is the number of different symbols seen in the current context (that in the worst case $k = q$), it is cheap and simpler to implement but costly in speed. Since our model is being designed for fast performance we consider it more convenient to follow the first approach.

Even this approach could be very expensive in terms of time and complexity because a network of connections would be required to connect the output of the shift operations to the input of the tree. So, in this section, we explore a form of approximating the *TIS* value that could be fast and simple without degrading the compression performance.

Reviewing how the process is done currently we observe that the fixed number of tokens, *FNT*, is constant along the compression process, that means that at any moment of the compression process:

$$FNT = \sum_{j=1}^k f_{s_j} \quad (6.1)$$

where f_{s_j} is the frequency count of the j^{th} symbol and k is the number of symbols seen in the current context.

Recalling the formulae of Table 6.8, when symbols donate tokens the frequency counts f_{s_j} are shifted m positions to the right, thus, the new frequency count, Nf_{s_j} is obtained as in (6.2).

$$Nf_{s_j} = f_{s_j} - (f_{s_j} \gg m) \quad (6.2)$$

TIS value is obtained by adding the tokens donated as in (6.3).

$$TIS = \sum_{j=1}^k (f_j \gg m) \quad (6.3)$$

The following experiment simulates Shift algorithm and observes how *TIS* value varies during the compression process.

Assumptions

Order of the model:	0 th order PPMC model
Dictionary size:	257 positions, no space restrictions
Block size:	4,096 bytes, due to this size representing a typical packet size found in many computers and telecommunication systems
Data set:	Canterbury Corpus and Memory Data Set
Discarding policy:	Not required
Number of tokens (<i>FNT</i>)	65,536 , shown in previous experiments to be a good value for the performance of the model (Section 6.2.6)
Positions to shift <i>FNT</i>	7, shown in previous experiments to be a good value for the performance of the model (Section 6.2.6)

Methodology

To compare how the number of tokens for the incoming symbols is computed, we observed the real ‘assignment’ of tokens to the incoming symbol, i.e. without approximating the value of *TIS*, running Canterbury Corpus and Memory Data. When a frequency count is updated, other symbols donate tokens to the incoming one. Tokens donated are added together and assigned to the corresponding frequency count.

The simplest approximation is to consider the *TIS* value constant, so no computation is required. The Shift model was simulated assigning a constant value to *TIS*; we observed the results and later on we simulated with a different approximation explained below.

Results and Analysis

The constant approximation consisted in setting *TIS* as in (6.4). In this case, $TIS = 512$. However it generated overflow in the arithmetic coder for almost every type of data. Diminishing this value avoids overflow but the degradation it generates in compression results makes this approximation unsuitable for this model.

$$TIS = FNT \gg m \tag{6.4}$$

Then, without the approximation, we compressed Canterbury Corpus and Memory Data and observed that the number of tokens assigned to the incoming symbol was related to the number of symbols seen, as Figure 6.8 shows. The values of the figure are the average number of tokens donated to a symbol for all the values of k (number of different symbols seen in the current context) in the different files. The first value of TIS is 511, that is:

$$TIS_{S_i} = (FNT \gg m) - 1 \tag{6.5}$$

This value diminishes as new symbols are entered, that is subtracting the value of k .

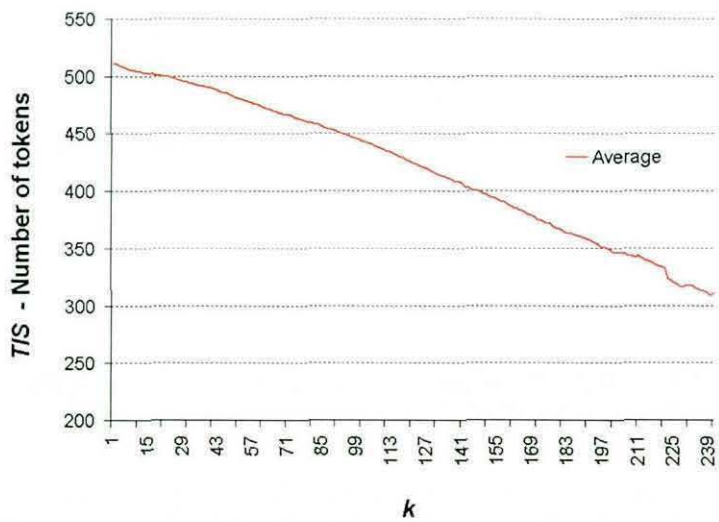


Figure 6.8 Average TIS as the number of symbols are seen in the current context increases

From the figure, it can be observed that the value of TIS_{S_i} diminishes with the number of symbols seen in the current context, k . So, we approximate the value of TIS as formula (6.6).

$$TIS_{s_e} = ((FNT \gg m) - 1) - k \quad (6.6)$$

The amount in (6.5) is computed only once at the beginning of the compression process and it is kept constant, we refer to it as C , then:

$$TIS_{s_e} = C - k \quad (6.7)$$

Formula (6.7) requires only subtracting k from C per each incoming symbol when obtaining TIS number for the current symbol. This approximation implies that from now on, FNT is no longer fixed in the model, but for coding it remains fixed so that the shift operations may be done.

The Shift model was simulated for orders 0, 1 and 2 with the best values of FNT and m of Table 6.11. It showed a degradation in compression ratio of about 3% to 5% for 1st and 2nd order models respectively with respect to the simulations with 'perfect' TIS value, which is not significant considering the savings in space, complexity and time. These savings are due to the simplification of the computation of TIS value as showed in Figure 6.7.

Conclusion

- It is possible to approximate the value of TIS with very significant savings in time and complexity at a low cost in terms of compression ratio (3% to 5%). The savings include performing a simple subtraction instead of the sum of the tokens donated.

6.3 SHIFT ALGORITHM

This section summarises the Shift algorithm including what we learnt from this and previous chapters. We explain the algorithm and show it in serial and ‘parallel’ form. The second form takes its name due to the parallel frequency updating process. We also mention its characteristics like dictionary size, discard policies, model order, approximations for model updating, etc.

The Shift algorithm, similarly to the PPMC algorithm, predicts symbols based on statistics of the past symbols. The best prediction is made by the longest context that provides more accurate probabilities. If the symbol can not be predicted in this context, the model escapes to the next lower order model. The operation is repeated until the model predicts the symbol or reaches the order ‘-1’ where all the symbols have the same probability.

Figure 6.9 shows the Shift algorithm.

```

Clear the dictionary;
set context order (CO) to order of the model;
set longest context (LC) to CO previous symbols;
DO
{
    read in a symbol from the data stream;
    DO
    {
        search for LC & incoming symbol in the dictionary;
        IF (found)
        {
            output cumulative frequencies;
            IF(CO!='-1') update in the dictionary LC contexts;
            Add symbol from LC+1 to CO contexts;
        }
        ELSE
        {
            output 'escape' cumulative frequencies from LC;
            subtract 1 from CO;
            IF(CO>0) 'drop' the most left symbol from the context;
        }
    }WHILE(symbol is not predicted);
    update LC;
    compute arithmetic coding operations;
} WHILE( more data is to be compressed);

```

Figure 6.9 Shift algorithm in serial form

This algorithm executes serial search operations; however, to speed up the compression system we redesigned the updating process for hardware implementation (Section 6.2.9), executing so parallel search operations. This algorithm is shown in Figure 6.10.

```

Clear the dictionaries;
Set LC to context;
set CO (context order) to order of the model;

DO
{
    read in a symbol from the data stream;
    search for LC & incoming symbol in all the dictionaries;
    select best match and set BMO = order of best match;
    IF (order of best match = CO)
    {
        output symbol cumulative frequencies;
        update frequencies in dictionary of CO;
    }
    ELSE
    {
        from BMO to CO do:
            output 'escapes' ( $CF_{Esc}$ ) of orders BMO+1 to CO;
            output CFs from BMO;
            add LC + symbol to dictionaries of orders BMO+1 to CO;
            update frequency counts in dictionary of BMO;
    }
    recompute cumulative frequencies, CFs;
    update LC;
    compute arithmetic coding operations;
} WHILE( more data is to be compressed);

```

Figure 6.10 Shift model in parallel form

This parallel algorithm for the Shift model uses the formulae in Table 6.8, but considering the approximation for TIS value as in section 6.2.10, would be:

1. Initialise the model with Escape symbol having its frequency count set to FNT
2. DO
3. {
4. Search symbol in all context orders;
5. From context order where the symbol is found (except -1^{st}) and above do:
6. All symbols with current context donate tokens $Nf_{s_j} = f_{s_j} - (f_{s_j} >> m) \quad \forall j \in \{1, \dots, k\}$;

7. Compute $TIS = C - k$, where $C = (FNT \gg m) - 1$;
8. In the order of best match: compute $Nf_{s_c} = f_{s_c} + TIS$;
9. If symbol is new verify if there is free space in every dictionary
10. If there is space,
11. Add new symbol, frequency and cumulative frequency counts
12. Else
13. Reset the dictionary where there is not space
14. Add new symbol, frequency and cumulative frequency counts
15. For all symbols with same context do $CF_{s_c} = CF_{s_{c-1}} + F_{s_c}$
16. Perform arithmetic coding operations
17. WHILE(more symbols in file)

In the algorithm, Nf_{s_i} indicates new frequency count of symbol i . The ' \gg ' symbol indicates shift operation to the right, k is the number of different symbols seen and c is the index of the current symbol.

