TÉCNICO LISBOA

Data Compression Algorithms in FPGAs

Gonçalo César Mendes Ribeiro

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor: Prof. Horácio Cláudio de Campos Neto

Examination Committee

Chairperson: Prof. António Manuel Raminhos Cordeiro Grilo
Supervisor: Prof. Horácio Cláudio de Campos Neto
Members of the Committe: Prof. Pedro Filipe Zeferino Tomás

May 2017

Acknowledgements

I would like to thank Professor Horácio Neto, who accepted to be a part of this journey as my supervisor and always helped me when I needed. I also thank the Electronic System Design and Automation (ESDA) group of INESC-ID — which I integrated during the course of this work — for the resources they provided me.

A thank you to Charles Bloom and Mark Adler, who did not leave me without a reply when I contacted them with some questions.

Also, I would like to thank the friends who accompanied me during my time at IST. Alongside some of them I helped build HackerSchool, which helped me grow and made these years more enjoyable.

I would also like to express my gratitude to my parents and family without whom I could not have started pursuing this course, much less finish it with this work. And finally, to Rita, who is always supportive, a great company and makes me smile.

Abstract

Data compression is increasingly important as more and more data is produced, transferred and stored on a daily basis. It allows to reduce storage costs, speed up data transfers over limited bandwidth, reduce network congestion and even improve energy efficiency in wireless devices.

In this work we propose a stream-based LZ77 hardware accelerator that implements the most computationally intensive part of many compression algorithms. The stream-based interface of this IP makes it easy to integrate with other hardware or software components to implement different compression applications.

This IP was integrated into a hardware/software architecture that implements the widely used Deflate lossless compression algorithm. This architecture targets embedded computing systems using software and hardware components. The hardware component accelerates the LZ77 algorithm, while the software selects the best matches and codes them with static Huffman. Executing the software part on a generic processor allows for greater flexibility of the encoding part.

A prototype implementation targeting a Zynq device demonstrated that the hardware accelerator can process 123 MiB/s and can easily be scaled in order to enhance the compression ratio. Compared with a software-only implementation running on the Zynq ARM processor a speedup of 1.5 is achieved.

The proposed IP can also be used to implement a system that surpasses the throughput of Gzip running on a Intel Core i-5 computer by a factor greater than 2, for typical files.

Keywords

Data Compression, FPGA, LZ77, Embedded Systems, Zynq

Resumo

A compressão de dados é cada vez mais importante à medida que mais dados são produzidos, transferidos e armazenados diariamente. Comprimir permite reduzir custos com armazenamento, aumentar a velocidade de transferências em ligações com largura de banda limitada, reduzir o congestionamento em redes e melhorar a eficiência energética de dispositivos que utilizam comunicações sem fios.

Neste trabalho propomos um acelerador de hardware do algoritmo LZ77, que implementa a parte computacionalmente mais intensiva de muitos algoritmos de compressão. A interface deste acelerador permite processar fluxos de dados e torna simples a integração do acelerador com outro hardware ou software com vista a implementar diferentes aplicações de compressão.

O acelerador foi integrado numa arquitectura hardware/software que implementa o algoritmo Deflate, que é amplamente usado para compressão sem perdas. Esta arquitectura visa ser implementada em sistemas embebidos com componentes de software e hardware. A componente de hardware acelera o algoritmo LZ77, enquanto o software selecciona as melhores *matches* e codifica-as com Huffman estático. A parte executada em software confere uma maior flexibilidade à codificação.

Um protótipo foi implementado recorrendo a uma Zynq e demonstrou que o acelerador de hardware consegue processar até 123 MiB/s e é facilmente escalável de forma a aumentar a razão de compressão. Comparado com uma implementação em software a correr no ARM da Zynq o protótipo é 1.5 vezes mais rápido.

O acelerador proposto pode também ser usado para implementar um sistema de compressão que ultrapassa a velocidade do Gzip a correr num processador Intel Core i5 em mais do dobro, para ficheiros comuns.

Palavras-chave

Compressão de Dados, FPGA, LZ77, Sistemas Embebidos, Zynq

Contents

List of Figures

List of Tables

List of Acronyms

ALM Adaptive Logic Module

ARM Advanced RISC Machine

ASCII American Standard Code for Information Interchange

AXI Advanced eXtensible Interface

BRAM Block RAM

BWT Burrows–Wheeler Transform

CPU Central Processing Unit

CRC Cyclic Redundancy Check

DDR Double Data Rate

DMA Direct Memory Access

FIFO First In, First Out

FPGA Field-Programmable Gate Array

GIF Graphics Interchange Format

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

HTTPS HTTP Secure

IANA Internet Assigned Numbers Authority

IP Intellectual Property

LE Logic Element

LSB Least Significant Bit

LUT Look-Up Table

LZMA Lempel–Ziv–Markov chain algorithm

LZW Lempel–Ziv–Welch

MM2S Memory Mapped to Stream

MSB Most Significant Bit

PCI Peripheral Component Interconnect

PNG Portable Network Graphics

RAM Random-Access Memory

RFC Request for Comments

RISC Reduced Instruction Set Computer

ROM Read-Only Memory

S2MM Stream to Memory Mapped

USB Universal Serial Bus

WOFF Web Open Font Format

1

Introduction

Contents

1.1 Motivation

In the last decades data compression has received increasing attention as the rate at which data is produced continues to increase. In 2013 about 2 million terabytes were generated per day and it is estimated that by 2018 the global Internet traffic will be 50 TB/s.

Data compression can help transmitting and storing these large amounts of data. The most obvious application is reducing the size of stored data, which can benefit both big data centres and small devices such as mobile phones. Compressing the data allows for faster transmissions as it virtually increases the bandwidth of links, which can also reduce congestion. Wireless devices can also benefit in terms of battery life, as transmitting a single bit can use as much energy as one thousand cycles of an embedded processor [1]. Finally, compression allows to save and retrieve data faster from slow storage media, which allows for example an operating system kernel to load faster, resulting in improved boot times.

Unfortunately, compressing data is a computationally intensive task. Its CPU-bounded nature means that a system can spend most of its processor time with a single compression task, seriously degrading performance for other tasks. It also means that, on a certain system, compression is only as fast as the available CPU. For these reasons, using compression coprocessors is beneficial for systems where compression is a common task, and many works in the literature have proposed such coprocessors.

Motivated by the merits of data compression and aware of its difficulties, the author proposed to study the field and make a contribution to it, mainly in the form of an architecture to speed up some compression algorithm. After a broad consideration of the data compression literature the work focused on lossless compression, of which various methods were studied before deciding to develop the remainder of the work around the ubiquitous Deflate algorithm, found in formats such as Zip, Gzip, Zlib and PNG.

1.2 Basic Concepts

This section describes some fundamental concepts of data compression necessary to understand the remainder of this work.

1.2.1 Lossy vs. Lossless Data Compression

In certain contexts compression might be lossy, i.e. degrade the original data in an irreversible way but such that the data that remains can still convey the original information to a certain degree. *Lossy compression* is widely used for image, audio and video, since removing details from such data usually still lets humans correctly perceive the content. Lossy compressors usually provide better compression ratios that lossless compressors. Also, the compression ratio can be controlled by trading quality for the amount of compression.

Unlike lossy compression, *lossless compression* is reversible: the decompressed data exactly matches the original data. This is desirable for applications in which it is unacceptable to degrade the

original data. There are many such applications some of which are text compression, computer program compression and genome compression. This work focuses on lossless compression algorithms.

1.2.2 Performance Metrics

The performance of a compression algorithm can be evaluated according to several metrics. Some commonly considered metrics are: how well the algorithm compresses (compression ratio); how fast one implementation compresses or decompresses on a certain machine (*throughput*); and how much memory is used. The combination of these parameters for a certain algorithm may make it better suited for some situation when compared to others. There is no "best compression algorithm." Each algorithm has some merits and demerits for a specific application.

Multiple definitions are commonly found for the compression ratio. The definition used throughout this work is that the *compression ratio* is the ratio of the number of bytes needed to store the original uncompressed data to the number of bytes needed to store the compressed data

$$\text{compression ratio} = \frac{\text{uncompressed size}}{\text{compressed size}} \,. \tag{1.1}$$

For example, a ratio of 2 would mean the uncompressed data is twice as large as its compressed version. With this definition a higher ratio means the data was compressed more efficiently.

1.3 Objectives

This work addresses the following objectives:

- Improving the throughput of a lossless data compression algorithm by using dedicated hardware;
- Offloading the most computationally intensive part of the compression algorithm from a general purpose processor to the specialised coprocessor;
- Demonstrate the functionality of the proposed architecture with a prototype;
- Analyse and evaluate the hardware/software implementation of the compression algorithm.

1.4 Contributions

The main contributions of this work are:

- A hardware architecture for a fast stream-based IP implementation of the LZ77 algorithm;
- A hardware/software prototype implementation of the Gzip compression software;
- A simulator of the compression ratio depending on implementation parameters;
- A study of the effects those implementation parameters have on the compression ratio;

- The conclusion that for this type of application a general purpose processor constitutes the bottleneck for a hardware/software implementation.

1.5 Outline

The remainder of this work is organised as follows: Chapter 2 starts by reviewing some of the most important lossless data compression algorithms, and then it presents some of the most recent works concerning hardware implementation of the LZ77 and Deflate algorithms in reconfigurable devices; Chapter 3 explains the Deflate algorithm and aspects of its implementation in the Gzip software; Chapter 4 studies the effects of several implementation parameters on the compression ratio of static-Huffman Deflate, using a compression ratio simulator designed to this effect; in Chapter 5 a hardware architecture for a LZ77 accelerator is presented, followed by its integration into a hardware/software system implementing Gzip; Chapter 6 shows performance results for the hardware accelerator and for the hardware/software system, and discusses how the performance of the LZ77 accelerator can be maximised; and finally, Chapter 7 concludes this work and offers insights on how it could be further developed in the future.

2

State of the Art in Lossless Data Compression

Contents

In this chapter a review on some of the most important lossless data compression algorithms is presented. A brief description of each algorithm is provided along with references for further reading. Entropy coding methods are presented in Section 2.1, followed by dictionary methods in Section 2.2. In Section 2.3 other lesser-known methods are introduced, some of which provide the better compression ratios to date. Based on this review the Deflate algorithm is chosen as a case study. Thus, a review on hardware implementations of the Deflate algorithm from the last ten years is presented in Section 2.5.

2.1 Entropy Coding Methods

In 1948 Claude Shannon laid the foundations for information theory. In [2], Shannon defines the entropy of a set of probabilities and the entropy of an information source. He shows that the entropy can be seen as the average number of bits needed to code the output of a source. He also proves that a lossless compressor can do no better than encode the output of a source with an average number of bits equal to the source's entropy.

Robert Fano independently developed similar work, which was published in [3]. Both works present a similar method to encode symbols in a way that reduces the average number of bits needed per symbol, known as *Shannon–Fano coding*. This algorithm takes into account the probabilities for each symbol and finds corresponding codes such that the codes for the most frequent symbols have a lower number of bits and vice-versa. Furthermore all the resulting codes are different and constitute a *prefix code*, i.e. a code in which no codeword is a prefix for any other codeword. This makes it possible for the receiver to decode the message unambiguously. Nonetheless, Shannon–Fano coding is suboptimal since the algorithm cannot always find the lowest length codeword for a symbol.

In [4], David Huffman presents a coding method which improves upon Shannon–Fano coding. Huffman proves that his algorithm finds an optimal prefix code given a set of symbols and their probabilities. Therefore *Huffman coding* is an optimal way to encode symbols individually. But this does not mean no better compression methods exist: when using Huffman coding, choosing which "symbols" to use plays an important role in attaining good compression ratios. Some compression algorithms use Huffman coding as their last step after carefully choosing symbols by transforming the input in some way.

For Shannon–Fano or Huffman coding to be used, the probability distribution of the symbols must be known. The most general way to know the distribution is to scan the input and count their occurrences. This is general in the sense that no assumptions are made about the input. But in some cases the input is known, expected or transformed to follow a particular distribution. For example, *run-length encoding* (RLE) of 0s and 1s outputs a count of successive 0s followed by a count of successive 1s. If the probability of occurrence of a 0 is p, 1s occur with probability $1 - p$. A sequence of n 0s followed by a 1 has probability of $p^n(1 - p)$. Therefore, probabilities of run lengths follow a geometric distribution — smaller run lengths occur with higher probability. In [5], Solomon Golomb presents a method to encode non-negative integers that follow a geometric distribution. He shows how $\mathbb{N}_0$

can be partitioned into groups of $m = -1/\log_2 p$ elements and a variable-length code generated such that groups with smaller integers have smaller codes: the number of the group is coded using unary coding (variable length) and the position inside that group is coded with the ordinary binary representation (fixed length). *Unary coding* simply means that a positive integer n is represented as n 1s followed by a 0. *Golomb coding* involves calculating a quotient $q = \lfloor n/m \rfloor$ and a remainder $r = n - qm$, which can be considered computationally expensive. By limiting the values of m to powers of 2 the quotient and remainder can be efficiently computed by shifting. This particular case of Golomb coding is called *Golomb–Rice coding* or *Golomb-power-of-2* (GPO2). Compared with Huffman coding, Golomb coding might be more practical if the alphabet of run lengths can be very large. Another example of data for which the probability distribution is approximately known a priori is English text. Some studies on the entropy of English are in [6–9].

For the coded message to be decoded the decoder must know which probabilities were used by the coder. If no model for the information source is known a priori the coder must transmit to the decoder the perceived probability distribution. This transmission can explicit or implicit. For example, when using Huffman coding and if the input can be fully scanned before encoding, a Huffman code can be constructed and explicitly transmitted to the decoder as a header prior to sending the coded data itself. An algorithm to create such an header can be seen in [10]. On the other hand, if no model is known and the input cannot be scanned before coding (to determine a model) an adaptive coding method can be used. In *adaptive coding* methods both the coder and decoder start with a predefined model which they adapt according to the transmitted symbols so that both have the same model. Adaptive Huffman coding was independently developed by Newton Faller [11] and Robert Gallager [12] and improved by Donald Knuth [13] (resulting in the FGK algorithm) and Jeffrey Vitter [14] (Vitter algorithm). For this coding method an updated source model is transmitted implicitly every time a symbol is coded. Each time a symbol is coded the code might need to be updated in order to respect the properties of Huffman coding. Another adaptive method is *Rice coding*, introduced by Robert Rice in [15]. The coder codes sequences of J symbols using four different "operators" and checks which operator resulted in the smallest coded sequence. This sequence is transmitted prefixed by a header of two bits identifying which operator was used. Rice points out that this might be considered brute force, but that the operators are so simple that it is feasible to compute them all. He also shows that his coding method is efficient for sources with average entropy ranging from 0.7 to 4 bits/symbol.

Even though Huffman coding is optimal when coding messages symbol by symbol, the expected length of the codewords can be up to $p_{max} + 0.086$ bits greater than the entropy, where p_{max} is the probability of the most frequent symbol [12]. The redundancy (expected length minus entropy) can be reduced by concatenating symbols to form a new alphabet and using the new symbols. While this can reduce the redundancy because p_{max} is reduced, the number of symbols in the new alphabet grows exponentially and so does the number of codewords. Beyond a certain number of concatenations using Huffman coding in this way becomes unfeasible. In [16], Jorma Rissanen presents the base for *arithmetic coding*, which allows to code a particular sequence of symbols without the need to produce codes for all the sequences of that length. A message is represented by an interval of real numbers

in the range $[0, 1[$. As more symbols are encoded the interval size must be reduced and more bits are needed to represent it; the most probable symbols reduce the range less than the least probable ones. The algorithm was generalised in [17] and [18]. It only became popular almost one decade later due to an effort of Ian Witten et al. [19] in pointing to the community that Huffman coding had been surpassed. Practical implementations are discussed in [20] and [21]. Adaptive models can be used with arithmetic coding as discussed in [19, 21].

Entropy coding methods are still in use in several applications. Huffman coding and arithmetic coding are the most popular methods and many times they are combined with other methods, such as dictionary methods.

2.2 Dictionary Methods

In 1977 Jacob Ziv and Abraham Lempel started a new branch in compression history with [22]. In this paper they describe an algorithm which uses a *dictionary* — a list of patterns occurring frequently in the input. The algorithm in that work became known as the *LZ77 algorithm* and it uses a sliding window as the dictionary, which slides over the input as it is encoded. This window stores the last n symbols emitted by the source. The encoding process consists in finding phrases in the input that occur in the sliding window. When a match (or the longest match) is found in the dictionary, the LZ77 coder emits a triple consisting of the offset in the window where the match was found; the length of the match; and the codeword for the next symbol of the input after the current phrase. Whenever a match is not found a triple $(0, 0, C(s))$ is produced, where $C(s)$ is the codeword for symbol s. Since the window has a finite length, repetitions in the input with period longer than n cannot be detected and compressed by LZ77. LZ77's dictionary contains all single symbols and all the sub-strings of the string that is in the window.

In the following year the same authors published [23] which originated the *LZ78 algorithm*. LZ78 uses an explicit dictionary instead of a sliding window. This dictionary stores phrases previously found in the input. The input is parsed and when a match is found the LZ78 coder outputs a double $(i, C(s))$, where i is the index for an entry already in the dictionary that has the longest match; and $C(s)$ is the codeword for the input symbol after the matched phrase. Then a new entry is added to the dictionary: its index is the next free index and its content is the concatenation of the content at index i with s. When no match is found the special index 0 is used (consequently the dictionary starts at index 1). Because of the way the entries are constructed, the new entries become longer and longer, allowing for lengthier matches. Unlike LZ77, once a phrase enters the LZ78 dictionary it will always be detected and compressed. In practice the growth of the dictionary must be limited in some way, which may mean matches will not grow beyond a certain length.

The Lempel–Ziv algorithms became popular and are the base for many other algorithms (see Figure 2.1). It has been proven in [24] and [25] that both achieve asymptotically optimal compression ratios for ergodic sources (i.e. sources for which every sequence is the same in statistical properties). Many LZ-class algorithms derive from LZ77. The adoption of LZ77 was greater than that of LZ78

because of patenting issues concerning the LZW variant of the latter (those patents expired in 2004) Many of the LZ descendents never became popular and are little known. In the following paragraphs some of the most relevant or recent ones are outlined.