As mentioned in Section 6.2.5, arithmetic coding operations now are:

$$\begin{aligned} high &= low + (range * CF_{s_{i-1}}) \gg FNT - 1; \\ low &= low + (range * CF_{s_i}) \gg FNT; \end{aligned}$$

where *range* is obtained by subtracting the *low* bound to the *high* bound. Divide operations have been substituted by simple shifts, which impact compression speed, making the algorithm faster. Since the *FNT* value can be different for every context order, the arithmetic coder requires this parameter together with the cumulative frequency counts of the symbol being codified. Other operations for the normalisation procedure are as Section 6.2.5 states them and they remain unchanged.

6.3.1 Assumptions

Order of the model:	0 th , 1 st and 2 nd order PPMC models
Dictionary size:	For 2 nd order model: 256, 256, 2K and 4K positions for – 1 st , 0 th , 1 st and 2 nd order contexts respectively For 1 st order model: 256, 257 and 4K positions for –1 st , 0 th and 1 st order contexts respectively
Block size:	4,096 bytes, due to this size representing a typical packet size found in many computers and telecommunication systems
Data set:	Canterbury Corpus, Memory and Thesis Data
Discarding policy:	Reset a complete dictionary once its space allocated has been consumed
Number of tokens (<i>FNT</i>)	as shown in Table 6.11
Positions to shift <i>FNT</i>	as shown in Table 6.11

The model stores escape frequencies as any other symbol frequency.

6.3.2 Methodology

To simplify the investigation, we studied the model and coder independently of each other. The model reads serially the data to be compressed and produces cumulative frequency counts of the symbols that are then transmitted as input to the coder. The arithmetic coder was taken from [Witten87] and adapted to the proposed model, i.e. substituting the division operation by shifts and accepting the *FNT* value as a parameter together with the cumulative frequency counts. Models of orders 0, 1 and 2 were simulated in Visual C++.

6.3.3 Results and Analysis

Figure 6.11 shows the compression results for Canterbury Corpus. As it can be seen in this figure, for all the files the Shift algorithm has a slight degradation compared with PPMC algorithm.

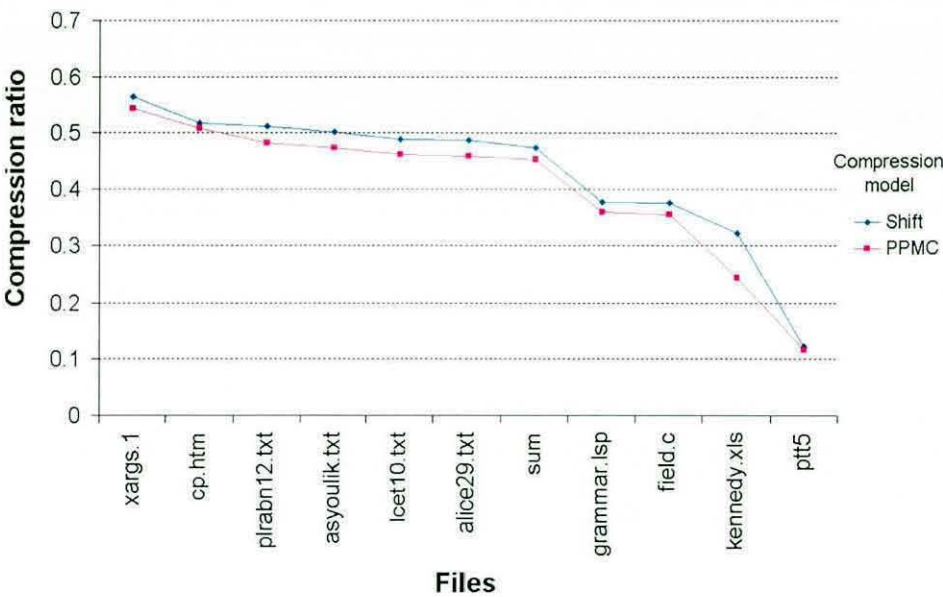


Figure 6.11 2nd order Shift algorithm compression for Canterbury Corpus

Memory Data compression is shown in Figure 6.12. Again, as for Canterbury Corpus, Shift algorithm degrades compression for all the files when compared with PPMC.

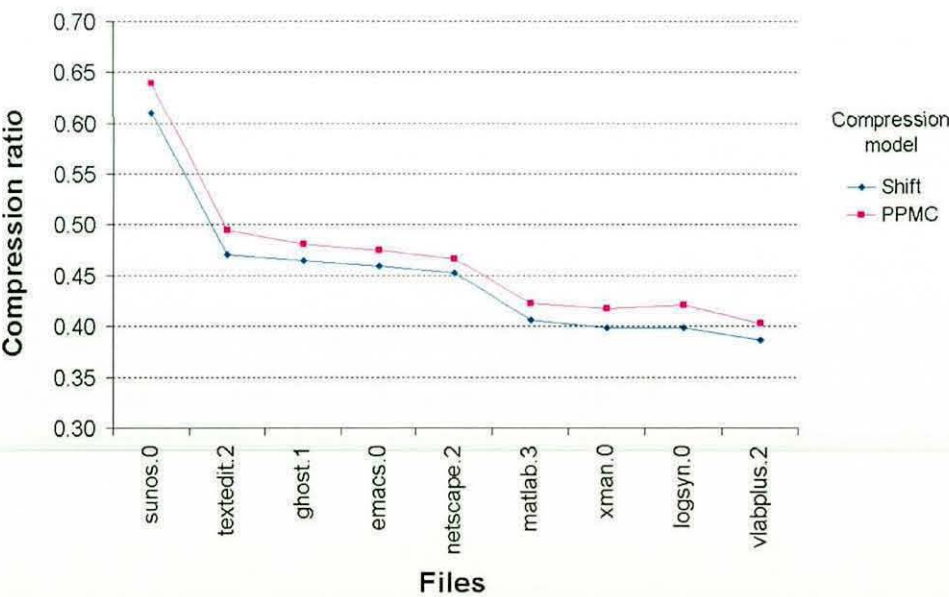


Figure 6.12 Compression ratios for Memory Data

Figure 6.13 shows the results of the compression of the Thesis Data Set. The first group of data comprises audio data, where it can be seen that for some files Shift algorithm gives a slight improvement in compression. For almost all the files of image data Shift algorithm performs better than PPMC and for text data it is the opposite, PPMC compressing better.

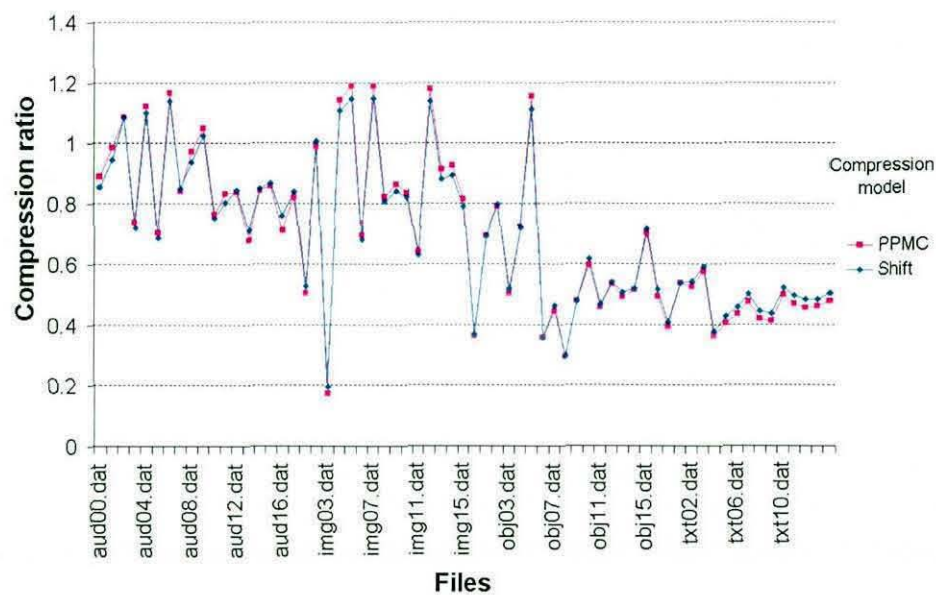


Figure 6.13 Compression ratios for Thesis Data

Table 6.12 shows a summary of the average the compression ratios that Shift algorithm provides for Canterbury, Memory and Thesis Data Sets showing comparisons with PPMC algorithm for each of them.

Data Set	Model	Order		
		0 th	1 st	2 nd
Canterbury Corpus	PPMC	0.56	0.45	0.40
	Shift	0.57	0.47	0.43
Memory Data	PPMC	0.61	0.46	0.45
	Shift	0.61	0.49	0.47
Thesis Data	PPMC	0.75	0.68	0.66
	Shift	0.75	0.68	0.67

Table 6.12 Compression ratio results obtained with PPMC and Shift algorithms

In Table 6.12 for each set of data, the first row shows results from PPMC and the second one from Shift algorithm. This table shows how Shift algorithm provides compression ratios close to PPMC.

A comparison of the computational cost of PPMC and Shift models, is presented in Table 6.13.

Process	Algorithm	
	PPMC	Shift
Updating	Few operations	Many simple operations
Searching	Many operations Complex process	Few operations Fast process
Arithmetic coding	Slow process	Fast process

Table 6.13 Comparing complexity of PPMC and Shift algorithms

It is difficult to do a fair comparison of the complexity of the PPMC and Shift algorithms since PPMC has been implemented in software, and Shift is PPMC being reorganised for hardware implementation. However, based on hardware implementations of other compression algorithms, we know that the data structure as well as the type of operations proposed for Shift algorithm are suitable for hardware implementations, and thus we expect Shift algorithm to perform faster than a direct implementation of the PPMC algorithm.

6.3.4 Conclusions

- It has been shown that it is possible to rearrange PPMC algorithm for its hardware implementation and it is expected to perform faster than if the original PPMC were implemented.

- The redistribution of tokens guarantees a proportional adjustment of symbol probabilities according to its occurrence in the input stream. However, this may add some complexity in the updating process.
- Approximating the number of tokens to assign to the incoming symbol (*TIS*) represents great savings mainly for hardware complexity. It changes either an adder tree or a single adder with $O(n)$ delay by a simple subtractor. However, it yields, in most of the cases, a small degradation in the compression ratio.
- Storing contexts in separated dictionaries has the advantage of reducing complexity mainly when dealing with discard policies other than resetting the dictionary. Properly ‘tuning’ the dictionary sizes, the space requirements may even diminish (see Section 6.2.4).

6.4 SUMMARY

This chapter gives an overview on the issues involved in optimising PPMC algorithm for its hardware implementation. On the one hand, the coder is reorganised by using simple operations, which lead to faster implementations. On the other hand, the model is also reorganised to adapt to these changes in the coding while maintaining the performance of PPMC algorithm. Additionally, some changes that help to further simplify a hardware implementation involves approximating some values. This modification leads to small degradation in compression ratios that are not significant when we consider the savings in complexity.

The set of modifications proposed give birth to a new algorithm, that albeit a slight degradation in compression ratio, it keeps its statistical nature and adaptability and is suitable for hardware implementation. However, the experimentation in this chapter has been through simulations of the algorithm, but no hardware modelling has been done that may test the hardware suitability, and this is the topic of the next chapter.

CHAPTER 7

HARDWARE MODELLING

7.1 OBJECTIVES OF THE CHAPTER

This chapter looks into the hardware architecture for the Shift algorithm and presents an estimation of hardware requirements and performance. The outcomes are analysed to determine how close it corresponds to the expected results. This chapter also describes the tools utilised in the hardware implementation. More specifically, the objectives of this chapter are to:

- Provide hardware support for the Shift algorithm.
- Analyse and evaluate how close its performance corresponds to the expected results.
- Explain its hardware architecture requirements and present an estimation of the performance.

7.2 HARDWARE MODELLING

This section shows the architecture of the Shift algorithm where a 1st order hardware-modelling unit proves its hardware suitability. The model is coupled with an arithmetic coder module.

7.2.1 Design Tools

The SystemC modelling platform [SystemC00] from the Open SystemC Initiative (OSCI) is used for the hardware modelling. This platform allows creating system-level

designs in a C++ environment. SystemC is a C++ class library and a methodology that can be used to create cycle-accurate models of software algorithms, hardware architectures and interfaces of SoC (System on a Chip) and system-level designs [SystemC00]. Our system is compiled with the VC ++ compiler, version 6.0, on a Windows NT platform.

7.2.2 Assumptions

This hardware modelling is an implementation of the Shift algorithm simulated in C language and described in Chapter 6.

Order of the model:	1 st order PPMC model to simplify the implementation
Dictionary size:	has 4096 positions for 1 st order and 256 and 257 positions for 0 th and -1 st respectively
Block size:	4096 bytes, due to this size representing a typical packet size found in many computers and telecommunication systems
Data set:	Canterbury Corpus
Discarding policy:	Not required

An arithmetic coder was implemented as a separated module using the code from [Witten87], which was adapted to the new requirements for Shift model. It interacts with the model by encapsulating it into a sub-block so it can accept the output signals that the model produces.

7.2.3 Model Architecture

This section explains the architecture of the model. We implement a 1st order model to which we will refer from now on as 'hardware Shift model'. Both, compressor and decompressor were built and the compression results were verified against the C model described in Chapter 6.

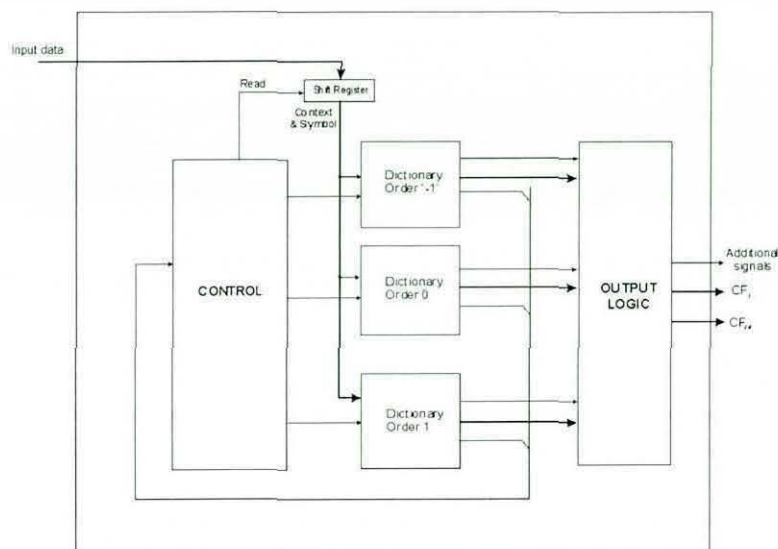


Figure 7.1 Architecture of the Shift compression model

Figure 7.1 illustrates the architecture of the compressor model: the bold lines indicate the flow of data and the others indicate the flow of control signals. Input data are entered to a shift register that assembles the context and the input symbol to produce the input for the dictionaries. When indicated by the control, this context and symbol are searched in parallel in the dictionaries. Each dictionary inputs the contexts according to its order. Signals indicating the result of the search operations are entered to the control and the cumulative frequency counts are transferred from the dictionaries to the output logic. If the search operation in a dictionary is not successful then the cumulative frequencies of the escape symbol are output. The output logic selects the best match and sends it to the coder together with other signals needed to codify them and form the compressed data.

There is one dictionary for every order of the model. Each of the dictionaries of order higher than -1^{st} have the architecture as shown in Figure 7.2. The dictionary of order -1 is simpler, it just contains the control logic and, instead of the memory block, an array with the frequencies of the symbols in cumulative form, where the index in the array indicates the symbol. Dictionaries of order 0^{th} and above contain a memory block and two registers as the figure shows. They are managed by simple control logic that indicates them when a searching or updating operation is to be performed. The memory block includes a CAM array to store the input data and two register files to

store the frequency counts and cumulative frequency counts of the symbols and their contexts. Search operations in the CAM array produce the address where the symbol is located. This address is then used to access the frequency counts from the arrays. The two registers, shown below the memory block in the figure, store the number of positions used in the dictionary and the match position of the symbol found respectively.

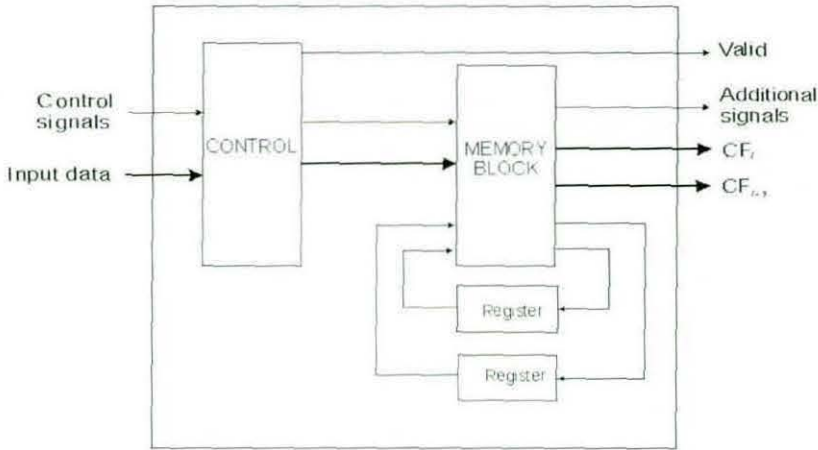


Figure 7.2 Dictionary architecture of shift model

The decompressor has similar architecture to the compressor, including its dictionaries. The task of the decompressor is more complex and requires more operations to perform it. These operations include serial and parallel searches. Serial searches are required when the model, inputting cumulative frequencies, looks for symbols. When the model is being updated, it requires searching for escape symbols and, in this case, when parallel searches in all the dictionaries are performed. Thus, the dictionaries in the decompressor perform the same operations as the compressor ones, just adapted for the decompressor requirements. Instead of searching for symbols, the decompressor looks for contexts to mark the valid ones from which the decoder can predict the incoming symbol.

The addition of one higher order in the model is straightforward, just one dictionary similar to the one of Figure 7.2 is required for every new order plus the appropriate signals to connect and extra wires for the control logic.

7.2.4 Test bench

Although *test bench* is the common term for referring to a test harness in the VHDL environment, we have adopted it. The test bench is used to exercise and verify the functional correctness of the hardware model in a simulation environment [Smith96].

The test bench, as well as the hardware Shift model, has been implemented in SystemC modelling platform.

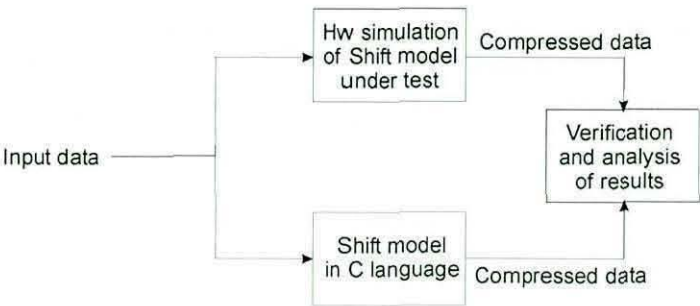


Figure 7.3 Flow of data in the test bench

Figure 7.3 shows the flow of data in the test bench where the output of the hardware Shift model is compared against the output obtained from the Shift model simulated in C language. Both outputs are the compressed data and must contain the same bit stream if the functionality of the hardware model is correct. It is worth mentioning that previously the Shift model, simulated in C language, was tested when the output of the decompressor matched the input data.