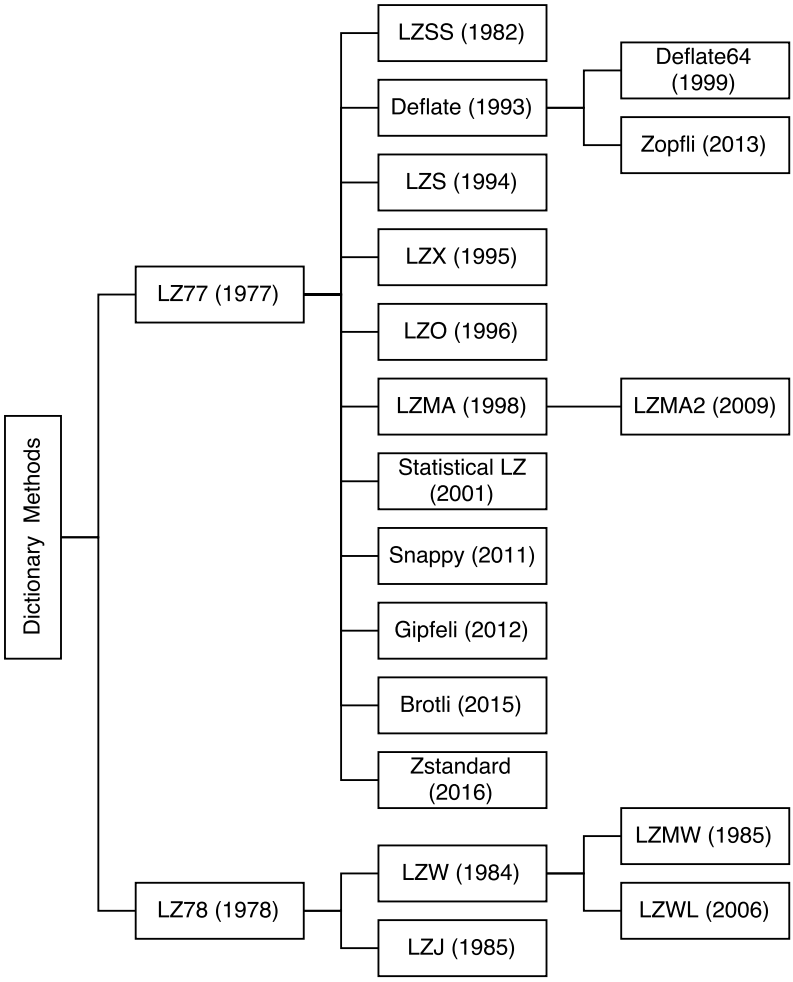


Figure 2.1: LZ class of algorithms

The Lempel–Ziv–Welch (LZW) algorithm is the most know variation of LZ78. It was proposed by Terry Welch and described in [26]. With LZW, Welch aimed to popularise the use of compression in commercial computer systems by presenting a simple, adaptive and fast algorithm with good compression ratio. The LZW dictionary starts with entries for all the symbols in the alphabet. The input is stored as a string ω until a character K is read that results in a pattern ωK that is not in the dictionary. At this point, the index for the entry with ω is emitted and a new entry with content ωK is added to the dictionary. While LZ78 emits a double of the form $(i, C(s))$, LZW emits a single: (i). LZW is used in UNIX's `compress` utility. When Welch published [26] he had a patent pending for it. Spencer Thomas did not know this and used LZW in `compress`. This meant the users of the program had to pay royalties to use it, and hence its popularity fell. The GIF image format also uses LZW and its developers were unaware that it was patented. The format became widespread before the patent was enforced. Nonetheless, the controversy surrounding GIF resulted in the development of the PNG format, which uses the Deflate algorithm (based on LZ77). The patent problems with LZW reduced

its popularity and consequently the popularity of LZ78. The LZW patent expired in 2004.

The Deflate algorithm and data format was developed by Phil Katz as a part of his PKZIP archiving program. The algorithm results from a combination of LZ77 followed by Huffman coding. The format is specified in RFC 1951 [27]. As Deflate uses LZ77 it needs to encode references to strings in the window as well as literal symbols (which in Deflate are bytes). The sliding window used in Deflate has a maximum size of 32 kB and consequently offsets are limited to 32 kB. Match lengths are specified to be in the range [3, 258]. The triples produced by LZ77 are encoded using two Huffman codes. One of the codes is for offsets: 30 ranges are specified, which combined with a variable length of bits fully specify an offset. The other Huffman code combines the alphabets for literals and for match lengths into a new alphabet. This alphabet has 286 elements: 256 literals; one end-of-block symbol; and 29 symbols to represent ranges for match lengths. Deflate is one of the most widespread compression algorithms. It is used in the popular Zip and Gzip [28] formats, Transport Layer Security (TLS) protocol version 1.0 to 1.2 [29–31], HTTP/1.x [32, 33] [1] , Secure Shell (SSH) [36], Zlib [37], PNG [38], etc. Katz filed a patent for his original implementation of Deflate [39]. Nonetheless, alternative implementations can be used — such as the one in [27] — that are not covered by patents and generate data in the Deflate format. Implementations of the algorithm exist both in software and hardware. A review of Deflate hardware implementations is presented in Section 2.5. Further details on Deflate and its implementation in Gzip are in Chapter 3.

Lempel–Ziv–Markov chain algorithm (LZMA) was apparently invented by Igor Pavlov for his file archiver 7-Zip, first released in 1999. No scientific publications explaining the algorithm exist, nor is it well documented. Its implementation is available in the LZMA SDK [40], which is in the public domain since 2008, so the algorithm can be studied and modified from its implementation. LZMA is a combination of LZ77 and arithmetic coding[2]. The original LZ77 is a greedy algorithm: it parses the input and uses the first match it can. By looking ahead some symbols of the input, a non-greedy algorithm can make better choices on which phrases to parse and hence achieve better compression ratio [41]. The implementation in LZMA uses non-greedy parsing. Another aspect of LZMA is that it uses a special “repeat match” (short) code to encode the three most recent matches’ offsets (as in the LZX algorithm [42]). Literals, matches and repeated matches are all coded using a binary (i.e. the used symbols are bits) arithmetic coder with order-n context. The context varies according to what is being coded. Literals, for example, start with the previous byte and lower bits from the current position as context (the bits from the position are an attempt to capture structure that repeats at powers of 2) and as the encoding of the literal advances bits from the literal are also used in the context. More details on LZMA can be gathered from [42–44]. LZMA can achieve compression ratios significantly higher than Deflate at the expense of an increase in computing time and memory. While Deflate is limited to a dictionary of 32 kB, LZMA can use up to 4 GB. The high compression ratio allied to increasingly powerful computers is making this algorithm more and more used. The most known implementations are in 7-Zip and in `xz`.

[1] Compression is seemingly going to be dropped in TLS v1.3 because of the CRIME and BREACH exploits. For HTTP/2 [34] a new header compression method was developed [35] to prevent these exploits.

[2] More precisely *range coding*, an integer based version of arithmetic coding.

In the last decade the engineers at Google have developed and open-sourced at least four compression algorithms, all based on LZ77: Snappy, Gipfeli, Zopli and Brotli. Snappy [45] aims to be very fast, both at compression and decompression. Its results show that it can compress at 250 MB/s and decompress at 500 MB/s, for the test files for which it was slowest (for comparison Zlib (Deflate) compresses at 74 MB/s in its fastest setting and 24 MB/s with default settings). This increased speed is attained at the expense of compression ratio: Snappy's compression ratio is 20 to 100 % lower than Zlib's. Version 1.1 uses a sliding window with 64 kB, but the format allows a dictionary up to 4 GB (larger windows can be slower to decode [46]). The encoding is byte-oriented and no entropy coding method is used. Because "incompressible" data can slow down LZ algorithms, Snappy uses an optimisation so that when no matches are found for 32 consecutive searches the next searches are performed only for alternate bytes. If after 32 alternate-byte searches no match is found, searches are performed only every third byte; and so on. The algorithm restarts "normal" matching when a match is found. Snappy has been used internally by Google in their MapReduce [47] and Bigtable [48] projects as well as in Google's remote procedure call (RPC) systems.

Fast compression algorithms like Snappy are so fast that I/O operations might be the bottleneck of the algorithm. Gipfeli [49] is another fast algorithm, about 30 % slower than Snappy but achieving 30 % more compression ratio. It is three times faster than Zlib (with fastest settings) and achieves similar (yet lower) compression ratio. A dictionary with a maximum of 64 kB is used. Unlike Snappy, Gipfeli entropy-codes literals and match information: for matches a static entropy code is used, which was built based on statistics for text and HTML files; literals use an ad-hoc entropy code based on taking samples from the input. Huffman or arithmetic coding were not used "because of their slow performance." For the algorithm not to slow down with data that is hard to compress the same approach was used as in Snappy. The researchers replaced Snappy for Gipfeli in MapReduce and obtained up to 10 % speed improvements in computations.

Zopfli [50] aims to compress data into the Deflate format with higher compression ratios than those achieved by implementations as `gzip` and `zlib`. Zopfli produces compressed data 3.7 to 8.3 % more compact than `gzip --best`. However, the execution time is around two orders of magnitude higher than that of `gzip`. Decompressors take an identical time to extract data compressed with either algorithm. As seen before, Deflate is widely deployed. Because of this, data compressed with Zopfli can be readily used in many applications. One use case is in the web: static pages can be better compressed with Zopfli and browsers can decompress the HTTP [33] content without the need for any update. The browser user might see relatively small improvements in bandwidth or page load times when compared to other Deflate data. Nonetheless, for mobile users the small extra compression ratio might mean less energy spent in wireless communications and therefore increased battery duration. PNG [38] images also use Deflate and can benefit from Zopfli's better compression, resulting in further gains since they are popular on the web. For content delivery networks (CDNs) and websites with high traffic, Zopfli-coded content can result in significant improvements in bandwidth, total data transferred and needed storage space. Unfortunately Zopfli's throughput might not be suitable to compress dynamic content. A good reference on implementation details seems not to

exist. A very brief remark can be found in [51] stating that Zopfli searches a graph of possible Deflate representations.

Brotli [52] is currently Google's newest open-source compression algorithm. It was first released in 2015. Unlike Zopfli, Brotli is not intended to be Deflate compatible. In fact Brotli is intended be a modern replacement for Deflate. Since Deflate is popular for its fast compression and decompression speeds along a relatively good compression ratio, for Brotli to be a suitable replacement it should be at least as fast and provide somewhat better compression. In [46], Google researchers compare Brotli with other algorithms, including Deflate, Zopfli and LZMA. Results show that for a target compression ratio Brotli generally provides faster compression than Deflate and both are on par when decompressing. Furthermore, Brotli can achieve significantly higher compression ratios when using its slowest settings: it can compress 20 to 26 % more than Zopfli — which in turn (as seen before) compresses more than Deflate — while being more than twice faster. For some files LZMA can attain around 2.5 % better compression ratio considerably faster than Brotli. But LZMA's decompression time is four to five times higher, making it unsuitable to replace Deflate. Brotli uses a sliding window up to 16 MB in size. Alongside this dynamic dictionary a static one is used. It contains syllables and words from several human languages and phrases used in computer languages as HTML and JavaScript. The static dictionary is 120 kB in size and transforms can be applied to the its entries such that a total number of about 1.6 million sequences can be represented. This dictionary is particularly useful when compressing small files. Data is encoded as a series of "commands." Each command is composed of three parts 1) the number of literals, n, encoded in this command and the length of the match, 2) a sequence of n literals and 3) the offset of the match. Each of these parts is Huffman-coded using an alphabet specific to that part. In fact more than one code can be used for each part, depending on the context the symbols appear in. For literals the context is the previous two uncompressed bytes; and for offsets the length of the match is used as context. Apart from these, additional contexts can be specified for each part of the commands. For further details please refer to [52]. As new as Brotli is, it is already in use on the Internet: the WOFF 2.0 font file format [53] uses Brotli for compression; and Brotli has been accepted into the HTTP Content Coding Registry [54] by the IANA. Chrome, Firefox and Opera web browsers currently support Brotli when compressing HTTPS.

Dictionary-based compression algorithms are undeniably the most commonly used algorithms in lossless data compression. Among them, LZ77 is one of the most known and constitutes the base of many others.

2.3 Other Lossless Methods

Entropy coding and dictionary methods are undoubtedly the most widely known categories of compression algorithms. Nonetheless other less known methods exist. In this section Burrows–Wheeler transform, context modelling and context mixing algorithms are presented. Many of today's programs with best compression ratios stem from one of these algorithms.

The Burrows–Wheeler transform (BWT) was invented by David Wheeler in 1983 and first published by Wheeler and Michael Burrows in 1994 in [55]. It is not a compression algorithm per se, but rather, it sorts blocks of the input in such a way that makes them more compressible by other lossless compression algorithms. It works by performing all the possible rotations of the input block. Then, these rotations are sorted lexicographically, forming a list where the original block can be found at index i. The string formed by the last character of each of these sorted rotations, along with index i, constitute the BWT of the original block. From this index and the new BWT-sorted block the original block can be recovered. One property of the BWT block is that it contains clusters of equal symbols, i.e. if a symbol s appears in a certain position in the block, there is a high probability that contiguous positions also contain s. This property can be exploited by the move-to-front (MTF) coding method, which is suited to take advantage of locality of reference [56, 57]. The most known implementation of BWT followed by MTF is found in the `bzip2` program.

Context modelling compression algorithms build a model that allows them to predict which symbol comes next in the input. These predictions are based on what is called a order-n context. Consider the word $symbol$ as an example. For the letter l the order-5 context would be $symbo$; the order-4 context is $ymbo$; and so on, until order-0 context, which is l itself. Using order-0 means when encoding a symbol we don't take past symbols into account. This means that the symbol's probability is simply the number of times it occurs divided by the total number symbols that occurred. When using a order-n context with $n > 0$ the probability of a symbol is a conditional probability given that the previous n symbols occurred. Generally, a higher order context gives a better chance of correctly predicting the next symbol, because that symbol probably has a high probability in the current context. A high probability means the symbol will take less bits to encode using for example arithmetic coding. Nonetheless, it might be unfeasible to gather probabilities for high order contexts, because the number of different contexts grows exponentially with the length of the context. Furthermore the majority of those contexts will never occur: for example the string $aaaa$ almost never occurs in a normal English text. Therefore it is wasteful to calculate all probabilities for all contexts a priori.

The most known context modeling algorithm, called Prediction by Partial Matching (PPM) [58], solves this problem by calculating the probabilities as needed. The maximum length of the context is decided beforehand. When encoding a symbol, the longest context is checked: if this symbol previously appeared in that context we know its probability and encode it accordingly; otherwise the size of the context is reduced and an *escape symbol* is encoded, signalling to the decoder that this reduction happened. This process is repeated until the symbol is found in some context. If this is the first time the symbol appears in the input, not even order-0 context will have information about it. Therefore a fallback fixed probability exists for each symbol. Before proceeding to the next symbol, the counts of the times this symbol appeared with these contexts is updated. The tricky part about PPM is which probability to assign to the virtual escape symbol. This is called the *zero-frequency problem*, which is described in [59]. Many variants of PPM appear in the literature, proposing different solutions for this problem. The most popular variation is PPMd, based on [60] (not to be confused with another version called PPMD [61]).

The algorithms mentioned is this section are little-known compared to the ones in the previous sections. They provide the best compression ratios to date, but are mostly used only for investigation.

2.4 Compression Benchmarks

The throughput and compression ratio a particular compression algorithm can achieve depends on the data being compressed. Therefore, comparing algorithms or implementations is only possible by testing them with the same data sets. For this reason, a few standard data sets — referred to as *corpora* — are commonly found in works about compression. The ones most frequently found are the Calgary, Canterbury, Enwik and Silesia corpora. Table 2.1 shows the sizes of these Corpora.

The Calgary and Canterbury corpora are relatively small, with around 3 MB. On modern computer systems these small file sizes can be compressed in less than a second by the fastest algorithms. Therefore, in order to obtain more reliable results the Enwik and Silesia corpora can be used, whose sizes are in the order of hundreds of megabytes. The Enwik corpus has several variants called Enwik9, Enwik8, etc. The largest is Enwik9 and the others are obtained by truncating the file to 10^n bytes, where n is the number in the name of the variant.

The contents of each corpora consist of a combination of text files such as books or HTML and binary files such as program's binaries or raw data, except for the Enwik corpora, which is a text excerpt from the Wikipedia. In this work, the corpora with multiple files are compressed as a whole, by first archiving the files into a `tar` file.

Table 2.1: Commonly used corpora and their sizes

Corpus	Raw size (B)
`canterbury.tar`	2821120
`calgary.tar`	3152896
`enwik8`	100000000
`silesia.tar`	211957760
`enwik9`	1000000000

2.5 Hardware Architectures for Deflate

From the previously mentioned algorithms, the one whose usage is most widespread is undoubtedly the Deflate algorithm. It is used in the widely available Zip and Gzip formats, and is also prevalent on the web, where it is used by 71 % of the top ten million websites [62]. For this reason, and because the algorithm is relatively simple and well-documented and there are numerous scientific literature works on it, Deflate will be used as the case study throughout this work.

This section presents several hardware architectures related to the implementation of Deflate, focusing on LZ77, which is the slowest part of the algorithm. Only works published in the last ten years are considered in this section, as they tend to best represent the current technologies and have best performance, while they refer older works when appropriate. We group the architectures into two main categories: architectures that use hashes to search for matches and architectures that do not.

These categories are then divided into architectures that were considered sufficiently dissimilar from the others.

2.5.1 Hashless Architectures

This type of architecture directly compares all the data in the window with the one in the lookahead, trying to find matches. It has no a priori knowledge of where in the window the best matches for the current lookahead might be. Therefore it needs to compare the lookahead with all the window's positions, making it impractical for "large" dictionaries. A general representation of this architecture can be seen in Figure 2.2. The window and the lookahead are fed into a comparison matrix, which compares all the sequences in the window with the sequence in the lookahead. The results of these comparisons enter a block which computes the best match (if any) and outputs the length of the match and its offset in the window buffer. If no match was found the literal will be encoded instead.

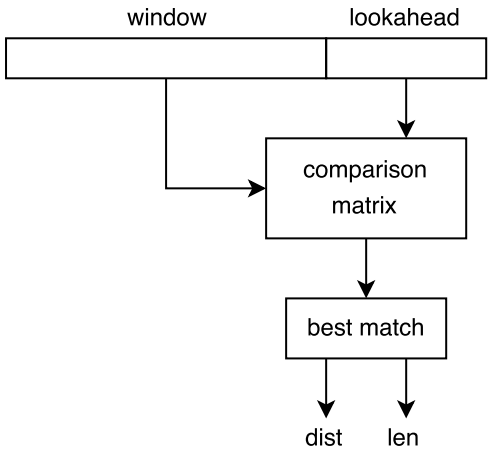


Figure 2.2: General hashless architecture

2.5.1.A Direct Implementation

The most direct implementation is an architecture that closely resembles the diagram in Figure 2.2. The main components are a comparison matrix and a best match calculator. Figure 2.3 depicts an example of all the comparisons that the comparison matrix must perform, for a dictionary of 8 bytes and a lookahead of 4 bytes. It can be seen that if the size of the window is W and L is the size of the lookahead then the comparison matrix must compare WL bytes. Figure 2.4 shows an architecture for the best match calculator block as proposed in [63]. The inputs of the block are of the form $w_i l_j$, which represents the result of the comparison of byte w_i of the window and l_j of the lookahead. The AND gates in each row of the architecture are chained in order to figure the match length of each particular combination of window and lookahead sequences. The ORs and multiplexers on the left side of the diagram select the longest length from all the rows. The rightmost part of the diagram features chains of multiplexers, which for each column forward the shortest distance for which a match exists, for the length corresponding to that column. The outputs of these chains of multiplexers are then selected by a multiplexer whose control signal is the best match length. In summary, this circuit finds the longest

match available at the shortest distance from start of the lookahead.