7.3 HARDWARE REQUIREMENTS

As seen in the previous section, the main component of the model is the memory block that contains a CAM array to store the symbols input, and two arrays of registers to store the statistics of these symbols. The architecture of the model, although simulated in behavioural form, was analysed to get an estimated number of gates required. The estimated gate count of the model is based on a 1st order model and is shown in Table

7.2. The estimated gate count was obtained with the help of Table 7.1 that shows the number of gates required per bit in every component.

Component	Implementation	Estimated gate count
CAM array	3x1 Mux, DFF, XOR, AND	15 gates
Register	FF	8 gates
Counter	FF, HA	12 gates
Subtractor	FA, NAND	10 gates

Table 7.1 Estimated size of compression components

The first column of the Table 7.1 shows the type of components used in the Shift model design. The next column shows the sub-components that integrate each component and the last column the estimated gate count per bit. From this table, the less common component is the CAM array. The architecture of the basic storage unit for a CAM array is illustrated in [Gooch96] and has been taken as the basis for the calculation of the gate count estimation of this component in Table 7.2.

The estimated gate count of the design of Figure 7.1 is shown in Table 7.2. The first column indicates the order of the contexts stored in a dictionary for which the components, in the second column, are required. The third column shows the size of the components, length and depth in case of CAM arrays, number and width of registers in case of frequency tables and number and width of other components.

The estimated of size of the compressor architecture is about 3 million NAND equivalent gates, from which most of the space is assigned to storage and updating of data. From the contribution figure in the table, it is clear that the components, which heavily affect the system size, are the memory and the arrays of the frequencies and cumulative frequencies. The parallel updating of cumulative frequency counts could simplify the complexity of the model by eliminating the frequency array, thus reducing its space requirements about 16%.

If devices with embedded memory were used for the hardware implementation, about one third of the total gate count must fit in this memory.

Order	Component	Implementation	Estimated gate count	Contribution
-1	CAM array	256 x 9 bits	18,432	0.56 %
0	CAM array	257 x 9 bits	34,695	1.05 %
	Frequency array	257 x 16 bits	32,896	1 %
	Cumulative frequency array	257 x 16 bits	32,896	1 %
	Shift logic, Adder/subtractor	257 x 16 bits	53,456	1.62 %
	CAM array	4096 x 18 bits	1,105,920	33.61 %
1	Frequency array	4096 x 16 bits	524,288	15.94 %
	Cumulative frequency array	4096 x 16 bits	524,288	15.94 %
	Shift logic, Adder/subtractor	4096 x 16 bits	851,968	25.9 %
	Mux	4096 x 9 bits	110,592	3.36 %
	Shift register	1 x 18 bits	144	< 0.01 %
Additional Logic	Control Unit	1 x 4 bits register	32	< 0.01 %
	Output Logic	1 x 9 bits register	72	< 0.01 %
	Subtractor	2 x 16 bits	320	0.01 %
Total Gate Size			3,289,999	

Table 7.2 Estimated system size based on a 1st order Shift model

If one higher order were added to the model, the size of the 1st order dictionary would be adjusted to 2K positions while the 2nd order would be 4K positions long. The reason, as explained in previous chapters, is that most of the matches are generated in the highest context, thus less space is required for one lower order, so, the sizes of the lower order dictionary may be even smaller. Thus, if a 2nd order is increased to the model, the overall gate count estimation for 1st order should increase about 1.5 million gates. That means that a 2nd order model would require about 4.5 million gates. The decompressor requires about the same number of gates as the compressor.

On average, 3.29 and 7.5 behavioural clock cycles are required per symbol for compressor and decompressor respectively. These figures were obtained dividing the number of behavioural clock cycles required to compress the sets of data in Canterbury Corpus by the number of symbols compressed. Note that the figures are simulation

times that do not necessarily correspond to machine cycles since synthesis tools may expand a behavioural cycle into several machine cycles.

From the total estimated gate size, this implementation approach is feasible to achieve using present day implementation technology. The system may be implemented in a single digital VLSI integrated circuit or in other technologies such as FPGAs, *e.g.* Xilinx FPGA Virtex-II family, that has up to 10 million usable gates.

Further work involves the synthesis of the SystemC model to directly produce a netlist without translating the model into a HDL language. This saves time by eliminating errors that may be introduced during translation and later may take significant time to track down.

7.4 SUMMARY

It was mentioned that the hardware implementation of the statistical PPMC compression model is impractical due to the nature of the algorithm. In the previous chapter we studied in detail the Shift model, derived from PPMC but suitable for hardware implementation. This chapter presented the hardware support for this Shift model.

The Shift model was designed with the aim of speeding up the compression process, not skimping gates in the design. The fulfilment of this design in SystemC language shows the suitability of the model for hardware implementation although unfortunately no speed measures were obtained since no real-time implementation was done.

According to the estimated gate counts of the model, it has been shown that the Shift model can be implemented in available digital devices.

CHAPTER 8

CONCLUSIONS

8.1 OBJECTIVES OF THE CHAPTER

This chapter concludes this thesis by summarising the main points and discussing whether the objectives have been achieved. Specifically, the objectives of the chapter are to:

- Review the objectives set out in Chapter 3.
- Summarise the conclusions of the investigations detailed in the previous chapters.
- Examine the strengths and limitations of the work.
- Outline further research areas that may be of interest.

8.2 REVIEW OF THE OBJECTIVES

This thesis tackles the problem of understanding how to design high-performance compression algorithms suitable for hardware implementation and to show that the knowledge gained from the thesis can be used to provide hardware support for an efficient compression algorithm.

From the review of data compression and compression implementations three key issues were identified as the lines of research in the pursuit of the objectives:

- Statistical compression algorithms, particularly the PPM class.
- Simplification of the PPMC algorithm for fast hardware implementation.
- Hardware support of this statistical algorithm.

No high-order statistical compression models have been implemented in hardware before, so it was necessary to gain a better understanding of the issues surrounding such models, their design and hardware architectures used in compression implementations of other algorithms that could be used in the implementation of statistical models. Thus, the following objectives were set for the work in this thesis:

- To identify the key computational requirements of efficient compression algorithms.
- To simplify the algorithmic processes and understand how they improve the performance of a compression system.
- To analyse the interaction and tradeoffs between algorithmic desired characteristics and hardware capabilities for ensuring the effective mapping of algorithmic computational requirements into compression architectures.

To fulfil these objectives, we analysed the issues involved and chose the investigations presented in detail in Chapter 3. Taking the PPMC model as the starting point of research, these investigations lead us to develop and simulate the compression model to identify its key computational requirements and to study whether other factors may impact its performance. Also, these simulations helped to observe if the model could be focused towards hardware implementation by using efficient hardware structures and simplifying its functionality. Then, further investigations and the integration of an entire system based on the previous results were followed doing software simulations to determine if the system could be able to reproduce the expected behaviour. Finally, hardware support was developed to determine if it would be possible to implement the system using present day implementation technologies.

8.3 CONCLUSIONS

In Chapters 1 and 2 we discussed the lack of a hardware implementation of high-performance statistical data compression algorithms and presented their potential benefits. From this, in Chapter 3, and in response to the objectives, we outlined the research investigations of this thesis that would lead us to develop such hardware implementation. The main conclusions drawn from these investigations are:

- The PPMC model is a very promising data compressor for hardware implementation as results of comparing it with commercial compression implementations that place it as the best performer.
- By reorganising and optimising the PPMC algorithm it is possible to produce a model that is suitable for hardware implementation.

From the statistical data compression investigation about the key computational requirements of PPMC compression model and some other factors that influence its compression performance we conclude that:

- The updating process of the PPMC model is the most computationally demanding, as it requires to compute cumulative frequency counts and total frequencies to estimate symbol probabilities. The operations used to compute such counts are complex and a large number of them have to be performed. Another computationally demanding process is searching, as it requires a large number of operations mainly in high order models. So, if these main features of the algorithm were simplified, practical hardware applications of this model could be implemented.
- Other factors such as the order of the model, the dictionary size, the input data block size and the discarding policy, that are not directly related with the algorithm itself but with the implementation, contribute in great part to the achievement of high performance in compression implementations. The higher the order of the model the better compression ratio but also the higher the space requirements. The

larger the size of the block to compress, the more accurate statistics the model can gather and thus the better the compression ratio. The simpler discarding policies the faster the model. So, the more resources we assign for the model and the simpler the operations to compute, the better overall performance will be achieved. Unfortunately practical implementations present space restrictions and care must be taken to find a balance between compression and space.

From the investigation about the simplification of the PPMC algorithm that could reduce the complexity of the hardware implementation we conclude that:

- The statistical nature of the PPMC algorithm requires a careful study when looking for some reorganisation of the model. Small changes in the model updating may lead to significant degradation of the compression ratio.
- The updating process may be simplified considerably at the cost of some degradation in compression ratio. We found a small degradation when compared with the PPMC model with the approximations that we investigated. These approximations were simple but failed at assigning probabilities to the symbols according to their occurrence in the input stream.
- The task of the arithmetic coder may be simplified and sped up by keeping constant and to a power of two the total frequency counts in the model. In this way, division operation may be substituted by simple shifts when computing symbol probabilities.

From the investigation about the Shift model implementation we conclude that:

- It is possible to implement practically a statistical model for lossless data compression that may be fitted in digital devices currently available.
- By keeping constant the total number of tokens assigned to contexts and redistributing them among the incoming symbols using a division operation, it is

possible to update accurately the model, leaving its performance very close to the PPMC model.