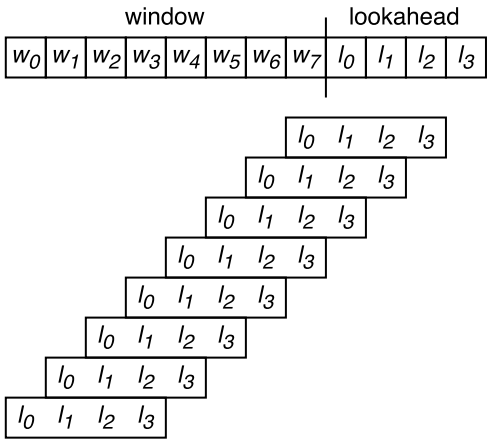


Figure 2.3: Representation of the comparisons that the comparison matrix must perform for $W = 8$ and $L = 4$

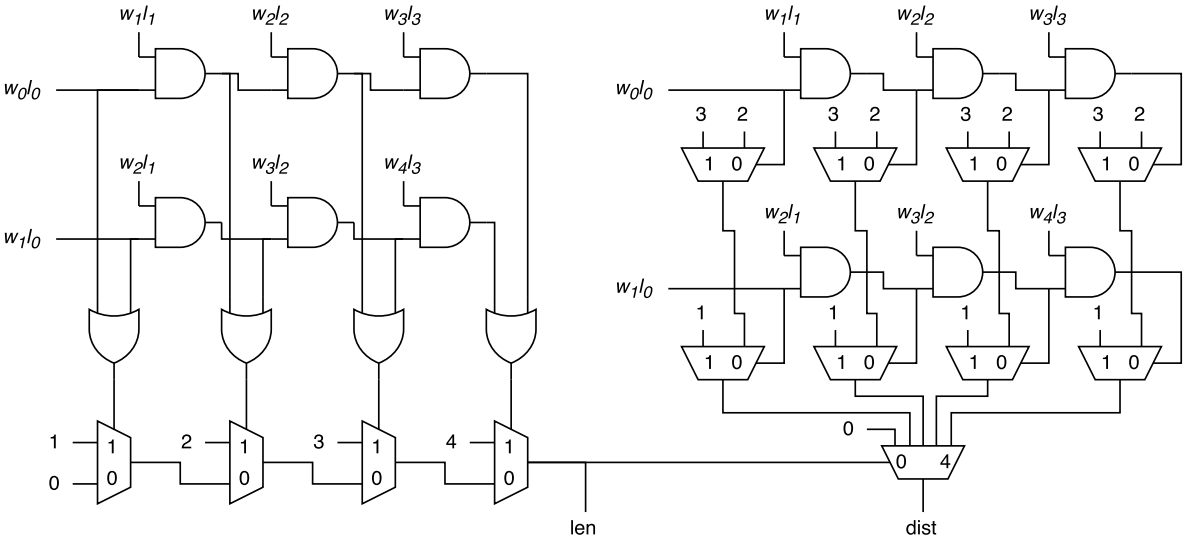


Figure 2.4: Architecture for a best match calculator with $W = 3$ and $L = 3$. The AND gates from the left and right halves of the diagram are the same.

In each clock cycle this architecture can shift the input in the window by a number of bytes equal to the length of the best match. This means a shift of a single byte in the worst case and of L in the best case. The worst case is when there are no matches or the best match is less than 3 bytes in length, because that is the minimum match length for Deflate. The throughput of this architecture will therefore depend not only on the clock frequency but also on the redundancy of the stream of data being processed.

In order to improve the throughput, it is possible to match future lookaheads in the present cycle. To do this more bytes of the input are appended to the lookahead. For each appended byte one additional future lookahead is evaluated in the current cycle. In terms of hardware each extra byte adds W byte-comparators to the comparison matrix and one extra best match calculator. Figure 2.5 shows that the results from the previously existing lookaheads can be used to find matches for the future lookaheads. Without reusing these results the increase in comparators would be WL instead.

However, extending the lookahead in this way does not increase the maximum match length. The maximum match length is still L, because that is the number of bytes each best match calculator can process. By extending the lookahead by L_e bytes, the input can be shifted by at least $L_e + 1$ bytes in each cycle and at most by $L + L_e$. However, additional circuitry is needed to decide which matches to commit, as the several discovered matches cannot overlap each other nor leave bytes of the input uncovered.

In [63], throughputs from 175 to 750 MB/s are reported, depending on the combinations of $W \in [16, 512]$ and $L \in [4, 32]$. The hardware usage ranges from 311 to 33k 4-input LUTs. In [64], a maximum throughput of 1.14 GB/s was reached, using 120k 6-input LUTs, but the sizes of the dictionary and lookahead are not specified. In both works the value of L_e is not clear, although in the latter it appears to be $L_e = L$.

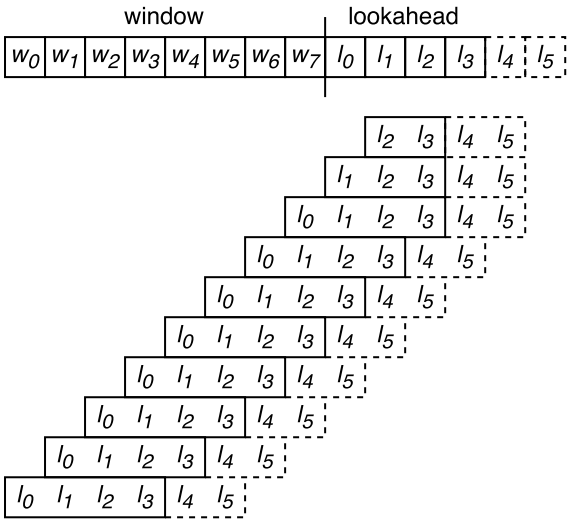


Figure 2.5: Unrolling of the comparison matrix to match $L_e = 2$ future lookaheads

2.5.1.B Systolic Arrays Architectures

A systolic array is a network consisting of a repeating processing element (*cell*). Each cell is connected to a small set of the cells close to itself. Cells execute some operations on the data and then pass it to its neighbours.

The comparison matrix in Figure 2.2 can be implemented using a systolic array, such that the cells gradually pass the window data to their neighbours and each cell compares it with bytes from the lookahead buffer. The results of the systolic array are then processed to find the longest match. Figure 2.6 shows the architecture of the simple systolic array described in [65]. The number of cells is equal to the length of the lookahead. Bytes from the lookahead enter the cells from the top, while the bytes from the window or lookahead enter the leftmost cell and are passed from one cell to the following. Each cell compares one byte from the window with one from the lookahead and outputs the result of that comparison. Each cell simply contains a byte-comparator and three registers (Figure 2.7).

The systolic array takes W clock cycles to compare all the sequences from the window with the

lookahead. The comparison results from each cycle, $m_{W-1}...m_0$ should be encoded into a binary integer to obtain the length for the candidate match. This can be accomplished for example with a chain of AND gates and an encoder of W to $\lfloor \log_2 W \rfloor$ bits. The best match calculator block must keep track of the best length found so far for the current lookahead as well as the position in the window for the corresponding match.

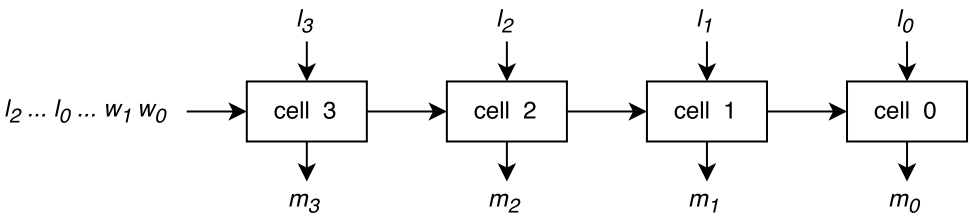


Figure 2.6: Systolic array architecture for $L = 4$

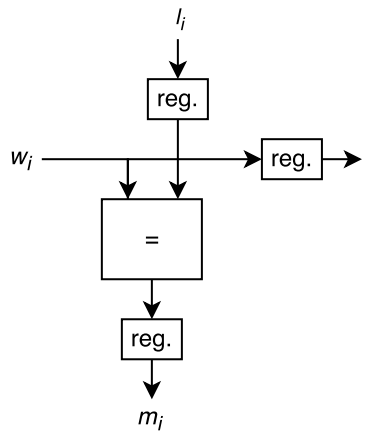


Figure 2.7: Architecture of a cell of the systolic array in Figure 2.6

A simple way to improve the throughput is to detect when a match with length equal to the maximum length — i.e. L — is found and immediately shift the window. In Deflate, shorter distances result in better compression, thus it would be better to feed the systolic array from the right to the left, starting with the lookahead and then the window. This would ensure that when a match with maximum length is found it is also the least distant match with that length. Another contribution to increase the throughput is to use more than one systolic array such that each one compares the lookahead to different segments of the window, as seen in Figure 2.8. Using n systolic arrays, the number of cycles to compare all window sequences is reduced from W to $\lceil W/n \rceil$. However, the number of length encoders increases to n and the best match calculator block must pick the best match from n different matches. Note that each time the lookahead advances, $L-1$ cycles are needed to propagate the window data so that all cells are ready to start matching the new lookahead in the next cycle. Therefore, the total number of cycles needed to match each lookahead is $\lceil W/n \rceil + L - 1$. In [65], a throughput of 1.6 MB/s is estimated for $n = 1$, $W = 1\ kB$ and $L = 16$. For the same parameters a total of 419 4-input LUTs are used.

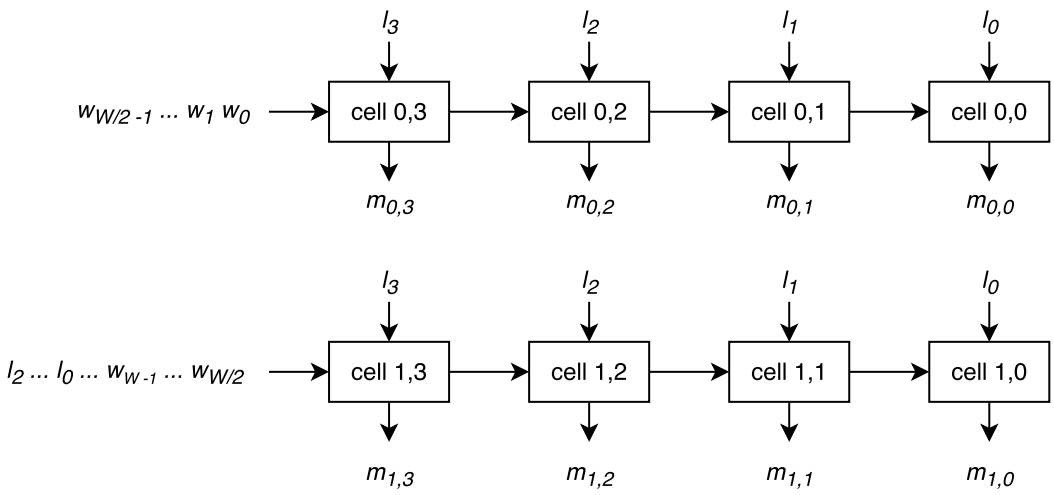


Figure 2.8: Using two systolic arrays to improve throughput, for $L = 4$. The upper array matches the first half of the window, while the lower matches the second half.

2.5.1.C Pipelined Dictionary

A hashless architecture fairly different from others is proposed in [66]. The architecture is divided into processing units each of which includes a part of the dictionary, comparison logic and match calculation logic. Each processing unit finds matches for a certain lookahead in a chunk of the dictionary memory. A diagram for the processing unit can be seen in Figure 2.9.

The lookahead is 16 bytes long and is compared to 16 dictionary sequences in each unit. Because each match can be at most 16 bytes long, the dictionary memory in each unit must contain 31 bytes. In order to reduce implementation area each unit features only four 16-byte comparators. Therefore, each comparator must perform four comparisons for each lookahead. This means the execution of each unit is divided into multiple cycles: one cycle is needed to access the dictionary memory and four more to perform the comparisons. While the memory is being accessed the four best matches found in the previous unit (PRENA) pass through the 2-1 multiplexers and the longest of those four matches is selected by the longest match circuit. This match can then be compared with LONMA — the longest match found so far for this lookahead. The LONMA output of the current unit is updated as needed. Then, during the four comparison cycles the 2-1 multiplexers pass the results of the lookahead-dictionary comparisons. In each cycle the best of those four matches is found and stored in one of the PRENA output registers. At this point each unit can start processing the next lookahead. Both PRENA and LONMA are two bytes long: the first 8 bits identify in which chunk of the dictionary the match was found; the next 4 bits are the offset in the chunk for the first byte of the match; and the last 4 bits are the length of the match.

The presented architecture, however, seems to be used with static dictionaries in [66], i.e. the dictionaries are fixed and implemented as ROMs. Hence, the purpose of the address ADDR on top of Figure 2.9 seems to be to select from one of multiple available static dictionaries. Notice too that the last 15 bytes of the dictionary in every unit are the same as the first 15 in the next. Therefore 93 % of the memory is duplicated. It should be possible to use non-static dictionaries with this architecture for example by implementing the dictionary memory with a shift register of bytes as seen in Figure 2.10.

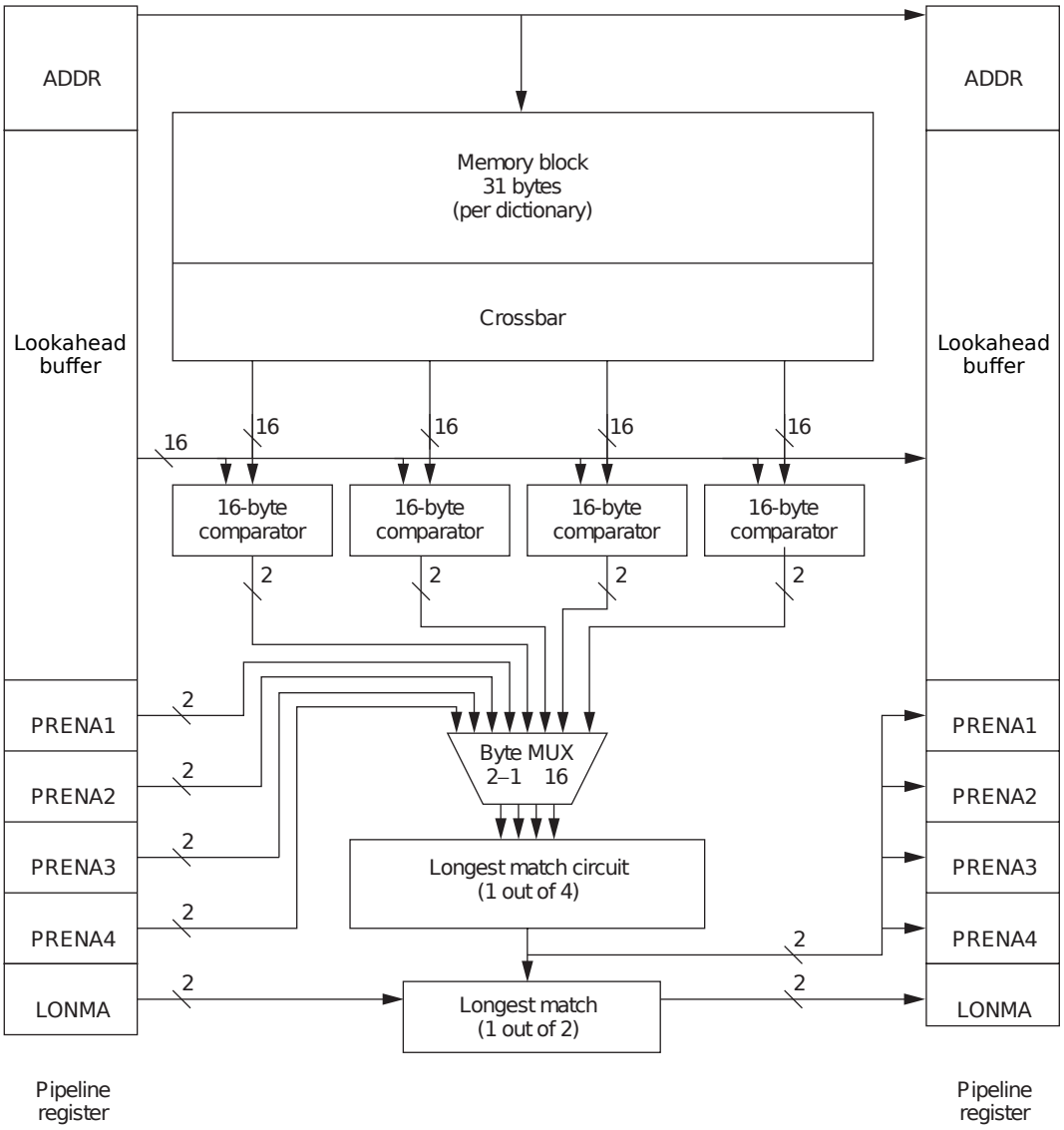


Figure 2.9: Pipelined dictionary architecture [66]. Bus widths are in bytes.

The example in that figure uses a lookahead of 3 bytes ($L = 3$). Therefore, each unit compares the lookahead with 3 sequences of the dictionary before moving to the next lookahead. When the lookahead moves from one unit to the following, the shift register shifts the dictionary by one byte in the opposite direction — in the figure the lookahead moves to the right and the dictionary to the left. With this solution no memory duplication is needed, but the number of used LUTs should increase.

In terms of performance a throughput of 315 MB/s is reported for this architecture. It is also referred that by making the pipeline more fine-grained a throughput of 1 GB/s is achieved. A total of 256 processing units are used, resulting in an area of 127k 6-input LUTs for the non-optimised implementation and 116k for the optimised one.

2.5.2 Architectures Using Hashes for Searching

While hashless architectures blindly compare the full contents of the window to the lookahead, the architectures in this category only compare portions of the window that are more likely to have a match

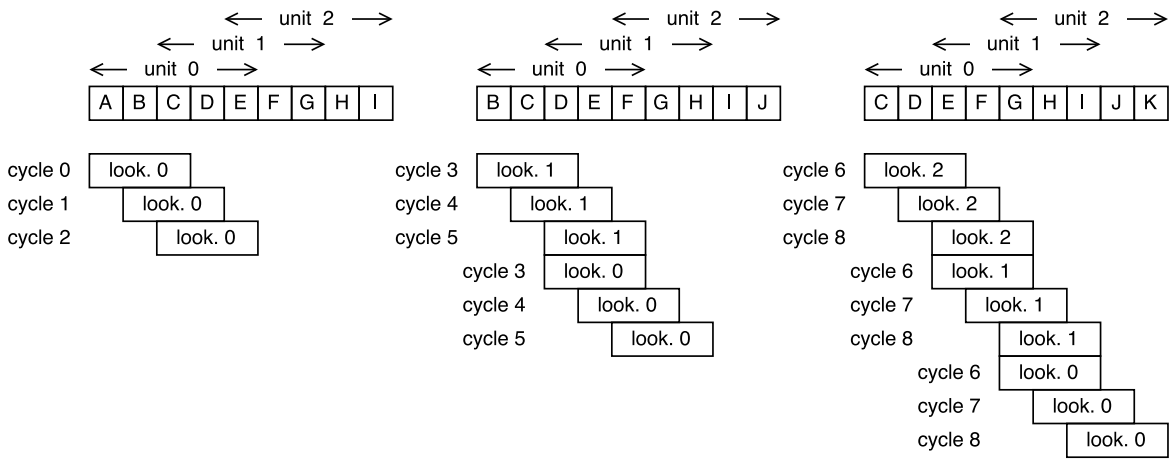


Figure 2.10: Scheduling of the comparisons in each processing unit in order to use a non-static dictionary, for $L = 3$. Note the dictionary moves one position from right to left for every 3 cycles.

for the current lookahead. This is accomplished by applying some hash function to the lookahead and using a hash table to store, for each hash value, positions of the window whose hash corresponds to that value. Figure 2.11 shows a diagram of a general architecture employing hashes. The way it works is as follows: the current lookahead is hashed and the hash value is used to index the hash table; the hash table outputs a position from the window that matches this hash value; this position is used to index the window, from which a sequence with the same length as the lookahead is read; this sequence is compared with the lookahead and the length of the match is calculated. The position given by the hash table is transformed into a distance relative to the beginning of the lookahead, suitable to be used during Deflate's Huffman coding stage. In order to improve the compression ratio the hash table may contain multiple window positions corresponding to a certain hash value. We refer to the number of positions as the *depth* of the hash table. For depths greater than one, the several matches might be processed sequentially or multiple comparators, length calculators and offset to distance converters might exist in order to process various matches in parallel.