- It is difficult to compute symbol statistics as in the PPMC model without involving complex or large numbers of arithmetic operations, which may restrict its practical implementation. However, it is possible to find alternative implementations without compromising significantly the compression ratios.
- The use of tokens for the frequency counts allows parallel updating in the frequencies, although it adds some complexity in the hardware design.
- Approximating the number of tokens to assign to the incoming symbol (*TIS*) yields, in most of the cases, a small degradation in compression ratio. However, it represents great savings in hardware complexity.
- Storing contexts in separated dictionaries according to their order has the advantage of reducing complexity, mainly when dealing with discarding policies if other than deleting the complete dictionary were used. Although, depending of their size, this also may increase the space requirements.

8.4 MEASUREMENTS OF SUCCESS

After reviewing the conclusions of the chapters, it is clear that all the objectives set out in the beginning of the thesis and in chapter three have been met. Specifically, the work in this thesis has resulted in the following:

- The work sets a test bench of commercially available lossless data compressors that may be used for benchmarking other possible compression implementations.
- We stated in Chapter 2 that not many hardware implementations of statistical data compression algorithms have been developed and even less statistical models. In fact, the only statistical models developed in hardware have been of 0th order and

have been coupled generally with the arithmetic coder. In this work we have studied higher-order statistical models to be implemented in hardware. By simulating and analysing the model, we have identified the requirements of the models as well as other factors that have an impact in their compression performance.

- Through experimentation it was shown how the PPMC compression model is implemented using efficient data structures utilised for hardware implementations.
- It was showed how the PPMC algorithm could be restructured and reorganised for its implementation in digital technologies. As a result of this reorganisation, a novel variant of PPMC, the Shift algorithm, was introduced. It exploits hardware architectures to speed up the compression process by substituting complex operations required to compute symbol probabilities by simple shifts. Although the new algorithm requires larger number of operations, they can be done in parallel, which improves compression speed.
- It was also shown how Shift algorithm was capable of reproducing the behaviour of the PPMC algorithm and its feasibility for hardware implementation. This work has also shown how the algorithm performs if space restrictions have to be imposed.
- The cycle-accurate implementation of the Shift algorithm has been proved to be feasible to implement while maintaining its excellent compression performance. This class of algorithms can be implemented with current technology and, as technology improves, the limits set by the implementation size will be reduced, making higher-order models of this algorithm even easier to implement in hardware.

8.5 SHORTCOMINGS OF THE WORK

Despite the good results obtained, this research has some limitations:

- High-order PPMC models were studied. However, when all the knowledge gained from the experiments was ‘consolidated’ in the Shift model, it considered just the 2nd order for further experiments. Once the experiments were finished, we proceeded to simulate in SystemC a 1st order model since this was enough to demonstrate the feasibility of its implementation. And although estimations of the requirements for 2nd order model were given, it was not implemented.
- Considering the limitation of time, we developed an approximation to compute the tokens to assign to the incoming symbol, which simplifies considerably the complexity of the updating process of the model. However, we did not study how to approximate the computation of cumulative frequencies, which may cause the system to be slow in this part due to possible serial computations of these counts.
- The design was simulated under ideal conditions but was not tested in real-time. Also, most of the simulations were measured in terms of compression ratios and although it could be deducted whether or not some models could compress fast, the compression speed was not measured.
- At the time of writing, tools for synthesis of SystemC were not available. If they were, the model could be synthesised to provide speed information and to give better and more accurate compression performance and hardware requirements.

8.6 FUTURE INVESTIGATIONS

In the work presented in this thesis, the statistical PPMC compression algorithm was modelled in hardware. However, some issues, as how to update cumulative frequency counts in parallel was not addressed. Therefore, the natural progression of this work would be to implement such a system so the compressor may simplify the hardware requirements by 16 % as observed in Chapter 7.

Other compression algorithms of the PPM class [Teahan95 and Bloom 96b] employ a technique called *local order estimation* to predict symbols with the context that provide the highest probability, thus producing further compression. In the Shift model the searches are in parallel in all the dictionaries of the model. Thus, the implementation of this technique requires little work, just some extra comparisons are required and the logic to implement it. We consider that time and space requirements would not be affected and the improvements in compression ratio could be considerable.

As the thesis gave hardware support of Shift model by simulating it in SystemC language, a natural step to follow is to provide a netlist and obtain speed information and evaluate more accurately its compression speed and hardware requirements.

8.7 SUMMARY

This chapter has presented the main conclusions that were reached as a result of the investigations described in this thesis. We have shown how these objectives have been achieved. The final conclusion was then drawn from these, which proved the feasibility of implementing the PPMC model in hardware.

Within the bounds of these investigations the thesis has achieved its objective. The strength and contribution to knowledge of this thesis lies in the quantitative conclusions, which enable us to identify under what conditions Shift model offers performance benefits. Suggestions were made on other possible future investigations that would extend this work. Furthermore, they may help designers in implementing statistical compression models in future compression chips.

APPENDIX A

DATA SETS

A.1 CANTERBURY CORPUS

Canterbury Corpus [Arnold97] is a data set for evaluating lossless data compression methods. It is freely available by anonymous ftp from <http://corpus.canterbury.ac.nz>.

The paper ‘*A Corpus for the Evaluation of Lossless Compression Algorithms*’ [Arnold97] explains how and why these files were chosen. The corpus consist of 11 files, which range in size from 3K to 1,029K, from C and LISP source code, html files, technical writings and text files. Table A-1 lists the files in the corpus, their size and their category. The average size of a file is 255,564 symbols (characters) and the total number of symbols is 2,811,210.

File	Category	Size (Bytes)
alice29.txt	text (English text)	152,089
asyoulik.txt	play (Shakespeare)	125,179
cp.html	HTML	24,603
fields.c	Csrc (C source)	11,150
grammar.lsp	lisp (LISP source)	3,721
kennedy.xls	Excl (Excel Spreadsheet)	1,029,744
lcet10.txt	tech (Technical writing)	426,754
plrabn12.txt	poem (Poetry)	481,861
ptt5	fax (CCITT test set)	513,216
sum	SPRC (SPARC Executable)	38,240
xargs.1	man (gnu manual page)	4,227
TOTAL		2,811,210

Table A-1. Files in Canterbury Corpus

A random collection of larger files consists of 3 files of textbooks and the genome of the E.Coli bacterium. These files are “useful for algorithms that can't ‘get up to speed’ on smaller files, and the other collections may be useful for particular file types”, according to the information provided in the mentioned web page. Table A-2 lists the files in the corpus, the total number of symbols is 11,159,482 and it is likely that more files are added to this collection.

File	Category	Size (Bytes)
E.Coli	Complete genome of the E.Coli bacterium	4,638,690
bible.txt	The King James version of the bible	4,047,392
world192.txt	The CIA world fact book	2,473,400
TOTAL		11,159,482

Table A-2. Files in Canterbury Corpus

A.2 MEMORY DATA SET

Memory Data Set contains memory data samples from 8 applications as well as the operating system itself of about 80 MB. A detailed explanation about how the files were obtained is in [Kjelso97] from where the table A-3 is taken. The set contains data from the SunOS operating system, Netscape, Emacs, Textedit, Ghostview, Xman, and Matlab. The Vlabplus and Logsyn are part of Intergraph's commercial CAE tool suit.

File	Category	Size (Bytes)
SunOS	Approximation to the operating system memory resident working set	10,842,112
Netscape	Captured during use of Netscape WWW browser, after some 'net-surfing' activity	7,172,096
Emacs	Captured during use of Emacs text editor, with a few buffers open	6,111,232
Textedit	Acquired during use of Textedit, having a small C source file open	3,223,552
Ghostview	Sample obtained from Ghostview postscript viewer, with a technical paper open	5,160,960
Xman	Captured from Unix manual-page viewer	3,145,728
Matlab	Captured from Matlab whilst running a benchmark program	12,025,856
Vlabplus	Obtained during execution of netlisting and spicer simulation of a 4-bit parallel multiplier	8,769,536
Logsyn	Collected during logic synthesis area optimisation of a 4-bit parallel multiplier	20,336,640
TOTAL		76,787,712

Table A-3. Files in Memory Data Set

A.3 THESIS DATA SET

Thesis data set is a collection of 65 files, which range in size from 3K to 450K, from audio, images, object and text files. This set was obtained from [Gooch96]. Tables A-4, A-5, A-6 and A-7 list the audio, object, image and text files respectively. All the tables show the corresponding files, its category and size. The total number of characters is 8,045,584.

Files	Category	Size (Bytes)
aud00.dat	Male voice. 6 seconds, mono, 8 bit a-law encoding	48,032
aud01.dat	Female voice. 7 seconds, mono, 8 bit u-law encoding	56,032
aud02.dat	Male voice. 15 seconds, mono, 4 bit g721 encoding	60,032
aud03.dat	Male voice. 9 seconds, mono, 8 bit pcm encoding	72,032
aud04.dat	Classical music. 12 sec., mono, 8 bit a-law encoding	96,032
aud05.dat	Female voice. 13 seconds, mono, 8 bit pcm encoding	104,032
aud06.dat	Music. 29 seconds, mono, 4 bit g721 encoding	116,032
aud07.dat	Classical music. 8 seconds, stereo, 8 bit u-law encoding	128,032
aud08.dat	Female voice. 17 seconds, mono, 8 bit a-law encoding	136,032
aud09.dat	Music. 20 seconds, mono, 8 bit u-law encoding	160,032
aud10.dat	Female voice. 21 seconds, mono, 8 bit pcm encoding	168,032
aud11.dat	Male voice. 25 seconds, mono, 8 bit u-law encoding	200,032
aud12.dat	Classical music, 15 sec., stereo, 8bit a-law encoding	240,032
aud13.dat	Classical music. 18 sec., stereo, 8 bit pcm encoding	288,032
aud14.dat	Music. 18 seconds, stereo, 8 bit a-law encoding	288,032
aud15.dat	Music. 19 seconds, stereo, 8 bit u-law encoding	304,032
aud16.dat	Music. 23 seconds, stereo, 8 bit pcm encoding	368,032
TOTAL		2,832,544

Table A-4. Audio files in Thesis Data Set, sampling rate 8 KHz

File	Category	Size (Bytes)
img00.dat	Raquel Welsh - 320x200 1 bit Portable Bitmap Image (pbm)	8,012
img01.dat	United Nations Flag – drawperfect v1.1 format	11,773
img02.dat	Outline map of the world - drawperfect v1.1 format	15,161
img03.dat	Plan of Welsh farmhouse - 640x400 1 bit Portable Bitmap Image (pbm)	32,012
img04.dat	Kate Bush album cover - 320x200 GIF format (b&w 64 shades)	51,184
img05.dat	Photograph of a mountain valley - 533x759 JPEG format	76,407
img06.dat	Photograph of an oriental womans face - 300x350 8 bit Portable Greymap Image (pgm)	105,015
img07.dat	Photograph of a castle - 950x800 JPEG format	119,969
img08.dat	Photograph of a cablecar and scenery - 400x400 8 plane sun rasterfile	160,686
img09.dat	Photograph of a house - 256x256 24 plane sun rasterfile	196,640
img10.dat	Photograph of football action - 512x480 8 plane sun rasterfile	245,792
img11.dat	Photograph of a house on a bleak hillside - 500x500 8 bit Portable Greymap Image (pgm)	250,016
img12.dat	Mona Lisa - 500x650 GIF format	263,472
img13.dat	Photograph of a yacht race - 250x400 24 plane sun rasterfile	300,032
img14.dat	Photograph of a cornfield and tractor - 440x260 24 bit Portable Pixmap Image (ppm)	343,216
img15.dat	Photograph of an F16 flying over mountains - 500x300 24 bit Portable Pixmap Image (ppm)	450,016
TOTAL		2,629,403

Table A-5. Image files in Thesis Data Set

File	Category	Size (Bytes)
obj00.dat	Unix DU command (displays disk blocks per file or directory). Sun Microsystems release 4.1 (09/09/87).	4,816
obj01.dat	Image edge detection program compiled on 80386 architecture (from Pascal source).	7,385
obj02.dat	DOS V5.1 edlin command (text editing program) on 80386 architecture.	12,642
obj03.dat	Unix LS command (lists contents of a directory). Sun Microsystems release 4.1 (02/10/89).	13,336
obj04.dat	Image display program compiled on 80386 architecture (from Pascal source).	25,281
obj05.dat	DOS V6.0 fdisk command (disk formatting program) on 80486 architecture.	29,333
obj06.dat	Huffman code length calculation program compiled on Sun SPARC-1 architecture (from C source).	32,768
obj07.dat	Unix SED command (stream editor). Sun Microsystems release 4.1 (03/02/89).	40,960
obj08.dat	DOS V5.1 smartdrv command (RAM disk program) for 80386 architecture.	42,073
obj09.dat	Lossless data compression algorithm (bstw001 by Mark Gooch) compiled on Sun SPARC-1 architecture (from C source).	49,152
obj10.dat	Windows V3.1 calendar program (calendar display and diary) on 80386 architecture.	59,824
obj11.dat	Image rotation program (pnmrotate from pbmplus toolkit) compiled on Sun SPARC-1 architecture (from C source).	65,536
obj12.dat	Fractal landscape generation program (ppmforge from pbmplus toolkit) compiled on Sun SPARC-1 architecture (from C source).	81,920
obj13.dat	Windows V3.1 cardfile program (electronic card storage system) on 80386 architecture.	93,184
obj14.dat	Power circuits analysis program compiled on Hewlett Packard HP9000/800 system (from FORTRAN source).	139,264
obj15.dat	Unix TAR command (tape archiving program). Sun Microsystems release 4.1 (16/02/88).	147,456
obj16.dat	Image type conversion program (convert from ImageMagick toolkit) compiled on Sun SPARC-LX architecture (from C source).	348,388
TOTAL		1,193,318

Table A-6. Object files in Thesis Data Set

File	Category	Size (Bytes)
txt00.dat	C source code for entropy calculation (heavily commented)	3,648
txt01.dat	Text of common provisions from the Maastricht Treaty	4,476
txt02.dat	Text of Desert Storm speech by George Bush	8,407
txt03.dat	Text of Antarctic survey trip by Peter Amati	12,409
txt04.dat	C source code for bstw data compression algorithm (heavily commented)	16,013
txt05.dat	Text of Examination Regulations for LUT	22,211
txt06.dat	A translation of the Magna-Carta by Gerald Murphy	28,469
txt07.dat	A document on Networking Standards by A. M. Rutkowski	31,231
txt08.dat	Sun Microsystems manual entry for 'make' command (release 4.1 , 15/09/89)	53,731
txt09.dat	Text of Protocols from the Maastricht Treaty	97,390
txt10.dat	Text and graphics of a report on storage media (in WordPerfect format) by Mark Gooch	149,006
txt11.dat	A Christmas Carol by Charles Dickens	156,583
txt12.dat	Alice Through The Looking Glass by Lewis Carol	156,635
txt13.dat	Text of 'A Vision of Change for America' by Bill Clinton	306,132
txt14.dat	The War of the Worlds by Herbert George Wells	343,978
TOTAL		1,390,319

Table A-7. Text files in Thesis Data Set

REFERENCES

- [Aberg97] Aberg J, Shtarkov Y. M., Smeets B. J. M. (1997) 'Towards Understanding and Improving Escape Probabilities in PPM', Proceedings of the IEEE Data Compression Conference, J. Storer editor, Los Alamitos CA, March, pp 22-31.
- [Abramson63] Abramson N. (1963) 'Information Theory and Coding', McGraw-Hill, New York.
- [Arnold97] Arnold R. and Bell T. (1997) 'A Corpus for the Evaluation of Lossless Compression Algorithms', Proceedings of the IEEE Data Compression Conference, J. Storer editor, Los Alamitos CA, March, pp 201-210. The corpus can be obtained from <http://corpus.canterbury.ac.nz>
- [Arps88] Arps R. B., Truong T. K., Lu D. J., Pasco R. C. and Friedman T. D. (1988), 'A Multi-Purpose VLSI Chip for Adaptive Data Compression Bilevel Images', IBM Journal of Research and Development, Vol. 32, No. 6, November, pp 775 - 795.
- [Ashley00] Ashley J., Bernal M.-P., Burr G. W., Coufal H., Guenther H., Hoffnagle J. A., Jefferson C. M., Marcus B., Macfarlane R. M., Shelby R. M. and Sincerbox G. T. (2000) 'Holographic Data Storage', IBM Research and Development, Vol. 44, No. 3, May, pp 341 - 368.
- [Azgomi99] Azgomi Sherri (1999) 'Content-Addressable Memory (CAM) and its Applications', Electronic Engineering, Vol. 71, August, pp 23-28.

- [Bell90] Bell T. J., Cleary J. G. and Witten I.H. (1990) 'Text Compression', Englewood Cliffs, NJ: Prentice Hall.
- [Bell97] Bell T. (1997) 'Experimental Results on the Canterbury Corpus', corpus.canterbury.ac.nz/results/cantrbry.html, first results from 1997, last checked November 2000.
- [Benschop96] Benschop L. C. (1996) '100 Mbit per Second VLSI Implementation of Sliding Window Coding', Proceedings of the IEEE Data Compression Conference, J. Storer editor, Los Alamitos CA, March, p. 424.
- [Bentley86] Bentley J.L., Sleator D.D., Tarjan R.E. and Wei V.K. (1986) 'A Locally Adaptive Data Compression Algorithm', Communications of the Association for Computing Machinery, Vol. 29, No. 4, April, pp 320-330.
- [Bloom96a] Bloom C. R. (1996) 'LZP: A New Data Compression Algorithm', Proceedings of the IEEE Data Compression Conference, J. Storer editor, Los Alamitos CA, March, p. 425.
- [Bloom 96b] Bloom C. R. (1996) 'New Techniques in Context Modeling and Arithmetic Encoding', IEEE Data Compression Conference, J. Storer editor, Los Alamitos CA, March, p. 426.
- [Bloom96c] Bloom C. R. (1996) PPMZ source code available from www.programmersheaven.com/zone22/cat166/2217.htm, last checked January 18th, 2001.
- [Bongjin98] Bongjin J. and Burleson W. P. (1998) 'Efficient VLSI for Lempel-Ziv Compression in Wireless Data Communication Networks', IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 6, No. 3, September, pp 475-483.