In general this type of architecture should use less hardware than a hashless architecture for the same dictionary and lookahead sizes and throughput. The lower amount of hardware resources can be traded for increased throughput. However, hashless architectures might be able to find some matches not found by some "hashful" architectures, due to the finite amount of references that the hash table in the latter might support.

2.5.2.A Implementation with Linked Lists

One way to implement the hash table is to use a linked list. It closely mimics the approach used in Gzip, which can be observed by reading Section 3.2.2. The hash table is organised using two memories, `head` and `prev`. The `head` memory stores, for each hash value, the last offset in the window where a sequence with that hash value exists. This position can be immediately used to index the window to get a candidate match. But it also doubles as a pointer to the second memory, which contains the remaining elements of the linked list. Each memory position in `prev` contains the offset for another window position with the same hash value. As seen in Figure 2.12 this offset also

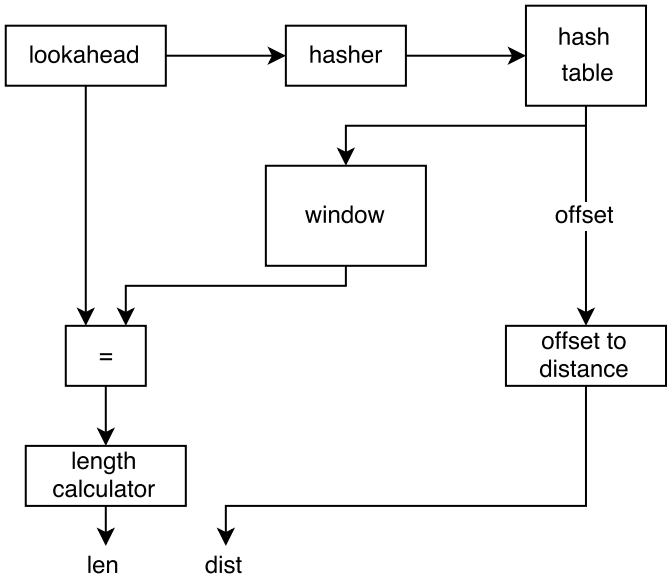


Figure 2.11: General hashful architecture

serves as the address for `prev` which contains the next element of the linked list. The end of each list is denoted by a memory word with null value.

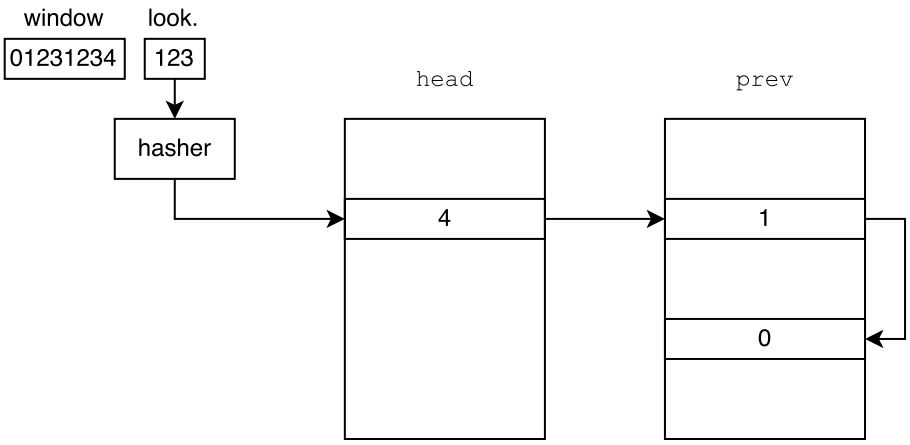


Figure 2.12: Example of addressing a linked list hash table

One disadvantage of using a hash table implemented as a linked list is that the positions for the matches are fetched sequentially, because a memory position must be accessed in order to retrieve the next one. Therefore, if several candidate positions exist their matches will be searched sequentially instead of in parallel. Because a null word is used to indicate the end of each list, another disadvantage of the linked list approach is that it could require "rotating" the `next` memory, i.e. for every W (the window size) input bytes the offsets that point out of the current window are zeroed. According to [67] this operation can represent from 25 to 75 % of the total running time of a hardware implementation. To mitigate this problem they propose adding k "generation bits" to each entry of the `prev` memory. This way the rotation is performed only once for each $2^k W$ bytes of input. They also suggest that the memory can be divided in several parts so that they can be rotated in parallel.

An implementation using a linked list does not require duplication of any of the blocks seen in

Figure 2.11. Furthermore, it does not require high throughput from the window and hash table memories, which in other architectures might lead to memory duplication in order to increase the throughput. Another advantage is that the depth for each hash value will be the same as the number of sequences in the window which have that hash. The increased number of candidate matches can result in better compression, as seen in Chapter 4.

The size of the `head` memory grows exponentially with the number of bits of the hash values and linearly with the number of bits required to store the offset for the window. The size of `prev` is proportional to the dictionary size and to the number of bits of the offsets. In practice, the size of the window (and therefore the length of the offsets) does not impact the size of the memories, because the data width of the memories is generally a multiple of 8 bits and the size of the dictionary is generally from 1 kB to 32 kB, resulting in offsets whose representation always requires two bytes.

An architecture of this type was proposed in [68]. This implementation only compares one byte of the window with one of the lookahead in each cycle. The dictionary and lookahead's sizes are the maximum supported by Deflate, i.e. $W = 32$ kB and $L = 258$ B. Unfortunately the throughput results are not clear, because the prototype uses an SD card to store the input data, which was found to be too slow to feed the architecture (the read bandwidth of the card is not reported). In terms of hardware usage the LZ77 part of the design uses 2077 LEs (logic elements), each containing a 4-input LUT.

Another hash-based architecture was proposed in [67]. Up to four bytes are compared during each cycle: the memory accesses are aligned to 32 bits, therefore the first comparison for each match can be of 1 to 4 bytes and, if the match is longer, the following comparisons will be of 4 bytes per cycle. Window sizes of 4 kB and 32 kB were used and $L = 258$ bytes. The reported throughputs are of 49 MB/s for the smaller dictionary and 46.2 MB/s for the largest. The effects of comparing a single byte per cycle were tested, resulting in throughputs of 30.3 MB/s and 25.9 MB/s for the two dictionary sizes, which represents a performance decrease of 63 to 78 %. Disabling the "generation bits" mentioned above was also tested, which lowered the throughput to 11.9 MB/s and 33.8 MB/s, respectively. Notice the decrease is more marked for the smaller dictionary, because it requires more frequent rotations of the hash table. The average number of cycles per input byte is two cycles. For the 32 kB dictionary the implementation area is of 2620 6-input LUTs.

2.5.2.B Multiple Matches per Cycle

In order to achieve high throughputs with a high compression ratio, several matches should be processed per cycle. This is not possible using a linked list hash table. Therefore, a new architecture for the hash table is needed, which must support storing and reading several candidate window positions in each clock cycle.

The simplest hash table organisation allowing reading multiple window positions corresponding to a single hash value is to have a fixed number of slots corresponding to each hash value. This can be implemented by using a memory with a data width sufficient to accommodate the number of bits needed to have several offsets in a single slot; or by using separate memories for each unit of depth. Which option is used will depend on the available memory configurations in the used device.

The multiple offsets for a hash value should be updated in a first-in first-out (FIFO) manner, so that the stored positions always correspond to the ones closest to the current lookahead. Consequently, the first implementation option is less desirable, as keeping track of which memory bytes to write would require extra logic. Therefore, separate memories are used to increase the depth, as seen in Figure 2.13. The total size of the hash memories increases exponentially with the number of bits of the hash value, linearly with the depth and remains constant with the number of bits of the offsets for the typical window sizes.

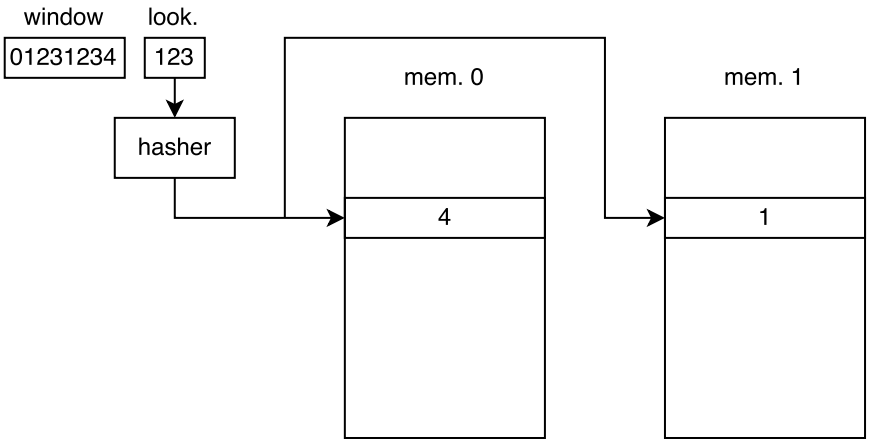


Figure 2.13: Example of addressing a multi-slot hash table implemented with two memories

Processing multiple matches per cycle implies reading the window memory multiple times per cycle. In FPGAs it is common for the BRAMs to support up to two accesses per cycle. These two accesses may be a combination of reads and writes, but only two address buses are available. Thus, memory duplication and/or increasing the memory operation frequency is generally necessary in this type of architecture.

Notice that the throughput of this type of architecture does not depend on the input stream. The throughput remains constant, irrespective of the input stream, because all the matches for a lookahead are processed in a constant number of cycles. This is unlike the architectures using a linked list, in which the number of cycles spent for a certain lookahead depend on the contents of the linked list.

The architecture described in [69] seems to fit in this category, but its description is very ambiguous. It uses a 16 kB dictionary divided into 16 separate BRAMs and the hash table seems to implement a depth of 2, using two separate BRAMs. A total of 4 matches seem to be processed per cycle. The throughput results reported are dependent on the input data, which is unexpected. It varies from 61 MB/s to 110 MB/s and the fastest throughputs are attained for highly compressible data sets. No hardware area utilisation is mentioned, nor is the size of the lookahead specified.

The previous descriptions assume several matches are processed per cycle, but that all of them correspond to a *single lookahead*. In order to further improve the throughput it is possible to compute multiple matches per cycle for *multiple lookaheads*. Works such as [70] and [71] can process 16 and 32 lookaheads in a single cycle. This puts the window, lookahead and hash table memories

under additional stress due to the high number of accesses required in each cycle. If only memory duplication is used to cope with the required bandwidth, the amount of needed resources becomes unacceptable. Consequentially, a more elaborate architecture is necessary.

In the next paragraphs a description of how this problem is solved in [71] is presented. Other implementations which process several lookaheads per cycle can be found in [70] and [72]. However, [70] is a reimplementation of [72] in OpenCL, which does not give sufficient details about the memory organisation; and the latter seems to have been a presentation in a conference, for which no published materials could be found.

In [71], a total of 32 lookaheads are processed in each cycle. Each lookahead is 32 bytes long. They implement a hash table depth of 1 and use 16 bits of hash size. Consequently, in each cycle 32 candidate positions must be read from the hash table. Assuming each BRAM has two ports it would be necessary to replicate the 2^{16} positions hash table 16 times, in order to perform all the needed reads and writes (note the read address is the same as the write address). Their solution is to divide the hash table into a series of banks and distribute the 2^{16} positions equally by the banks. Each bank is independent from the others, hence it is able to be accessed two times in a single cycle. They chose to use 32 banks. The 32 hash values must now be used to address the 32 banks. The least significant bits of each hash value are used to select the bank where the information about that hash can be found. It is clear that collisions can happen. In the worst (improbable) case all the 32 hashes will have the same lowest bits and therefore 32 accesses will be need from a single bank in a single cycle. The best case is that no more than two hash values correspond to the same bank in each cycle. Because of the possibility of collisions a crossbar switch is used to forward the hash values to their respective hash banks. This crossbar switch includes an arbiter which chooses up to two requests per bank per cycle. This hash table architecture is represented in Figure 2.14. An additional crossbar switch is then necessary to forward the outputs of the banks so that at the hash table output the order or the candidate positions match the order of the hash values at the input. Notice that the requests that are dropped by the arbiter tend to reduce the compression ratio that would be attained without those collisions. With 32 banks 64 candidate positions can be read per cycle, which should minimise collisions to a certain degree. However, in order to further reduce the number of collisions, the authors use a frequency for the hash table which is double from that of the rest of the datapath.

Since 32 lookaheads are processed per cycle and each lookahead is 32 bytes long, the window memory should be able to output 32 consecutive bytes from any address, in a total of 1024 bytes per cycle. It must also be updated with 32 new consecutive bytes. In order to accomplish this, the authors chose to replicate the window memory 32 times. Each replica outputs 32 bytes which are aligned so that any byte aligned address can be accessed in a single cycle. A window size of 64 kB is used, which means the Deflate64 variation of Deflate is implemented, as the original only supports dictionaries up to 32 kB. The lookahead memory is simply stored in 64 eight-byte registers.

The implementation in [70] uses a lookahead of 16 bytes and processes 16 lookaheads per cycle. The size of the dictionary is not specified. A throughput of 2.84 GB/s is reported and about 110 kALMs (adaptive logic modules) are used. On the other hand, in [71] 32 lookaheads of 32 bytes

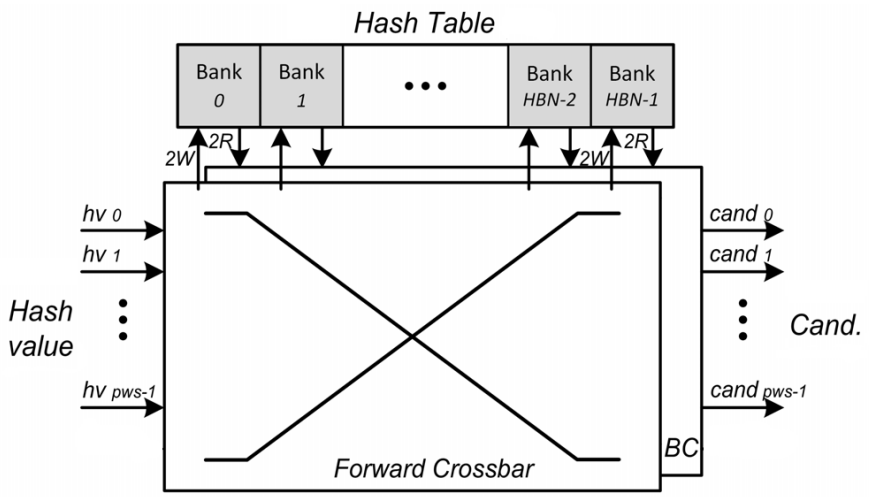


Figure 2.14: Multi-bank hash table architecture [71]

are processed per cycle, resulting in a a throughput of 5.6 GB/s, using 108 kALMs. This is the best throughput currently found in the literature. This throughput is about the double of the former, which is expected since it processes two times more bytes per cycle with a similar frequency. The mentioned areas include the Huffman coding parts of the architecture.

2.5.3 Summary of Implementations

Table 2.2 provides a summary of performance parameters for the implementations of the architectures described in the previous sections. In general the hashful architectures support longer dictionaries than their hashless counterparts, which results in better compression ratios, as will be seen in Chapter 4. In terms of throughput, both types of architectures can reach 1 GB/s. However, the hashful architectures have a better throughput per area ratio.

Table 2.2: Comparison metrics for several implementations of Deflate or LZ77

Ref.	Architecture type	Win. size	Look. len.	Thr. (MB/s)	Area	
[63]	Hashless (direct)	512	8	750	23743	4-input LUT
[64]	Hashless (direct)	?	?	1167	120000	6-input LUT
[65]	Hashless (systolic array)	1024	16	2	419	4-input LUT
[66]	Hashless (pipelined dict.)	4096	16	1024	116000	6-input LUT
[68]	Hashful (linked list)	32768	258	?	2077	4-input LUT
[67]	Hashful (linked list)	32768	258	46	2620	6-input LUT
[69]	Hashful (multi-match)	16384	?	62	?	
[70]	Hashful (multi-match)	?	16	2908	110000	ALMs
[71]	Hashful (multi-match)	65536	32	5734	108000	ALMs

2.6 Conclusion

This chapter introduced the main lossless data compression algorithms currently available. It also introduced the data compression benchmarks usually used to test the algorithms for both throughput and compression ratio. We concluded that due to its widespread use and relative simplicity compared

to other algorithms the Deflate algorithm is the most interesting case study for hardware acceleration. Hence, works from the last ten years concerning implementations of LZ77 or Deflate in reconfigurable devices were presented and evaluated, and will serve as a basis for the development of our proposed architecture.

3

Deflate and Gzip

Contents

This chapter describes several aspects of the Deflate algorithm and format along with details of Deflate's implementation in the Gzip software.

3.1 Deflate

3.1.1 The Deflate Algorithm

Deflate results from a combination of the LZ77 algorithm with Huffman coding. As described in Section 2.2, the LZ77 algorithm allows to describe a stream of data as a combination of distance-length pairs and literals. The distance-length pairs point to a *dictionary* (also called *window*) consisting of the last W bytes of input data, relative to the start of the sequence being currently compressed, i.e. the *lookahead*.

In the traditional LZ77 description the output consists entirely of distance-length-literal triples. The literal in the triple refers to the literal that comes right after the end of the previous matched sequence. When no match is found the first symbol of the lookahead is still encoded as a triple with distance and length of zero. The LZ77 algorithm does not specify how the triples should be encoded. The most obvious way would be to encode each of the elements in the triple with a fixed number of bits depending on the number of values that the element can take. However, using an entropy coding method will generally result in better compression ratios.

Deflate uses Huffman coding to encode the output of LZ77. The literals and lengths are gathered into a single alphabet, while the distances constitute another. This way, both the literal-length codes and the distance codes are encoded with a variable number of bits, according to their relative frequency. Encoding a match constitutes of emitting a length code and a distance code, hence forming the distance-length pair. Only a double is emitted, unlike in the traditional LZ77 which would also emit a literal. On the other hand, when no match is found only the code for a literal is produced. The decoder figures whether a code is for a literal or a length. If is is for a literal it decodes that literal. Otherwise it will read the next code in order to retrieve the match distance and can then output the matched sequence. Even though the codes have a variable number of bits, the decoder knows where each code ends because Huffman codes are a prefix code.

In Deflate the lengths range from 3 to 258 and the distances from 1 to 2^{15}. Instead of using one code for each of these lengths and distances — which would result in fairly big Huffman codes — Deflate assigns a range of lengths or distances to each code. Then a variable number of extra bits are used to distinguish the several lengths/distances that fall into each code. Table 3.1 and Table 3.2 respectively show the alphabets for lengths and distances along with the respective number of extra bits. Regarding the first table, the symbols do not start from zero because symbols 0 through 255 are for literals and 256 is for the end-of-block symbol.

Two types of Huffman coding can be used for the encoding phase: static Huffman and dynamic Huffman coding. The latter changes on a per-block basis, i.e. the output is divided into blocks each of which is encoded with a particular Huffman code. The code for each block is stored in a compact manner at the beginning of the block. A new block is started when a new code is deemed necessary

Table 3.1: Symbols and extra bits for the lengths

Symb.	Extra	Len.	Symb.	Extra	Len.	Symb.	Extra	Len.
257	0	3	267	1	15,16	277	4	67 – 82
258	0	4	268	1	17,18	278	4	83 – 98
259	0	5	269	2	19 – 22	279	4	99 – 114
260	0	6	270	2	23 – 26	280	4	115 – 130
261	0	7	271	2	27 – 30	281	5	131 – 162
262	0	8	272	2	31 – 34	282	5	163 – 194
263	0	9	273	3	35 – 42	283	5	195 – 226
264	0	10	274	3	43 – 50	284	5	227 – 257
265	1	11,12	275	3	51 – 58	285	0	258
266	1	13,14	276	3	59 – 66			

Table 3.2: Symbols and extra bits for the distances

Symb.	Extra	Dist.	Symb.	Extra	Dist.	Symb.	Extra	Dist.
0	0	1	10	4	33 – 48	20	9	1025 – 1536
1	0	2	11	4	49 – 64	21	9	1537 – 2048
2	0	3	12	5	65 – 96	22	10	2049 – 3072
3	0	4	13	5	97 – 128	23	10	3073 – 4096
4	1	5,6	14	6	129 – 192	24	11	4097 – 6144
5	1	7,8	15	6	193 – 256	25	11	6145 – 8192
6	2	9 – 12	16	7	257 – 384	26	12	8193 – 12288
7	2	13 – 16	17	7	385 – 512	27	12	12289 – 16384
8	3	17 – 24	18	8	513 – 768	28	13	16385 – 24576
9	3	25 – 32	19	8	769 – 1024	29	13	24577 – 32768

to improve the compression ratio. The dynamic Huffman coding used in Deflate is outside of the scope of this work. Further details can be found in RFC 1951 [27].

Blocks coded with Deflate's static Huffman coding use the code seen in Table 3.3. The length of the codes ranges from 7 to 9 bits. Symbols in the range 0–143 use 8-bit codes while 9 bits are used for the 144–255 range. Since the literals correspond to symbols from 0 through 255, the 0–143 range includes all the ASCII characters. Therefore, Deflate's Huffman code assumes the probability of finding ASCII characters in the input is higher than that of finding non-ASCII characters, which makes it more efficient at encoding "plain text" streams rather than "binary" streams. The codes in the table are not used to encode symbols in the distance alphabet: since only 30 distance symbols exist, fixed-length codes of 5 bits are used to encode the distance symbols. The value in the 5 bits corresponds to the binary representation of the symbol number.

Figure 3.1 shows that: 1) the distance generally contributes the most to the total of encoded bits; 2) lengths of 258 (the maximum) are encoded with few bits in order to improve the compression ratio of highly redundant data; 3) the maximum number of bits encoded per distance-length pair is 31 bits. Although the number of encoded bits might be more than that of encoding three literals, it is always less than that of encoding four literals. Gzip uses a simple optimisation to minimise the wastefulness of encoding matches of length 3 with very long distances (see Section 3.2.5).

Because the static codes are prespecified there is no need to encode them into the output, as is the case with the dynamic codes. In addition there is no need for the encoder nor decoder to build

Table 3.3: Static Huffman code used in Deflate for the alphabet containing literals, lengths and the end-of-block symbol

Symb. value	Bits	Codes
0 – 143	8	`00110000` – `10111111`
144 – 255	9	`110010000` – `111111111`
256 – 279	7	`0000000` – `0010111`
280 – 287	8	`11000000` – `11000111`

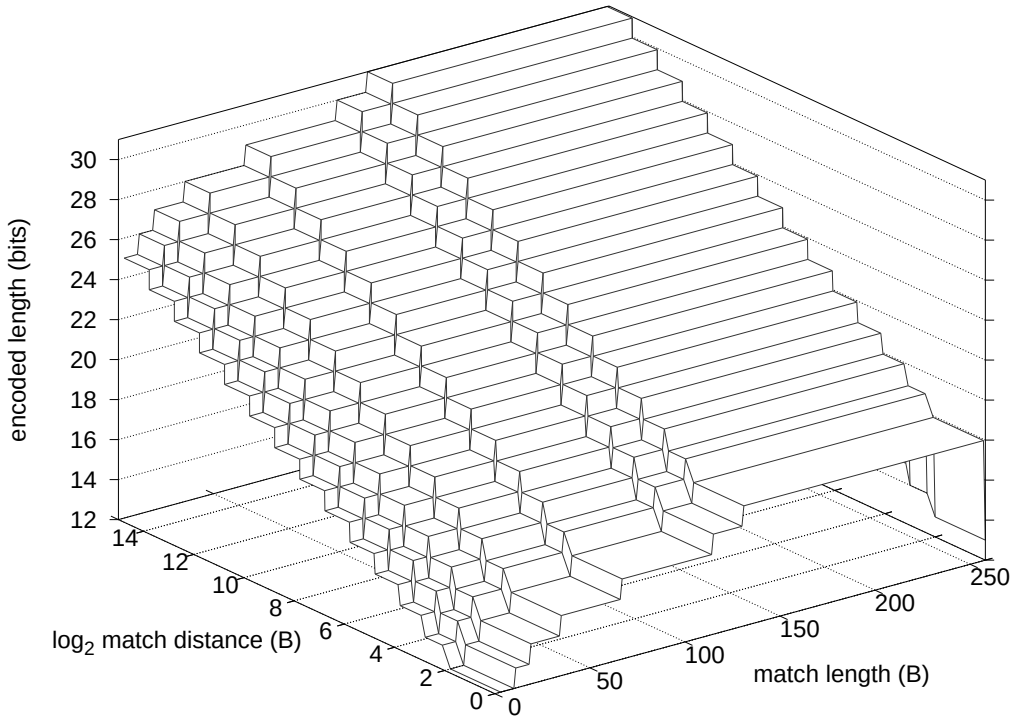


Figure 3.1: Number of encoded bits for distance-length pairs

the codes and update them. This, however, comes with the disadvantage that in most cases lower compression ratios are achieved when using static Huffman. To explore the losses in compression ratio that can be expected from using static Huffman, the simulator described in Section 4.1 was used to explore the maximum compression ratios achievable with static codes versus dynamic ones. The results for the test corpora can be seen in Table 3.4. The results for dynamic Huffman codes were obtained using `gzip -8`, which searches at most 1024 matches; and the same depth was used in the simulator. Dynamic codes perform 13 to 25 % better than their static counterparts.

Table 3.4: Comparison of the compression ratios achievable using Deflate's static and dynamic Huffman codes

	Comp. ratio		
Corpus	Static	Dynamic	Difference (%)
Calgary	2.62	3.08	17.4
Canterbury	3.06	3.82	24.8
Enwik8	2.35	2.74	16.5
Silesia	2.75	3.13	13.6

3.1.2 The Deflate Format

The format produced by Deflate is simple. The first thing to note is that the output stream is divided into blocks. Each block is delimited by the end of the previous block (or the beginning of the stream) and the end-of-block symbol. There are no length limits for each block: it can be empty (i.e. contain only the end-of-block symbol) or it can be as long as needed.

Each block starts with a header of 3 bits. The first bit in a block indicates whether this block is the last block in the stream or not. If it is, the bit is set to one, otherwise its value is zero. The two following bits indicate the type of the block, which can be one of three: uncompressed block (`00`), static Huffman block (`01`) or dynamic Huffman block (`10`). The code `11` is not used and will result in an error if found by the decoder. A Deflate stream may contain a combination of these types of blocks. It may, for example, contain mostly dynamic Huffman blocks but use uncompressed blocks for a part of the stream with particularly high entropy.

After the header, the remaining of the block consists of literal-length and distance codes with their respective extra bits. Literals are encoded with their code, while for distance-length pairs the length is encoded first, followed by its extra bits (if any) and then the distance code and its extra bits. The packing units of the output are bytes, which are filled starting from their least significant bit (LSB) to the most significant bit (MSB). Codes are packed starting from their MSB, while the extra bits are packed starting from the LSB.

Figure 3.2 shows an example of how a certain sequence of literals and matches is packed into a Deflate block (Tables 3.1, 3.2 and 3.3 help to understand this example). The first encoded bit is set, indicating that this is the last block. Next, the block type `01` is encoded, signalling this as static Huffman block. The literal for the character A corresponds to symbol number 65 in the literal-length alphabet and from Table 3.3 the literals in the range 0–143 are encoded with 8 bits, with code `00110000` (decimal 48) corresponding to symbol 0. Therefore, the code for A is `01110001` (decimal $113 = 48 + 65$). Since the codes are packed with their MSB first it is shown as `10001110` in the example.

Next, a match with distance 1 and length 21 is encoded, which might look strange since the distance is less than the length. Nevertheless, this is valid and it means that part of the lookahead itself is repeated. From Table 3.1, length 21 corresponds to symbol number 269, which is coded as `0001101` and packed as `1011000`. Length 21 also requires 2 extra bits: since symbol 269 contains lengths in the range 19–22 and $21 - 19 = 2$ the extra bits will be `10`. The same process is used to encode distance 1 as `00000` with no extra bits, according to Table 3.2 and to the fact that all distance symbols are encoded with 5 bits. The remaining literal and matches are encoded in the same manner. When the input stream ends, Deflate must also terminate the block. Thus, it produces the code for the end-of-block symbol, `0000000`.

Input stream:

```
AAAAAAAAAAAAAAAAAAAAAA00000000000AAAAAAAAAAAAAAAAAAA
```

LZ77 output:

```
0x41 (1,21) 0x30 (1,10) (31,20)
```

Deflate output:

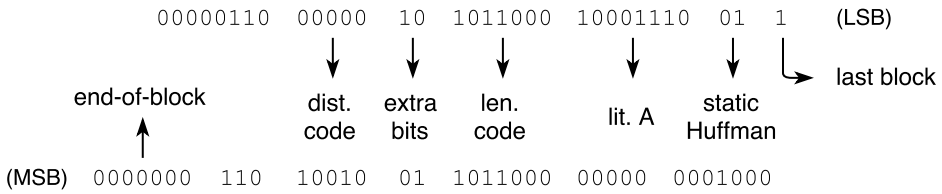


Figure 3.2: Example of encoding an input stream as a Deflate block

3.2 Gzip

Deflate streams are commonly packed into another format with more features. Gzip is the name of a free compression software and respective format which use Deflate. This section describes the Gzip format and some aspects of how it implements Gzip, including some optimisations.

3.2.1 The Gzip Format

Gzip streams add a header and a footer to a Deflate payload. The header supports features such as specifying the original file name, its modification time, adding a description of the file and the operating system where it was produced. The header includes a field specifying the compression method used, which means that Gzip files can support other types of payloads, but only Deflate is used. The footer contains a cyclic redundancy check (CRC) of the uncompressed data — which allows to detect errors in the uncompressed data — as well as the original size of the compressed stream. The specification of Gzip is found in RFC 1952 [28].

Figure 3.3 shows a minimal example of a Gzip file. This file uses the shortest valid Gzip header (i.e. no optional fields are included) and the payload is an empty Deflate block. The first two bytes constitute the `ID`, which is a “magic number” identifying the file as a Gzip file. The following byte, `CM`, is the compression method used for the payload, for which RFC 1952 specifies only the value 8, meaning Deflate compression is used. Some the bits of the `FLG` field can be set in order to specify that extra fields for the header are present. For example, the most commonly set bit, bit 3, indicates that the optional field containing the original file name is present. The next four bytes store the modification time of the compressed file in seconds relative to the Unix epoch (January 1, 1970), in little-endian ordering. A null modification value means that no modification time was included. The `XFL` byte allows to specify extra flags for the used compression method, while the byte `OS` specifies the operating system where the file was compressed (for example `03` is for Unix and `FF` means the system is “unknown”). The footer follows the Deflate payload, which (if needed) is padded with null

bits so that the footer starts in a byte-aligned position. The first four bytes of the footer store the CRC-32 of the uncompressed data, obtained using the CRC-32 polynomial whose reverse representation is `EDB88320`. Finally, the last four bytes are the size of the original stream modulo 2^{32}. These two fields as stored using little-endian ordering.

```
Header:   1F8D  08  00  00000000  00  03
           ID   CM  FLG   MTIME   XFL  OS

Payload:  0300

Footer:   00000000  00000000
           CRC32     ISIZE
```

Figure 3.3: Header and footer for a minimal Gzip file. The Deflate block is empty.

3.2.2 Hashing in Gzip

Gzip's speed comes in part from how it searches for matches. As it reads the input, it hashes every 3 bytes from it and stores the locations that match each hash value. When it starts searching for a new match it hashes the first 3 bytes of the lookahead buffer. Then it accesses each of the locations in the window whose hash is the same. For each of those locations Gzip: 1) verifies if the 3 bytes do indeed match, which may not be the case due to hash collisions; 2) compares the window with the lookahead to find the length of the match.

The hash function used by Gzip is a recursive hash, i.e. a hash that is calculated for a n-gram based on the hash for the previous n-gram and the new symbol in this n-gram. In Gzip, trigrams ($n = 3$) are used, because the minimum allowed match length is three bytes. More specifically, the type of recursive hashing used is called *hashing by cyclic polynomial*. In this type of hash, multiplications can be computed with shifts and additions with exclusive-ors, resulting in reduced calculation times.

Gzip hashes a trigram (s_i, s_{i+1}, s_{i+2}) from a sequence of symbols $(s_0, \ldots, s_{N-1})$ using the following function

$$h(i) = \left[\left[\left(\left(2^5 s_i\right) \oplus s_{i+1}\right) 2^5\right] \oplus s_{i+2}\right] \bmod 2^{15}. \tag{3.1}$$

The multiplications by 2^5 amount to left shifts of 5 bits, while $x \bmod 2^{15}$ means x is truncated to its lowest 15 bits. The hash for the trigram following the one in position i can be readily calculated with

$$h(i+1) = \left[\left(2^5 h(i)\right) \oplus s_{i+3}\right] \bmod 2^{15}. \tag{3.2}$$

As expected from a recursive hash, only the value of the previous hash and the new symbol are needed. Also, the symbol s_i does not influence $h(i+1)$, because its contribution is cleared by the modulo operation.

All the positions in the window that contain trigrams hashing to the same value are stored in a linked list. To implement this Gzip uses two arrays: `head` and `prev`. The `head` array is indexed by

hash values and contains the most recent position in `window` where a trigram with a certain hash was found. The `prev` array is indexed by a position and contains another previous position where another trigram with that hash is found; or 0 if no more positions for trigrams with that hash were stored. In other words, `prev` stores linked lists (stored in an array) of positions with the same hash, while `head` stores the position in `prev` where each list begins. These linked lists are called "hash chains" in Gzip's source. Figure 3.4 shows an example of a hash chain. Because the window size is 32 kB, at least 15 bits must be used to refer to positions in it. Both `head` and `prev` store 2^{15} positions and therefore they are declared with a size of 64 kB.

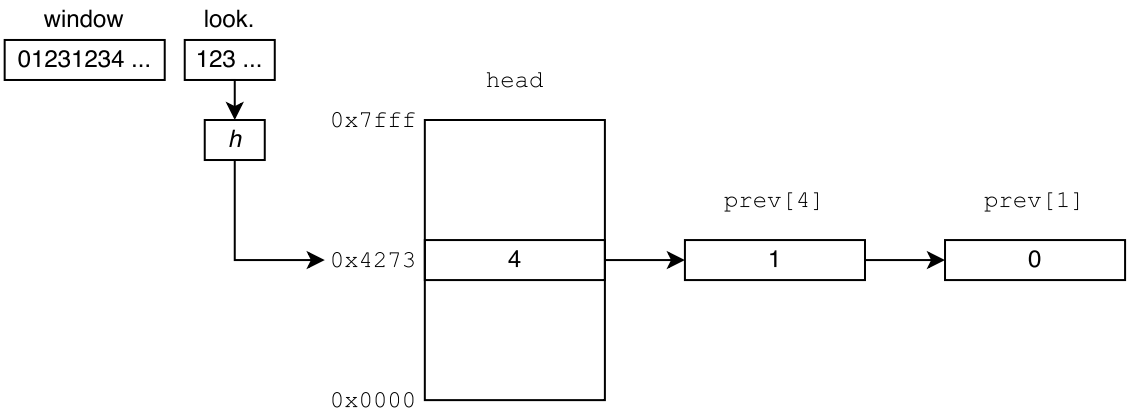


Figure 3.4: Example of a hash chain containing two positions with the same hash as the current lookahead

In Gzip the input is read to a buffer with twice the size of the window. When the buffer becomes full, the second half is moved to the first. When this happens, the values stored in `head` and `prev` must be updated to remain valid: the size of the window, `WSIZE`, is subtracted from each value. Values that would become negative are set to 0, effectively removing from the hash chain positions that are too old. This update is $\mathcal{O}(1)$, considering that 2^{16} values are updated for every 32 kB read. Insertion into the hash chain is also a constant time operation as it always involves updating just one value both in `head` and `prev`.

3.2.3 Compression Levels

Gzip supports nine compression levels which allow to trade compression speed for improved compression ratio, by limiting how much time Gzip spends searching for matches. Levels range from 1 (the fastest) to 9 (best compression ratio). Executing `gzip -6` uses compression level 6, which is the default level.

Four parameters (`good`, `lazy`, `nice` and `chain`) are associated with each level, as seen in the portion of `deflate.c` in Listing 3.1. The `chain` parameter defines the maximum number of sequences that can be searched for a certain lookahead. If a match is found whose length is at least `nice` no more matches are searched for the current lookahead. When the match length for the previous lookahead is greater than or equal to `good` the value of `chain` used for the current lookahead is halved. The `lazy` parameter has a different meaning depending on the level: for levels 1 to 3 only matches with length greater than `lazy` are inserted into the hash table; while for levels 4

to 9 Gzip only tries to find a match for the current lookahead if the match for the previous lookahead is less than `lazy`.