- [Bucholtz00] Bucholtz C. (2000) 'Stretching Storage by Shrinking Files', HP World Magazine, February. Also available from www.solution-soft.com, last checked February 7th, 2001.
- [Bunton92] Bunton S. and Borriello G. (1992) 'Practical Dictionary Management for Hardware Data Compression', Communications of the Association for Computing Machinery, Vol. 35, No. 1, January, pp 95-104.
- [Bunton97] Bunton S. (1997) 'Semantically Motivated Improvements for PPM Variants', Computer Journal, Vol. 40, No. 2/3, pp 76-93.
- [Burrows94] Burrows M. and Wheeler D. J. (1994) 'A Block-Sorting Lossless Data Compression Algorithm', Technical Report SRC 124, Digital Systems Research Center, Palo Alto, CA., May.
- [Chevion91] Chevion D., Karnin, E. D. and Walach E., (1991) 'High Efficiency, Multiplication Free Approximation of Arithmetic Coding', IEEE Data Compression Conference, J. Storer editor, Los Alamitos CA, March, pp 43-52.
- [Cisco97] Cisco IOS Data Compression, White Paper, Cisco Systems, 1997. Available from www.cisco.com.
- [Cleary84] Cleary J. G. and Witten I. H. (1984) 'Data Compression using Adaptive Coding and Partial String Matching', IEEE Transactions on Communications, Vol. 32, No. 4, April, pp 396-402.
- [Cleary93] Cleary J. G. and Teahan W. J. (1993) 'Unbounded Length Contexts for PPM', Computer Journal, Vol. 36, No. 5, pp 1-9.

- [Cleary95] Cleary J. G. and Teahan W. J. (1995) 'Experiments on the Zero Frequency Problem', IEEE Data Compression Conference, J. Storer editor, Los Alamitos CA, March, pp 43-52.
- [Comenford00] Comenford R. (2000) 'Magnetic Storage: The Medium that wouldn't Die', IEEE Spectrum, Vol. 37, No. 12, December, pp 36-39.
- [Cormack87] Cormack G.V. and Horspool R.N. (1987) 'Data Compression Using Dynamic Markov Modeling', Computer Journal, Vol. 30, No. 6, pp. 541-550.
- [Craft98] Craft D. J. (1998) 'A Fast Hardware Data Compression Algorithm and some Algorithmic Extensions', IBM Journal of Research and Development, Data Compression technology in ASIC cores, Vol. 42, No. 6.
- [DCP] Information available though the world wide web at www.datacompression.com
- [Effros00] Effros M. (2000) 'PPM Performance with BTW Complexity: A New Method for Lossless Data Compression', Proceedings of the IEEE Data Compression Conference, J. Storer editor, Los Alamitos CA, March, pp 203-212.
- [Elias87] Elias P. (1987) 'Interval and Recency Rank Source Coding: Two On-Line Adaptive Variable-Length Schemes', IEEE Transactions on Information Theory, Vol. 33, No. 1, January, pp 3-10.
- [Fenwick94] Fenwick P. M. (1994) 'A New Data Structure for Cumulative Frequency Tables', Software – Practice and Experience, Vol. 24, No. 3, March, pp 327-336.

- [Fenwick95] Fenwick P. M. (1995) 'A New Data Structure for Cumulative Probability Tables: An Improved Frequency-to-Symbol Algorithm', Technical Report 110, Department of Computer Science, The University of Auckland, New Zealand.
- [Fenwick96a] Fenwick P. M. (1996) 'Block Sorting Text Compression – Final Report', Technical Report 130, Department of Computer Science, University of Auckland, New Zealand, April.
- [Fenwick96b] Fenwick P. M. (1996) 'Symbol Ranking Text Compression', Technical Report 132, Department of Computer Science, University of Auckland, New Zealand, June.
- [Fenwick98] Fenwick P. M. (1998) 'Symbol Ranking Text Compressors: Review and Implementation', Software – Practice and Experience, Vol. 28, No. 5, April, pp 547 – 559.
- [Gajski97] Gajski D. D. (1997) 'Principles of Digital Design', Prentice Hall International, USA.
- [Gollomb66] Gollomb S. W. (1966) 'Run-Length Encodings', Correspondence, IEEE Transactions on Information Theory, Vol. 12, July, pp 399-401.
- [Gooch96] Gooch M. (1996) 'High Performance Lossless Data Compression Hardware', PhD Thesis, Loughborough University, UK, August.
- [Held96] Held G. (1996) 'Data and Image Compression, Tools and Techniques', John Wiley and Sons Ltd, Fourth edition, Great Britain.
- [Hifna] Hifn application notes 'How LZS Compression works', www.hifn.com.

- [Hifnb] Hifn MPPC-386 Data Sheet, Version 6.0, Data Compression Software.
- [Hifn01] Hifn (2001) 'The First Book of Compression and Encryption', White Paper, Available from www.hifn.com.
- [Hirschberg92] Hirschberg D. S. and Lelewer D. A. (1992) 'Context Modeling for Text Compression', Image and Text Compression, J. A. Storer, ed., Kluwer Academic Publishers, Norwell, MA, pp 85-112.
- [Holtz93] Holtz K. (1993) 'The Evolution of Lossless Data Compression Techniques', WESCON Conference, Vol. 37, pp 140-145.
- [Horowitz95] Horowitz E., Sahni S. and Mehta D. (1995) 'Fundamentals of Data Structures in C++', Computer Science Press, United States of America.
- [Hovingh01] Hovingh R. (2001) 'Toward eDiskTM, Expanding Capacit of a Macintosh hard Disk', White paper from ns1.pressgo.net/edtechnology.htm. Last checked Sept. 2001.
- [Howard93a] Howard P. G. (1993) 'Design and Analysis of Fast Text Compression Based on Quasi-Arithmetic Coding', IEEE Data Compression Conference, J. Storer editor, Los Alamitos CA, March, pp 98-107.
- [Howard93b] Howard P. G., (1993) 'The Design and Analysis of Efficient Lossless Data Compression Systems', Report CS-93-28 and PhD. Thesis, Department of Computer Sciences, Brown University, Providence, Rhode Island, June.

- [Howard94] Howard P. G. and Vitter J. S. (1994) 'Arithmetic Coding for Data Compression', Proceedings of IEEE, Vol. 82, No. 6, June, pp 857-865.
- [Hsieh98] Hsieh M.-H. and Wei C.-H. (1998) 'An Adaptive Multialphabet Arithmetic Coding for Video Compression', IEEE Transactions on Circuits and Systems for Video Technology, Vol. 8, No. 2, April, pp 130-137.
- [Huffman52] Huffman D.A. (1952) 'A Method for the Construction of Minimum-Redundancy Codes', Proc. Institute of Electrical and Radio Engineers, Vol. 40, No. 9, September, pp 1098-1101.
- [IBM01] Information from IBM Almaden Computer Science Research www.almaden.ibm.com/cs/compression/. Last checked Sept. 2001
- [Itagaki00] Itagaki S. and Yokoo H. (2000) 'PPM*-Style Context Sorting Compression Method using a Prefix List', Proceedings of the IEEE Data Compression Conference, J. Storer editor, Los Alamitos CA, March, p. 556.
- [Jiang94] Jiang J. and Jones S., (1994) 'Parallel Design of Arithmetic Coding', Proceedings IEE, Part E, Vol. 141, November, pp 327-333.
- [Jones92] Jones S. (1992) '100 Mbit/s Adaptive Data Compressor Design using Selectively Shiftable Content-Addressable Memory', Proceedings IEE Part G, Vol. 139, No. 4, August, pp 498-502.
- [Jones00] Jones S. (2000) 'Partial-Matching Lossless Data Compression Hardware', IEE Proceedings Comput. Digit. Tech. Vol. 147, No. 5, September, pp 320-334.

- [Jou99] Jou J.-M. and Chen P.-Y. (1999) 'A Fast and Efficient Lossless Data-Compression Method', IEEE Transactions on Communications, Vol. 47, No. 9, September, pp 1278-1283.
- [Kim95] Kim Y.-J., Kim K.-S. and Choi K.-Y. (1995) 'Efficient VLSI Architecture for Lossless Data Compression', IEE Electronics Letters, Vol. 31, No. 13, June, pp 1053 – 1054.
- [Kjelsø96] Kjelsø M., Gooch M. and Jones S. (1996) 'The Design and Performance of a Main Memory Hardware Compressor', Proceedings 22nd Euromicro Conference, IEEE Computer Society Press, September, pp 423-430.
- [Kjelsø97] Kjelsø M. (1997) 'A Quantitative Evaluation of Data Compression in the Memory Hierarchy', PhD Thesis, Loughborough University, UK, April.
- [Knuth97] Knuth, D. E. (1997) 'The Art of Computer Programming', 3rd edition, Vol.1 Fundamental algorithms, Publisher Reading Mass, Harlow: Addison-Wesley.
- [Knuth98] Knuth, D. E. (1998) 'The Art of Computer Programming', 3rd edition, Vol. 3, Sorting and searching, Publisher Reading Mass, Harlow: Addison-Wesley.
- [Kuang98] Kuang S.-R., Jou J.-M. and Chen Y.-L. (1998) 'The Design of an Adaptive On-Line Binary Arithmetic-Coding Chip', IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, Vol. 45, No. 7, July, pp 693-706.
- [Langdon82] Langdon Jr. G.G. and Rissanen J. (1982) 'A Simple General Binary Source Code', IEEE Transactions on Information Theory, Vol. 28, No. 5, pp 800-803.