The `chain`, `good` and `nice` parameters reduce the number of searched sequences for the current lookahead and consequently the time spent looking for matches. For levels 1–3 `lazy` skips the insertion of sequences that would probably not result in significant compression, while it also prevents the hash chain length from increasing due to those sequences. This reduces the time spent with insertions and with future searches for matches in this chain. For levels 4–9 `lazy` allows to skip the lazy match evaluation for some matches, which can obviously reduce the execution time.

```
local config configuration_table[10] = {
/*      good lazy nice chain */
/* 0 */ {0,    0,  0,    0},  /* store only */
/* 1 */ {4,    4,  8,    4},  /* maximum speed, no lazy matches */
/* 2 */ {4,    5, 16,    8},
/* 3 */ {4,    6, 32,   32},

/* 4 */ {4,    4, 16,   16},  /* lazy matches */
/* 5 */ {8,   16, 32,   32},
/* 6 */ {8,   16, 128, 128},
/* 7 */ {8,   32, 128, 256},
/* 8 */ {32, 128, 258, 1024},
/* 9 */ {32, 258, 258, 4096}}; /* maximum compression */
```

Listing 3.1: Portion of Gzip's `deflate.c` defining parameters for each compression level

3.2.4 Match Selection (Lazy Matching)

When more than one match is found for a lookahead it is necessary to choose one of them. The chosen match should be the one that best contributes to improve the compression ratio. Selections can be divided into two types: intra-lookahead and inter-lookahead. Intra-lookahead selection picks the best match for one lookahead alone (which can be seen as the best local lookahead), while inter-lookahead selection picks a combination of matches and literals that best cover the input. Gzip uses a combination of both, which is described in this section. Section 4.3 presents a performance comparison of match selectors, including the one used in Gzip.

Gzip's intra-lookahead selector picks the lengthier match whose distance to the lookahead is the shortest. Choosing the longest match results in covering more input symbols with a single match, which locally results in a better compression ratio. Furthermore, choosing the shortest distance may reduce the number of bits necessary to encode the match (recall Figure 3.1) which contributes to an additional improvement of the compression ratio.

The inter-lookahead selector must ensure that all input symbols are covered by either matches or literals and that no coverage overlap exists (for example two matches cannot cover the same symbol). The idea behind Gzip's selector is simple: if the length of the match for the current lookahead is less than or equal to the match length for the previous lookahead, then the previous match is encoded; if, however, the current length is better, then a literal is encoded to cover the last uncovered symbol and the algorithm continues by advancing the window by one unit and finding matches for the next

lookahead. In Gzip this technique is called “lazy match evaluation.”

Figure 3.5 shows two examples of a portion of an input stream. In the example at the top a match of length 3 will be found for the current lookahead, 123. This match is not immediately encoded and the processing continues with the lookahead at the next position for which the only match is 23. Since the length of 123 is better that match is encoded. The next uncovered byte is 6 so the processing continues the lookahead starting at that byte. For the example at the bottom the match 123 is found and the lookahead advances, as in the previous example. But then a longer match, 2345, is found. Consequently, a literal is encoded to cover byte 1 and the lookahead advances to 345. Note that in this case no bytes are skipped when advancing the lookahead, because a better match might exist once again at the next byte.

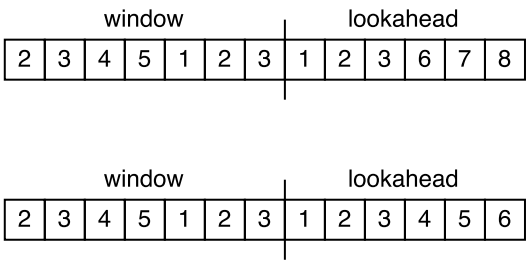


Figure 3.5: Example inputs to illustrate Gzip’s inter-lookahead selection. Top: a match for 123 will be encoded. Bottom: literal 1 will be encoded followed by a match for 2345.

There is one last detail: after a match is encoded — and therefore some bytes of the input are skipped — no bytes of the input will be left uncovered. For this reason, for the first processed lookahead after a match nothing can be encoded. Instead, the algorithm just takes note of the best match for that lookahead and moves to the one at the next byte. From there it continues as usual, encoding either a literal or a match.

The previous description of Gzip’s inter-lookahead selector can be implemented using a state machine based on three variables: the previous match (`prev`), the current match (`curr`) and a flag indicating whether there are uncovered bytes (`can encode`). Table 3.5 shows how `prev` and `can encode` are updated for the next state depending on the current one. It also shows whether a literal or match should be encoded in the current state. From this table it is possible to derive a compact pseudocode which describes the same transitions. Listing 3.2 shows that pseudocode, which matches the actual code used by Gzip.

3.2.5 The “Too Far” Optimisation

Figure 3.1 shows that a match of length 3 (the minimum length) may be encoded with up to 25 bits. However, encoding 3 literals requires only $3 \times 7 = 21$ bits in the best case and $3 \times 8 = 24$ in the worst. This means that the Deflate format allows the encoding of a match of length 3 to be less efficient than encoding three separate literals. Therefore, Gzip handles matches of length 3 specially: a match of length 3 is only considered valid if its length is less than or equal to the `TOO_FAR` parameter, whose value is 4096. Matches of length 3 at greater distances are encoded as three literals. The reasoning

behind the value of 4096 is that a match of length 3 with that distance is encoded with 22 bits — only one more than the 21 bits of the best literal case — while for a distance of 4097 the encoded length already increases to 23 bits. Consequentially, this rule saves bits for most length-3 matches, while not rejecting too many matches of such length.

Table 3.5: Next values for `prev` and `can encode` based on the current state. The "encode" column shows what should be encoded in the current state.

Current cycle					Next cycle	
`prev`	`curr`	`curr > prev`	`can encode`	Encode	`prev`	`can encode`
X	X	X	`false`	—	`curr`	`true`
`none`	`none`	—	`true`	`lit`	`curr`	`true`
`none`	Some	—	`true`	`lit`	`curr`	`true`
Some	`none`	—	`true`	`prev`	`none`	`false`
Some	Some	`false`	`true`	`prev`	`none`	`false`
Some	Some	`true`	`true`	`lit`	`curr`	`true`

```
if prev != none and (curr == none or curr <= prev):
    encode prev
    prev = none
    can encode = false
else if can encode == true:
    encode lit
    prev = curr
else:
    prev = curr
    can encode = true
```

Listing 3.2: Pseudocode for Gzip's inter-lookahead match selection

3.3 Conclusion

The Deflate lossless compression algorithm combines LZ77 and static or dynamic Huffman codes. Gzip is a popular compressor which implements Deflate. It uses several techniques to guarantee fast compression speeds while attaining good compression ratios. It also offers options to trade speed for better compression.

The next chapter explores how parameters such as the size of the dictionary and the match selection method influence the compression ratio.

4

Design Exploration

Contents

There are several aspects of the Deflate algorithm that can be adjusted. These adjustments can impact the throughput of the implementation as well as the compression ratio. In this chapter we study the impact of varying Deflate parameters on the compression ratio and the hardware resources needed to implement the accelerator. It is assumed that a constant number of input bytes is processed per cycle, such that the throughput is constant for a certain clock frequency. The Huffman part of the algorithm is considered static and uses the static code described in RFC 1951.

The graphs presented in this chapter that feature compression ratio values refer to the `enwik6` corpus — the first 10^6 bytes of the English Wikipedia — unless noted otherwise. The conclusions that can be drawn from the results for the Calgary, Canterbury and Silesia corpora are similar to the ones for `enwik6`.

4.1 The Compression Simulator

In order to test the effects of each parameter in the compression ratio, a software compression ratio simulator was developed. This simulator reads a stream of bytes and when the stream ends it outputs information about the size of the input stream, how many bytes it would compress to with the current parameters and the corresponding compression ratio.

The general workflow of the simulator is as follows: it reads some input into the lookahead, hashes it, consults a hash table for positions in the dictionary that might have a match for the sequence in the lookahead, searches those positions for matches, selects the best match (if any) according to some criteria, commits a match or a literal and finally updates the hash table; and the process repeats.

The simulator accepts options to configure several parameters: the size of the dictionary, the minimum and maximum lengths for a match, the depth of the hash table and the bit-length of the hash values. The depth of the hash table defines the number of alternative dictionary positions the hash table stores for a certain hash value. The records for a certain hash value are updated in a round-robin way, such that only the most recent positions are stored. Behaviours of the simulator such as best match selection can be altered by changing the functions assigned to the function references that are called.

4.2 Match Length, Window Size and Hash Table Depth

The main parameters that can be adjusted are the maximum length each match can have, the size of the dictionary and the hash table "depth". This section discusses the effects of these parameters on the compression ratio.

The maximum match length limits the longest sequences that can be compressed into a single codeword. A low limit implies that only short sequences can be captured by the compressor, while greater maximum lengths allow to capture longer redundancies in a stream. For highly redundant streams (all zeroes, for example) this is the limiting parameter for the compression ratio. In Deflate the match length ranges from 3 to 258. For Deflate with static Huffman the maximum compression ratio is approximately 158.7, because a match with length 258 and distance 1 takes 13 bits to encode.

The dictionary (or window) size can be thought of as the maximum match *distance*. It limits how far back in the input stream a match might point to. A longer dictionary allows to capture patterns farther apart in the stream. Rare sequences might not be compressed due to using a dictionary that is not big enough. Deflate specifies a relatively small dictionary of at most 2^{15} bytes. This means that if some sequence repeats only more than 32 kB apart in the stream it cannot be compressed.

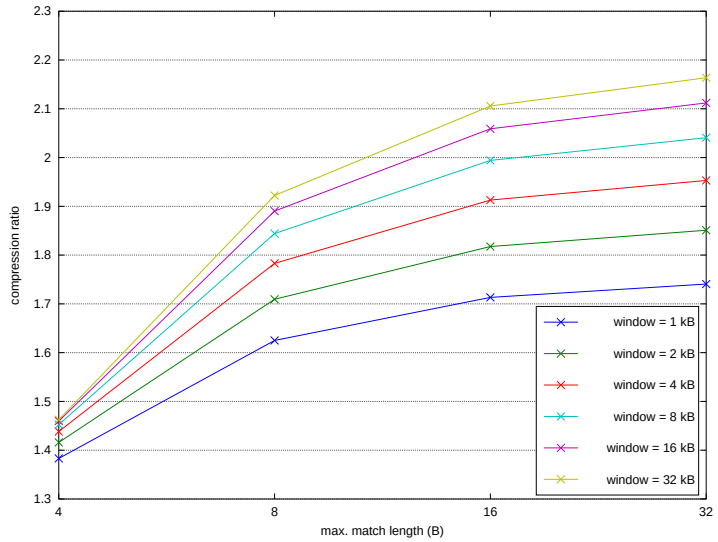
The depth of the hash table is the number of positions that are stored corresponding to a certain hash value. It determines the maximum number of alternative matches that can be found for some sequence. Some of these matches might be better than others and therefore it is desirable to have alternatives. Furthermore, alternative positions allow to reduce the number of matches lost due to hash collisions.

Figure 4.1 shows the results of varying the several parameters discussed above. The general conclusion is that, as expected, increasing each of those parameters results in better compression ratios. All subfigures indicate that a maximum match length of 4 bytes vastly reduces compression when compared to other greater lengths and that increasing other parameters for such a length is hardly advantageous. Figure 4.1a shows that doubling the window size significantly improves compression. From a 1 kB to a 2 kB window the compression ratio increases by about 6 %. This percentage decreases for larger windows and from 16 kB to 32 kB the gain in compression is of about 2 %. From the same figure it can be seen that doubling the maximum match length from 8 to 16 bytes improves the compression ratio by about 5 % for a 1 kB window, while for a 32 kB window the increase is of about 10 %. An increase in the match length from 16 to 32 B results in a gain of approximately 2 % and 3 % in compression for the same window sizes. The trend is that for a larger window enhancing the match length augments compression more significantly than for smaller windows, but that this effect becomes less evident as the longest matches increase in length. Figure 4.1c shows the same data as Figure 4.1a in a way that might easier to visualise.

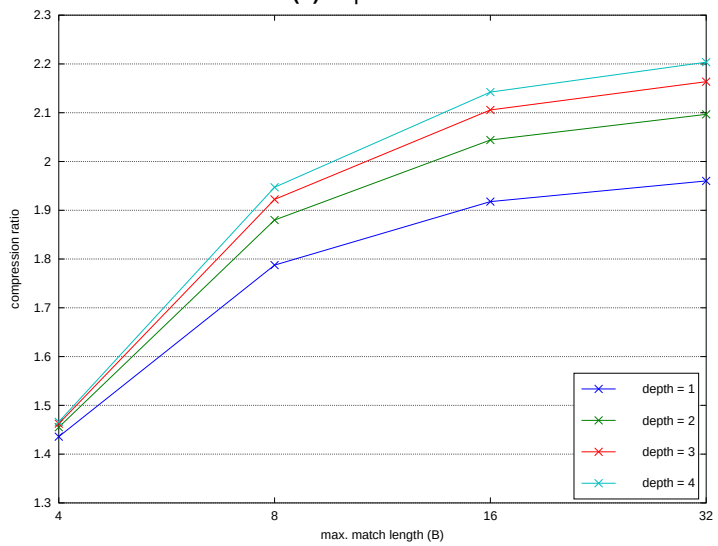
From Figure 4.1b we can see how different hash table depths impact compression. The most striking conclusion from that graph is that increasing the depth from 1 to 2 increases compression significantly more than from 2 to 3 and 3 to 4. In fact from 1 to 2 the increase is nearly 7 % while from *2 to 4* it is about 5 %. The gain seems to roughly halve for each increment in depth. Regarding the increase of the maximum match length for different depths, it can be seen that increasing the length for a higher depth results in a higher gain that for a smaller depth. Nonetheless, this effect is reduced as the maximum match length increases. This conclusion is similar to the one drawn from Figure 4.1a regarding the match length and window size. Also note that to produce the graph in Figure 4.1b a fixed window size had to be picked. The value chosen was 32 kB, but the appearance of the graph is similar for other window sizes.

4.3 Match Selection Method

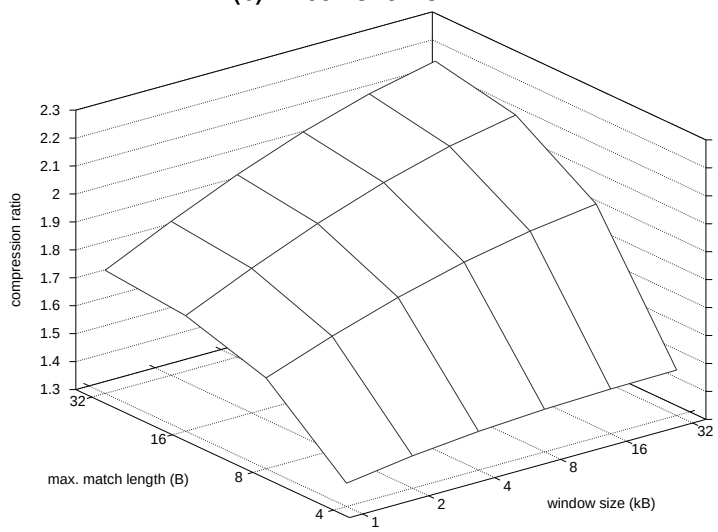
When several matches are found for a sequence it is necessary to choose which one to encode. This section describes several match selection methods and how they influence the compression



(a) depth = 3



(b) window size = 32 kB



(c) depth = 3

Figure 4.1: Effects of maximum match length, window size and hash table depth on compression ratio

ratio.

Simulations were made for five distinct selection methods. We classify four of these methods as "greedy", in the sense that after they search for matches for the sequence in the lookahead they choose a match from among those and immediately encode it. The fifth method is considered "lazy" since the encoding phase is deferred and happens only when the matched sequence is no longer in the lookahead.

The two first greedy methods consist in picking the match which takes less (Method 1) or more (Method 2) bits to encode. These two methods are not very clever, which can be easily be verified by comparing them with the remaining ones in Figure 4.2. The problem with Method 1 is that it can opt for matches with short lengths instead of lengthier matches, which would result in better compression. For example when choosing between two matches one of length 3 and other of length 30 with similar distances Method 1 will choose the one with length 3, losing an opportunity to encode 30 bytes into a single codeword. On the other hand, Method 2 fails in that it can select matches with an unnecessarily large distance given two matches with the same length. Because the match distance has a greater contribution to the codeword size than the match length (recall Figure 3.1), Method 2 performs even worse that Method 1. Other than these problems there is one more drawback for these two methods: to use them it is necessary to compute the encoded size for each match.

The other two greedy methods are closely related. Method 3 selects a match with the longest length. If several matches exist with the best length it uses the first found. Because the hash table in the simulator is implemented in a round-robin way and read in a fixed order, the matches with the same length are found essentially in a random order. The problem with this is that the compression ratio might be reduced if matches with unnecessarily long distances are used in place of matches with shorter distances, because for a constant length the codeword size grows with the match's distance. Method 4 solves this problem by selecting the match that has the shortest distance among the longest matches.

Method 5 is the same lazy method used by Gzip. In short, it tries to find a best match at the next input position instead of outputting the best match for the current position right away. This method is described in detail in Section 3.2.4.

Figure 4.2 compares the compression ratio attained by the several selection methods described above. From this figure it is apparent that Method 5 achieves the highest compression levels, followed by Method 4 and Method 3. Figure 4.2a reveals that for a depth of 1 all the greedy methods result in exactly the same compression ratio. This is to be expected because, with only one match to select from, all of them will pick the same match. On the other hand, Gzip's lazy method is 4.5 % better than the others for that same depth. This is justified because even when the depth is 1 Method 5 is not restricted to choose that match, as it can postpone the decision to after checking if a more advantageous match exists for the next input position. For depths greater than 1, Method 4 performs increasingly better than Method 3 as the depth is incremented. This is expected, since Method 4 always selects the longest match with less bits, while Method 3 may select a match with the same length but which takes more bits to encode. In other words, when the depth increases the

probability of Method 3 selecting the best match decreases, and therefore its compression ratio drops when compared to Method 4. Gzip's selection method compresses about 4 % more than Method 4 regardless of the hash table depth. This gain is not truly constant, as there is a very slight decrease as the depth grows: from a gain of 4.5 % for depth 2 it drops to 4.1 % for depth 4; and it only gets down to 3 % for depths close to 256.

One interesting thing to note from Figure 4.2a is that the compression ratio for Method 4 with depth 4 is roughly the same as that for Method 5 with depth 2. Table 4.1 further shows that the compression ratios of Method 5 for depth d are approximately (less than 1 % difference) the same as those of Method 4 with depth $2d$, for $d \in [2, 8]$. The conclusion is that for depths in this range Gzip's lazy method seems to indirectly double the depth.

Table 4.1: Comparison of the compression ratio values for some depths using Method 4 and Method 5. These values are for a maximum match length of 32 and 32 kB window size.

Method 5		Method 4		
Depth	Comp. ratio	Depth	Comp. ratio	Difference (%)
1	1.96	2	2.00	-2.35
2	2.09	4	2.11	-0.98
3	2.16	6	2.16	-0.29
4	2.20	8	2.20	0.05
8	2.28	16	2.26	0.87
9	2.29	18	2.26	1.04
32	2.36	64	2.31	2.38

Figure 4.2b shows that the compression gain from Method 3 to Method 4 remains nearly constant as the maximum match length increases. The same graph also reveals that as the length increases Method 5 performs better than Method 4. The increased gain can be explained because as the length limit is relaxed Method 5 may be able to find lengthier matches not only for the current position (as Method 4) but also for the subsequent ones. Figure 4.2c reveals a somewhat linear increase in the compression ratio with doubling window sizes, irrespective of the selection method used. Nonetheless Method 5 has a higher slope than the other methods.