- [Larsson99] Larsson N. J. and Moffat A. (1999) 'Offline Dictionary-based Compression', Proceedings of the IEEE Data Compression Conference, March, pp 296-305.
- [Lee95] Lee C-Y., and Yang, R-Y. (1995) 'High Throughput Data Compressor Designs using Content Addressable Memory', IEE Proceedings on Circuits Devices and Systems, Vol. 142, No. 2, pp 69-73.
- [Lee97] Lee H.-Y., Lan L.-S., Sheu M.-H. and Wu C.-H. (1997) 'A Parallel Architecture for Arithmetic Coding and its VLSI Implementation', 39th Midwest Symposium on Circuit and Systems, pp. 178-191.
- [Lei95] Lei, S.-M. (1995) 'Efficient Multiplication-Free Arithmetic Codes', IEEE Transactions on Communications, Vol. 43, No. 12, December, pp 2950-2958.
- [Liu94] Liu L.-Y., Wang J.-F., Wang R.-J. and Lee J.-Y. (1994) 'CAM-Based VLSI Architectures for Dynamic Huffman Coding', IEEE Transactions on Consumer Electronics, Vol. 40, No. 3, August, pp 282 - 289.
- [Mello] Mello J. 'Importance of Data Compression in Branch Office Networks', White Paper from Motorola Information Systems Group, Network Systems Division. Available from www.rycom.ca/solutions/whitepapers/motorola/importance_data_compression.htm.
- [Moffat89] Moffat A. (1989) 'Word-based Text Compression', Software-Practice and Experience, Vol. 19, No. 2, February, pp 185-198.

- [Moffat90] Moffat A. (1990) 'Implementing the PPM Data Compression Scheme', IEEE Transactions on Communications, Vol. 38, No. 11, November, pp 1917-1921.
- [Moffat98] Moffat A. (1998) 'Arithmetic Coding Revisited', ACM Transactions on Information Systems, Vol. 16, No. 3, July, pp 256-294.
- [Morimoto94] Morimoto K., Iriguchi H. and Aoe J.-I. (1994) 'A Method of Compressing Trie Structures', Software – Practice and Experience, Vol. 24, No. 3, March, pp 265-288.
- [Nelson91] Nelson M. (1991) 'Arithmetic Coding + Statistical Modeling = Data Compression', Dr. Dobbs, November.
- [Nelson96] Nelson M. (1996) 'Data Compression with the Burrows-Wheeler Transform', Dr. Dobbs, September, pp 46, 48-50.
- [Nunez01] Nunez J.L., Feregrino C., Jones S. and Bateman S., 'X-MatchPRO: A ProASIC-Based 200 Mbytes/s Full-Duplex Lossless Data Compressor', Proceedings of the 11th International Conference FPL 2001, Lecture Notes in Computer Science, Springer, pp. 613-617, August, 2001.
- [Pawlikowski95] Pawlikowski K., Bell T., Emberson H. and Ashton P. (1995) 'Compression of Data Traffic in Packet-based LANs', Proceedings of the IEEE Data Compression Conference, p 430, March.
- [Pennebaker88] Pennebaker, W. B., Mitchell, J. L., Langdon, G. G., and Arps, R.B. (1988) 'An Overview of the Basic Principles of the Q-Coder Adaptive Binary Arithmetic Coder', IBM Journal of Research and Development, Vol. 32, No. 6, November, pp 717-726.

- [Printz93] Printz H. and Stubbley P. (1993) 'Multialphabet Arithmetic Coding at 16 Mbytes/sec', Proceedings of the IEEE Data Compression Conference, pp 296-305, March.
- [Ranganathan93] Ranganathan, S. and Henriques, S. (1993) 'High-speed VLSI Designs for Lempel-Ziv-based Data Compression', IEEE Transactions on Circuits and Systems.-II: Analog. Digit. Signal Process., Vol. 40, No. 2, February, pp 96-106.
- [Rissanen81] Rissanen J.J. and Langdon, G.G. (1981) 'Universal Modeling and Coding', IEEE Transactions on Information Theory, Vol. 27, No. 1, January, pp 12-23.
- [Rissanen89] Rissanen J. and Mohiuddin K. M. (1989) 'A Multiplication-Free Multialphabet Arithmetic Code', IEEE Transactions on Communications, Vol. 37, No. 2, February, pp 93-98.
- [Salomon98] Salomon D. (1998) 'Data Compression, the Complete Reference', Springer.
- [Schindler97] Schindler M. (1997) 'A Fast Block-sorting Algorithm for Lossless Data Compression', Proceedings of the IEEE Data Compression Conference, J. Storer editor, Los Alamitos CA, March, p. 469. A technical report is available from www.compressconsult.com/szip.
- [Shannon48] Shannon C.E. (1948) 'A Mathematical Theory of Communications', Bell System Technical Journal, Vol. 27, July, pp 398-403.
- [Smith96] Smith D. J. (1996) 'HDL Chip Design: A Practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs using VHDL or Verilog', Doone Publications, USA.

- [Smith99] Smith S. W., (1999) 'The Scientist and Engineer's Guide to Digital Signal Processing', Second edition, California Technical Publishing, USA.
- [SystemC00] SystemC User's Guide. Version 1.0, Synopsys, Inc., Coware, Inc., and Frontier Design, Inc., 2000. Available from www.systemc.org
- [Teahan95] Teahan W. J. (1995) 'Probability Estimation for PPM', Proc. Second New Zealand Computing Sciences Research Students' Conference (NZCSRSC), April.
- [Teuhola93] Teuhola J. and Raita T. (1993) 'Application of a Finite-State Model to Text-Compression', The Computer Journal, Vol. 36, No. 7, pp 607-614.
- [Thomborson92] Thomborson, C. (1992) 'The V.42bis Standard for Data-Compressing Modems', IEEE Micro, October, pp 41-53.
- [Tong96] Tong Lai Yu, (1996) 'Dynamic Markov Compression', Dr Dobb's Journal, pp 30-31, January.
- [UC-ICT] White paper on the UC-Xpress data compressor by ICT (Intelligent Compression Technologies). Available from www.ictcompress.com.
- [V.42bis] Recommendation V.42 bis, (01/90), 'Data Compression Procedures for Data Circuit Terminating Equipment (DCE) using Error Correction Procedures, CCITT (ITU).
- [Wang00] Wang P. (2000) 'Understanding Online Archiving', Computer Technology Review, Vol. XX, No. 1, January. Also available from the World Wide Web at www.solution-soft.com, last checked February 7th, 2001.

- [Welch84] Welch T.A. (1984) 'A Technique for High-Performance Data Compression', IEEE Computer, Vol. 17, No. 6, June, pp 8-19.
- [Williams91a] Williams R. (1991) 'An Extremely Fast ZIV-Lempel Data Compression Algorithm', Proceedings of the 1991 Data Compression Conference, J. Storer ed., Los Alamitos, CA, IEEE Computer Society Press, pp 362-371.
- [Williams91b] Williams R. 'LZRW4', www.ross.net/compression/lzrw4.html, last checked January 23, 2001.
- [Williams93] Williams R. (1993) 'Adaptive Data Compression', Kluwer Academic Publishers, Second Printing, USA.
- [Witten87] Witten I. H, Neal Radford M. and Cleary John G. (1987) 'Arithmetic Coding for Data Compression', Communications of the Association Computing Machinery, Vol. 30, No. 6, June, pp 520-540.
- [Witten91] Witten I. H and Bell T. C. (1991) 'The Zero-Frequency Problem: Estimating the Probabilities of Novel Events in Adaptive Text Compression', IEEE Transactions on Information Theory, Vol. 37, No. 4, July, pp 1085-1094.
- [Ziviani00] Ziviani N., Silva de Moura E., Navarro G. and Baeza-Yates R. (2000) 'Compression: A Key for Next-Generation Text Retrieval Systems', IEEE Computer, November, pp 37-44.
- [Ziv77] Ziv, J. and A. Lempel (1977) 'A Universal Algorithm for Sequential Data Compression', IEEE Transactions on Information Theory, Vol. 23, No. 3, May, pp 530-536.

- [Ziv78] Ziv, J. and A. Lempel (1978) 'Compression of Individual Sequences via Variable-Rate Coding', IEEE Transactions on Information Theory, Vol. 24, No. 5, September, pp 530-536.

PUBLICATIONS

Nunez J.L., **Feregrino C.**, Bateman S. and Jones S., *The X-MatchLITE FPGA-based data compressor*, Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays, Monterey, CA USA, February, 1999.

Nunez J.L., **Feregrino C.**, Bateman S. and Jones S., *The X-MatchLITE FPGA-Based Data Compressor*, Proceedings of the 25th EUROMICRO Conference, Digital Systems Design: Architectures, Methods and Tools, pp. 126-132, Milan, Italy, September, 1999.

Jones S., Nunez J.L., **Feregrino C.** and Mahapatra S., *Gbits/s Lossless Data Compression Systems*, IEE Colloquium Data Compression: Methods and Implementations, IEE Savoy Place, London, U.K., November, 1999.

Mahapatra S., Nunez J.L., **Feregrino C.** and Jones S., *Parallel Implementation of a Multialphabet Arithmetic Coding Algorithm*, IEE Colloquium on Data Compression: Methods and Implementations, IEE Savoy Place, London, U.K., November, 1999.

Nunez J.L., **Feregrino C.**, Jones S. and Bateman S., *X-MatchPRO: A ProASIC-Based 200 Mbytes/s Full-Duplex Lossless Data Compressor*, Proceedings of the 11th International Conference FPL 2001, Lecture Notes in Computer Science series, Springer-Verlag, pp. 613-617, Belfast, North Ireland, August, 2001.

Stefo R., Nunez J.L., **Feregrino C.**, Mahapatra S. and Jones S., *FPGA-Based Modelling Unit for High Speed Lossless Arithmetic Coding*, Proceedings of the 11th International Conference FPL 2001, Lecture Notes in Computer Science series, Springer-Verlag, pp. 634-647, Belfast, North Ireland, August, 2001.

Feregrino C. and Jones S., *Optimisation of PPMC Model for Hardware Implementation*, Proceedings of the 2001 Euromicro Symposium on Digital Systems Design (DSD'01), IEEE Computer Society Press, pp. 120 – 126, Warsaw, Poland, September, 2001.