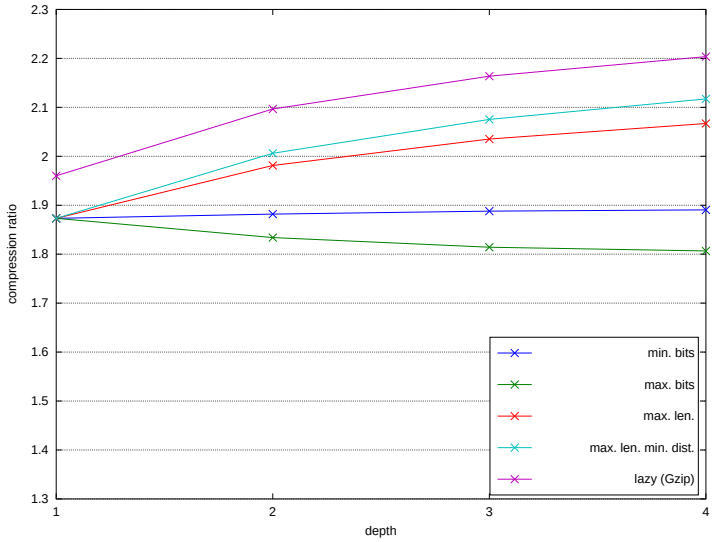
4.4 Hash Size

To speed up the search for matches, a hash table keeps references for the positions in the stream that might contain prefixes for some sequences. This section explores how varying the bit-length of the hash values influences compression.

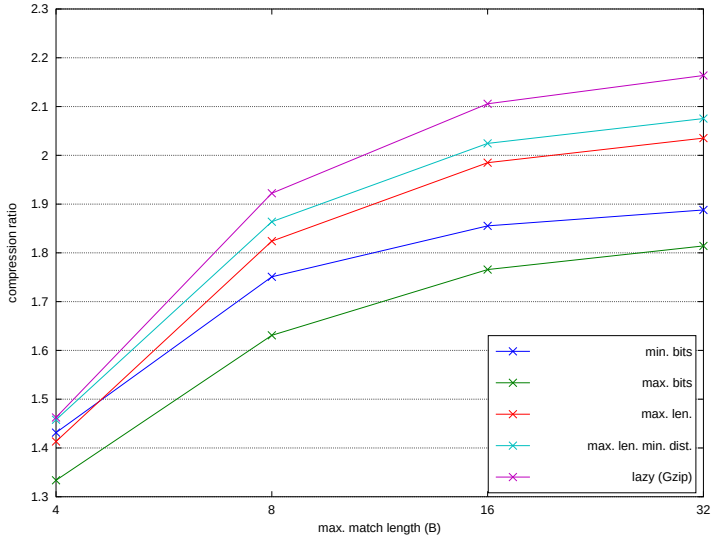
Two different hash functions were tested. The first is the hash function used in Gzip, which is described in Section 3.2.2. The second function is the one presented in [70], which hashes a sequence $(s_i, s_{i+1}, s_{i+2}, s_{i+3})$ using the function

$$h(i) = \left(2^2 s_i\right) \oplus (2s_{i+1}) \oplus s_{i+2} \oplus s_{i+3}. \tag{4.1}$$

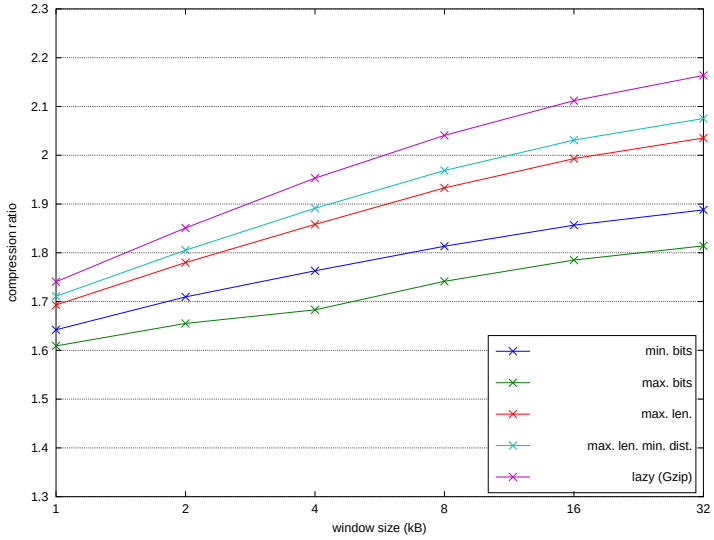
This function hashes a four-gram, unlike Gzip's which hashes trigrams. As a consequence using this hash captures matches with length equal or greater to 4, while Gzip also finds matches of length 3.



(a) max. length = 32, window size = 32 kB



(b) depth = 3, window size = 32 kB



(c) depth = 3, max. length = 32

Figure 4.2: Effects of match selection methods on compression ratio

The practical use of reducing the number of bits of the hash values is in reducing the memory needed to store the hash table. Depending on how the hash table is implemented, a reduction of one bit in the hash should reduce the needed memory up to half the original size. The consequence of reducing the hash size is that more collisions occur and therefore some matches are lost, reducing compression.

Figure 4.3 shows the effect of the hash size on the compression ratio. As expected, the compression ratio decreases as the hash size is reduced. The lines plotted for the several depths are not parallel: the absolute value of the slope is higher for lower depths. The reason is that for smaller depths it is more probable that the slots in the hash table become saturated due to the increase in collisions. For high depths the lines should be almost horizontal for the first reductions of the hash size, before the ratio starts to fall significantly. In other words, the greater the depth, the more the compression ratio resists to hash size reductions.

Figure 4.3 also shows results for the hash function in Equation 4.1. From the graph it can be seen that for this function relaxing the restriction on the number of bits does not improve the compression ratio. This is because the function was designed to produce 10-bit hash values. For this number of bits the function performs from 2 to 3 % worse than Gzip's function with the same number of bits. It is worth to note that in [70] this hash is used in conjunction with dynamic Huffman coding, which compensates the compression ratio for the inefficiencies of the hash.

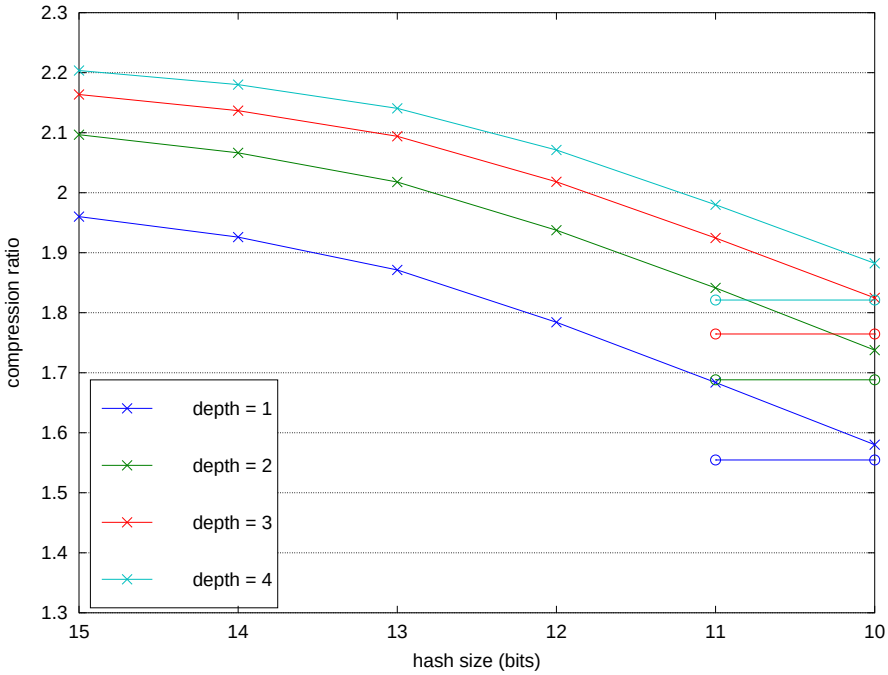


Figure 4.3: Effect of the hash size on the compression ratio. Crosses represent points for Gzip's hash function; circles are for the function in Equation 4.1. The reductions in the hash size are performed by zeroing the most significant bits. Maximum match length is 32 B and window size is 32 kB.

The shortening of the hash values that lead to the results in Figure 4.3 were obtained by progressively zeroing the most significant bits of the hash, i.e. only the least significant bits were kept. An alternative way to reduce the size of the hash would be to shift the hash value to the right, which

would maintain the *most* significant bits. While the first approach progressively loses information about the first byte in the lookahead and then about the first and second byte (they are exclusive-ored), the second approach loses information purely about the third byte. Figure 4.4 compares the compression ratios for both approaches. It reveals that the difference is not very significant: in general using the most significant bits results in less compression, but the difference is below 1 %.

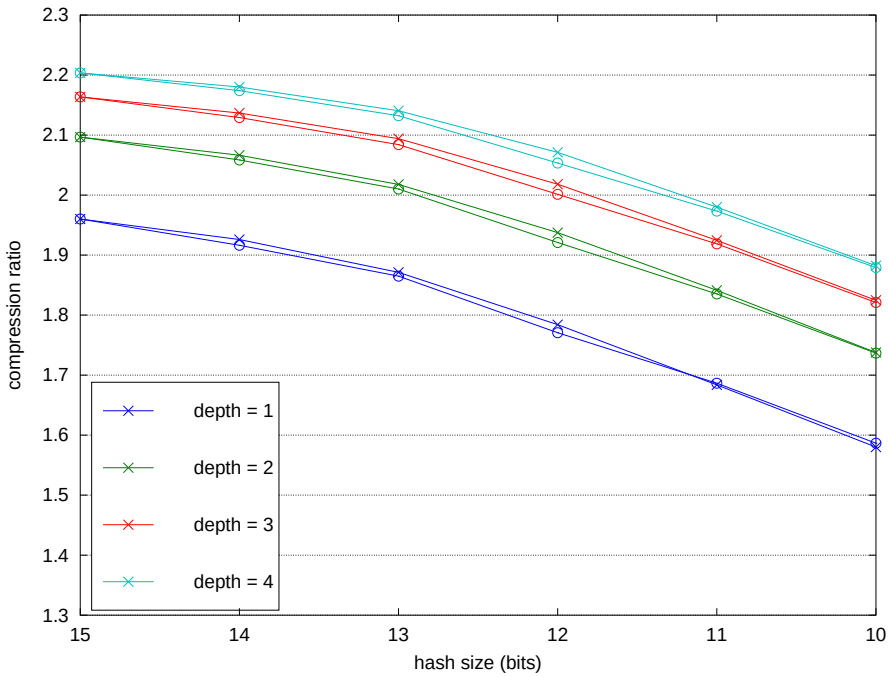


Figure 4.4: Comparison of the compression ratios obtained by using the least (crosses) and the most (circles) significant bits from Gzip's hash when reducing its size. Maximum match length is 32 B and window size is 32 kB.

One last thing to note from the figures is that there are some combinations of hash size and depth that result in roughly the same compression ratio. Some of these combinations have this property for all the tested corpora. Some of those points can be seen in Table 4.2. These points may offer useful trade-offs for implementations.

Table 4.2: Some combinations of hash size and depth that results in roughly (less than 1 % difference) the same compression ratios, ordered by decreasing compression ratio

Point 1		Point 2	
H. size	Depth	H. size	Depth
14	3	13	4
15	2	13	3
14	2	12	4
13	2	12	3

4.5 Memory Requirements

Some of the parameters discussed in the previous sections greatly affect the amount of memory needed to implement the hardware accelerator. These parameters are the maximum match length,

the hash table depth and the hash size. This section describes the hardware impact of these parameters in terms of the amount of block RAMS (BRAMs) needed.

The two main memories needed to implement the accelerator are for the dictionary and the hash table. The target device for the implementation is from the Zynq-7000 family. These devices have two BRAM primitives: RAMB36E1 and RAMB18E1. A RAMB36E1 stores 36 kilobits and can be configured as two RAMB18E1, each storing 18 kb. These primitives can be configured to used one from several aspect ratios, i.e. the data width and the number of positions of the memory can be changed. Each primitive has two data buses and two address buses. For each address bus there is a write enable (WE) pin. This means that for a single primitive at most two distinct positions can be read or written per clock cycle.

The size of the dictionary memory depends on the maximum match length and hash table depth. This memory must be able to perform byte-aligned accesses of a number of bytes equal to the maximum match length. Furthermore, it must support a number of reads per cycle that is at least equal to the hash table depth. To meet these requirements — along with the required dictionary size — a custom memory must be designed using multiple BRAMs. Figure 4.5a shows the number of RAMB36E1 primitives needed to meet the read requirements, for a dictionary of 32 kB. As the depth increases data replication is used to cope with the increase in the number of accesses. That graph does not take into account that the dictionary must be written to. However, the memory should support writing at least one byte per cycle. As mentioned above only two address buses exist per primitive. Writing requires increasing the number of address lines, because in most cycles the addresses that are read do not overlap the ones that must be written (if they did the same address buses could be used). This results in essentially doubling the number of primitives used to implement the dictionary, as seen in Figure 4.5b. The fact that the points for maximum match length of 8 and 16 overlap is related with the extra flexibility provided by using RAMB18E1 primitives. Note that the points in the graph for match lengths of 16 and 32 are valid for a dictionary of 32 kB but also for smaller dictionaries. The reason is that the number of needed primitives must be at least equal to the maximum match length: as more primitives are used, the total size of each primitive should be reduced in order to keep the total size of the memory constant, but the smallest primitive available is RAMB18E1. The result is that smaller dictionaries use the same amount of primitives, but these are only partially used. An alternative solution to cope with writes while not increasing the number of primitives might be possible by operating them with a clock frequency that is the double of the one for the surrounding circuitry. This way one cycle could be used to perform two reads and another to do a read and a write.

The size of the memory for the hash table depends on its depth and on the size of the hash values. Each slot of the hash table has to store an address pointing to a position in the dictionary, i.e. a n-bit address for the dictionary memory. n needs to be greater or equal to 10 for dictionaries with at least 1 kB, resulting in slots of 2 bytes. The most direct implementation for the hash table consists in reserving one slot for each possible hash value and per unit of depth. With this simple memory organisation the total number of slots is $2^b d$, if b is the hash size and d the depth. The hash memory must be able to perform d reads of 2 bytes per cycle and at least one write of 2 bytes, which leads to

the memory requirements shown in Figure 4.6.

4.6 Conclusion

The information in the previous sections allows to make an informed selection of the several parameters to use for implementation. In order to identify which set of parameters is most advantageous we must set constraints of the compression ratio. Hence, we choose to restrict the minimum compression ratio of the accelerator to 2. Because the compression ratio varies depending on the input data we must also choose for which input this minimum compression ratio should be validated. With this in mind, our objective will be to choose parameters such that the English Wikipedia corpus (`enwik`) — the worst compressing test corpora — compresses with a ratio of at least 2.0.

From the simulation results it is evident that maximum match lengths of 4 and 8 do not allow to attain the desired compression ratio levels. The same is true for windows of 1, 2 and 4 kB. Hashes with less than 12 bits do not compress enough using Gzip's selection method, while using Method 4 requires a hash size of at least 13 bits. A hash table depth of 1 cannot be used either. These simple observations exclude a set of the possible parameter combinations.

As mentioned in Section 4.5, reducing the size of the dictionary from 32 kB does not reduce the number of BRAMs needed to implement it, except for a maximum match length of 8 bytes. Since that maximum match length has already been excluded, we conclude that the window size should be 32 kB.

The preferred selection method should be the lazy selection method used by Gzip, but Method 4 (maximum match length and minimum distance) is also acceptable if a simpler implementation is desired at the expense of some compression. The difference in resource utilisation between the two methods should not be significant when compared to the total resources used by the hardware core.

Table 4.3 shows the combinations of depth, hash size and maximum match length that result in compression ratios above 2, using selection Methods 4 or 5. For each combination it also shows the total number of BRAMs necessary to implement the hash table and dictionary. The table reveals that a minimum of 48 primitives are necessary, followed by combinations that need 60 or 64. While it could be tempting to chose one of the options that need 48 BRAMs, notice that both use a maximum match length of 16 bytes. From Section 4.2 we know that this parameter is the one that limits the compression ratio for highly redundant inputs. Using static-Huffman Deflate the maximum compression ratio for a maximum length of 16 bytes is 9.8, while for 32 bytes it is 18.3 — an increase of 86 %. The option that uses 60 BRAMs uses a match length of 32. Therefore, the improvement of 86 % can be accomplished by an increase in area of only 25 %. For this reason we adopt a maximum match length of 32.

From the remaining combinations, one uses 60 BRAMs and the other uses 64. The depth and hash size combination for these options results in nearly the same compression ratio, as seen in Table 4.2. The combination that requires 60 BRAMs uses a hash size of 13 bits and depth of 3. Using Method 5 this results in a compression ratio of 2.09 for `enwik6`. The depth of 3 can be an advantage in the event that the implementation can benefit of doubling the operation frequency of the dictionary

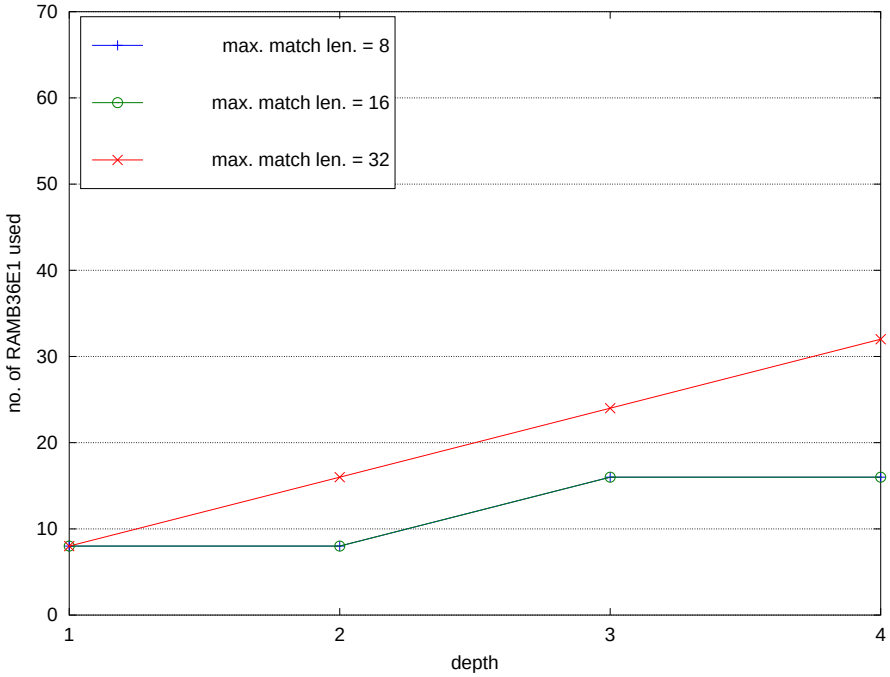
memory, as it would fully utilise the three read accesses. In this event the total number of BRAMs would be reduced to 28, i.e. less than half. Furthermore, if a slight compression increase is required this combination allows to reach the next best combination by increasing only the hash table size, resulting in a BRAM usage of 72; or 40, if double frequency if used for the dictionary memory.

In short, the following parameters can be used to reach a compression ratio greater than 2 for all the test corpora:

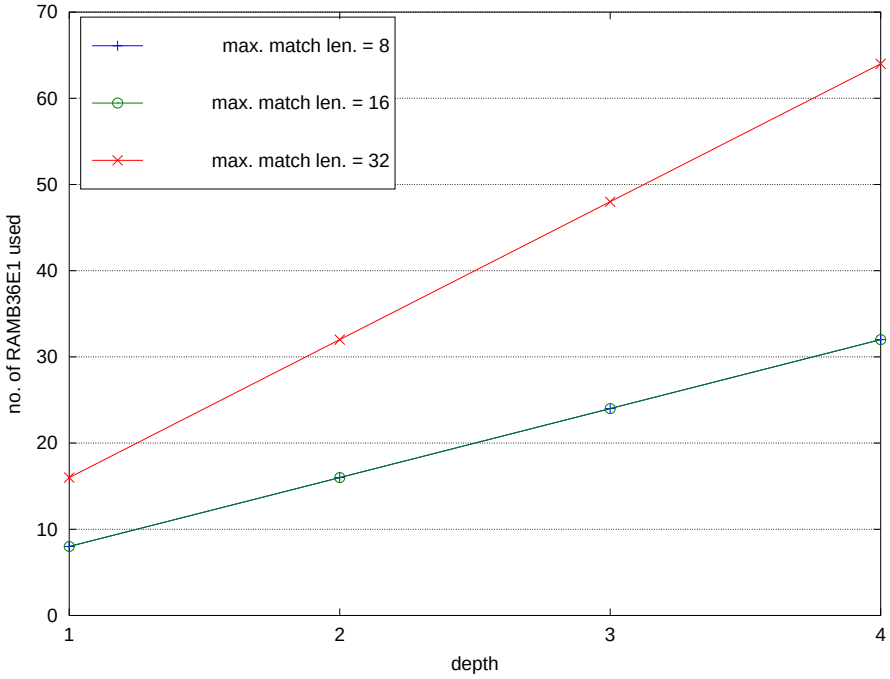
- Maximum match length: 32 bytes
- Window size: 32 kB
- Hash table depth: 3 slots
- Selection method: Method 5 (preferred) or Method 4
- Hash size: 13 bits.

Table 4.3: Total number of RAMB36E1 primitives needed to implement the hash table and dictionary memories, for parameter combinations that result in a compression ratio above 2 for both Methods 4 and 5. The compression ratio values are for Method 5.

H. size	Depth	Max. len.	BRAMs	Comp. ratio
13	3	32	60	2.09
13	4	16	48	2.08
13	4	32	80	2.14
14	3	16	48	2.08
14	3	32	72	2.13
14	4	16	64	2.11
14	4	32	96	2.18
15	2	32	64	2.09
15	3	16	72	2.10
15	3	32	96	2.16
15	4	16	96	2.14
15	4	32	128	2.20



(a)



(b)

Figure 4.5: Number of RAMB36E1 primitives necessary to implement a 32 kB dictionary in function of maximum match length and hash table depth: (a) taking only reads into account; (b) accounting for both reads and writes

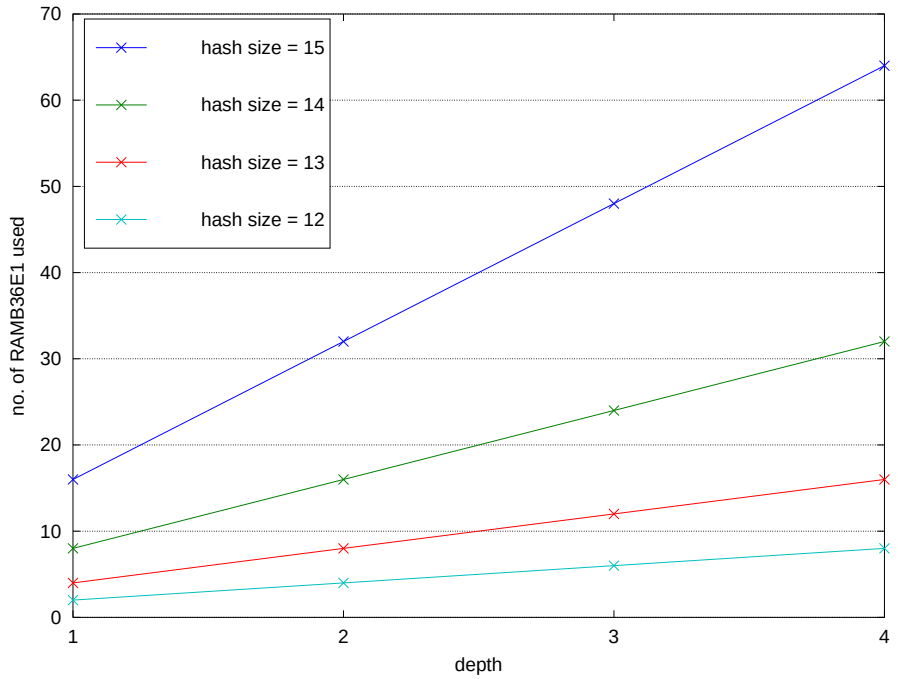


Figure 4.6: Number of RAMB36E1 primitives necessary to implement a hash table directly addressed by the hash value, as a function of depth and hash size

5

LZ77 IP Architecture

Contents

In this chapter we propose an architecture for a fast hashful LZ77 IP. We start by confirming that LZ77 is indeed the slowest part of the Deflate algorithm Then, we describe the proposed LZ77 IP architecture. Finally, we show how LZ77 can be effectively integrated into a hardware/software system to implement Gzip.

5.1 Hardware/Software Partition

Gzip was profiled in order to verify that LZ77 is the slowest part of the Deflate. The results were obtained using the `gprof` profiler with Gzip running with the standard compression level (level 6) running on a Core i5 processor and for the Enwik9 corpus — which should provide more accurate results than a smaller corpus. The relative time of execution of each function and the number of times it was called are shown in Table 5.1.

The results show that 67 % of the execution time is spent finding the longest matches. The `longest_match()` function combined with `fill_window()` constitute Gzip's implementation of the LZ77 algorithm. As expected, LZ77 is the most computationally intensive part of Deflate and the use of a LZ77 hardware accelerator is a good solution. The remaining functions implement dynamic Huffman coding, lazy match selection and computing a checksum of the input.

Table 5.1 also indicates the partition that will be used when integrating the LZ77 IP proposed in the next section into an implementation of Gzip as a hardware/software system.

Table 5.1: Profiling results and hardware/software partition

Time (%)	No. of calls	Function name	Partition
66.87	250046472	`longest_match`	Hardware
13.66	1	`deflate`	Software
5.78	30518	`updcrc`	Software
3.99	30517	`fill_window`	Hardware
3.75	6733	`compress_block`	Software
2.58	423137465	`send_bits`	Software
2.56	189383267	`ct_tally`	Software
0.29	1	`bi_init`	Software

5.2 LZ77 Hardware IP

Two architectures were designed for the LZ77 IP. The proposed hashful architecture is described in Section 5.2.2. A hashless architecture was first designed but it was abandoned due to its poor performance (Section 5.2.1).

5.2.1 Hashless Architecture

This section briefly describes the initially designed architecture, which was abandoned. This architecture aimed to be a simple "brute-force" implementation of the LZ77 algorithm in hardware.

The algorithm this architecture implements is simple: for every position in the window the sequence in that position is compared to the current lookahead; the comparison terminates when

two different bytes are found and the match length is calculated. The search of the window starts from the position immediately before the lookahead such that the matches with the shortest distances are found first.

Figure 5.1 shows how the output of the dictionary and lookahead are compared in order to detect matches. One byte from the dictionary is compared to the lookahead at a time. In the case the match overlaps into the lookahead both read ports of the lookahead memory are used to read different bytes and they are compared to each other. The `match` signal is used to control a counter which counts the length of the matches.

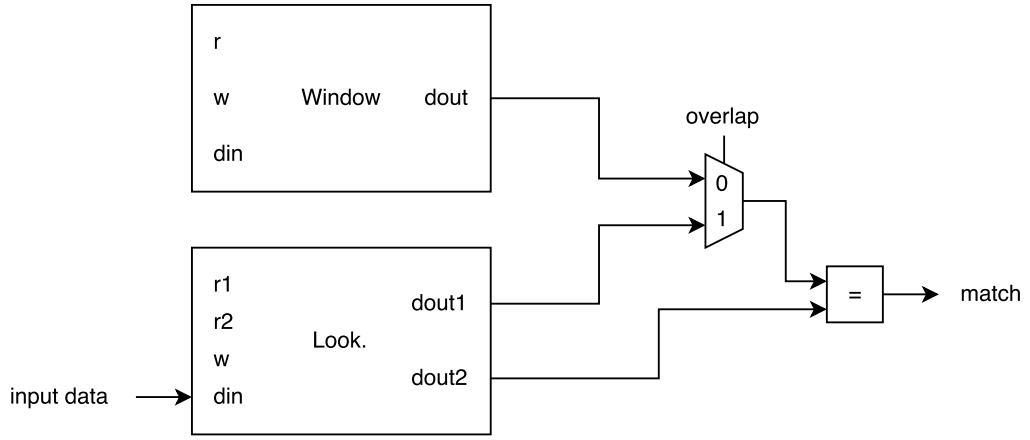


Figure 5.1: Comparison of the dictionary and lookahead

A set of registers keeps track of the current limits of the window and lookahead memories and which sequence is currently being compared to the lookahead. The information in these registers allows to control the addressing of the memories.

This architecture has two advantages: it can find *every* match of the lookahead in the dictionary, i.e. the search for matches is exhaustive; and the resource utilisation is extremely low and, furthermore, it does not increase with the size of the dictionary and the lookahead. The fact that every match is found and that a large dictionary and lookahead can be used means that the best possible compression ratio can be achieved. The hardware utilisation is only 185 LUTs and — for a 32 kB dictionary and a 256-byte long lookahead — 9 BRAMs.

However, these advantages come with a major drawback: because of the exhaustive nature of the algorithm the architecture is extremely slow. In the worst case processing a single lookahead requires a number of cycles proportional to the size of the window. The worst and average cases can be adjusted by using some condition to abort searches early (at the risk that some matches are not found). For a 32 kB dictionary and an operating frequency of 70 MHz it can only achieve a throughput of around 2 kB/s.

This architecture is clearly not appropriate to achieve high throughputs and therefore it was abandoned. However, it gave one important insight: in order for a LZ77 architecture to achieve high throughputs it should know a priori where in the dictionary the matches can be found, so that no time is spent searching for them. This conclusion led to the hashful design described in the next section.

5.2.2 Hashful Architecture

This section describes the architecture of the proposed LZ77 stream-based IP. Figure 5.2 shows the topmost level view of the architecture. The core has four blocks: the lookahead buffer, a lookahead position register (`strstart`), a hasher and a LZ77 unit. The LZ77 unit is the most important block and contains almost all the complexity of the system. By replicating this unit the overall depth of the system can be increased.

The lookahead consists simply of a chain of 1-byte registers. The `strstart` block is a counter register which maintains the position in the window memory corresponding to the beginning of the lookahead. It allows to calculate the match distance. The hasher is a very simple block implementing the Gzip's hashing function from Equation 3.1.

In Chapter 4 several important parameters for the compression ratio were studied, using a compression ratio simulator developed for that purpose. The size of the dictionary memory, the maximum length of the matches and the number of bits of the hash values were explored. These parameters were chosen such that a LZ77 IP with depth 3 combined with the static Huffman codes from Deflate achieves a compression ratio greater than 2 for all the test corpora. Thus, the dictionary is 32 kB long, the maximum match length is 32 and the 13 least significant bits of the hash value given by Equation 3.1 are used as the hash function.

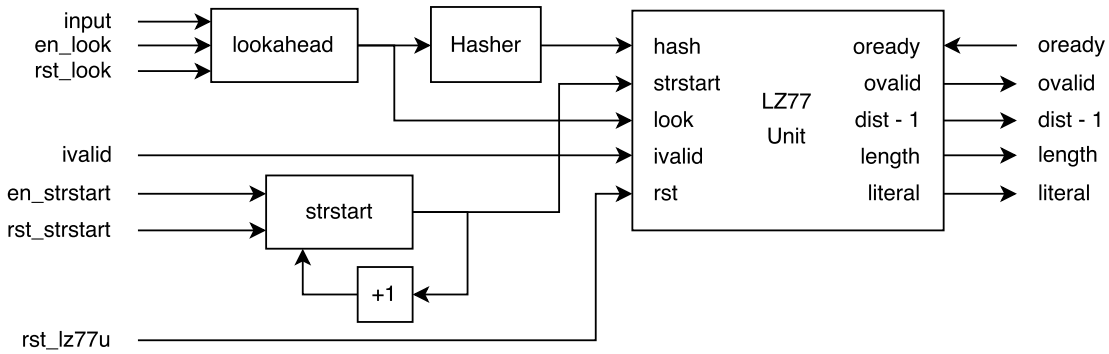


Figure 5.2: Topmost level view of the LZ77 IP datapath

5.2.2.A LZ77 Unit

The LZ77 unit receives the current lookahead, its hash and the current lookahead position and outputs a distance-length pair for matches, when a match is found, or a literal byte, when no match was found. The block diagram for the proposed LZ77 unit is shown in Figure 5.3.

The functionality of the block is as follows. The hash value coming from the hasher is used to index the hash table memory, which outputs a position in the window where a match for the current lookahead is believed to exist. This *candidate position* is then used to index the custom window memory, which outputs 32 bytes. These 32 bytes constitute the *candidate match* that will be compared to the lookahead in order to determine its length. Because in LZ77 the lookahead can match a portion of itself (see below), the output of the dictionary memory is fed into an overlapper block, which outputs a sequence of bytes which may consist only of bytes from the window or a combination of bytes from

the window and the lookahead. Lastly, the overlapper output is compared to the lookahead in order to determine the length of the match, which can range from 0 (there is no match) to 32 (the maximum match length).

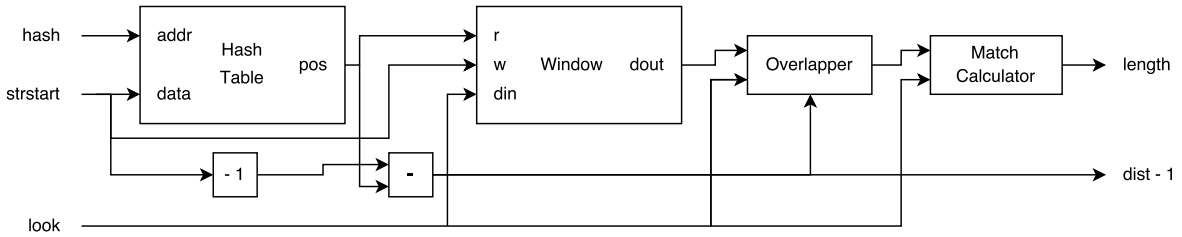


Figure 5.3: Block diagram for the LZ77 unit

In parallel with the previously described process, the match distance is determined. First, one unit is subtracted from the `strstart` value. Then, the candidate position coming from the hash table is subtracted from `strstart-1` and the match distance of the match relative to the current lookahead position is obtained.

5.2.2.B Window Memory

In order to achieve the highest possible throughput the architecture has to process at least one lookahead per cycle. To accomplish this, it is important to be able to make byte-aligned accesses to the window. Furthermore, in order to use a maximum match length of 32 the window must be able to output at least 32 bytes per cycle.

To satisfy these requirements a custom memory was designed for the window using BRAM primitives as building blocks. Taking into account the aspect ratios and the maximum number of accesses per cycle which the BRAMs in the target device support, the solution was to use 32 BRAMs in parallel, as seen in Figure 5.4.

The dictionary is stored sequentially in the BRAMs such that consecutive bytes of the memory are stored in different BRAMs. The address of the dictionary memory is split in two: the lowest 5 bits identify the alignment, while the upper bits are used to address the BRAMs. When an access aligned to 32 bytes is made, all the BRAMs are addressed with the same address. Otherwise, a subset of the BRAMs is addressed directly with the upper part of the original address and another subset is addressed with that addressed incremented by one unit.

The 32 bytes read from the BRAMs will be in an order not suitable to be used during the comparison with the lookahead. Hence, the output of the BRAMs is passed through a rotator circuit. There, 32 bytes are rotated such that the most significant byte of the output is the most distant byte of the window; and the least significant byte is the least distant. The output is now ready to be used by the subsequent blocks.

Regarding the writing of new bytes to the dictionary, the alignment bits are used to generate a write enable signal only for the BRAM that has the oldest byte of the window.

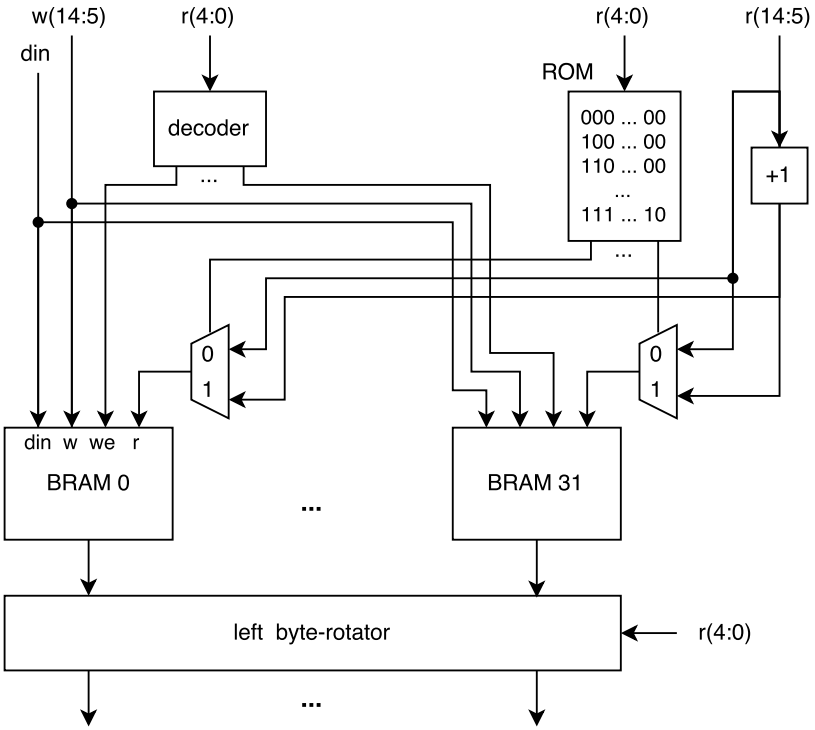


Figure 5.4: Block diagram for the window memory

5.2.2.C Hash Table Memory

The hash table memory is simple, as a direct mapped organisation for the hash table is used. Since 13-bit hash values are used and the window has 32k positions, the hash memory has 2^{13} positions with a word width of at least 15 bits.

This memory can be implemented by using four RAMB36E1 primitives. The four primitives are stacked such that they form a continuous address range, as seen in Figure 5.5. Each memory is addressed with the 11 lowest bits of the hash value, while the two most significant bits are used to select the output of the appropriate BRAM. The least significant bits are also used to generate the write enable for the suitable memory.

5.2.2.D Overlapper

For distances smaller or equal to the size of the lookahead, the candidate match extends into the lookahead. Therefore, for these cases the candidate match will be a combination of bytes from the window and the lookahead.

Figure 5.6 shows that the overlapper consists of a 31-byte shifter and multiplexers. The match distance (calculated in the previous stages of the architecture) is used to select how many bytes should be shifted. The result of the shifter is fed into the multiplexers, whose select signals are also determined according to the match distance.

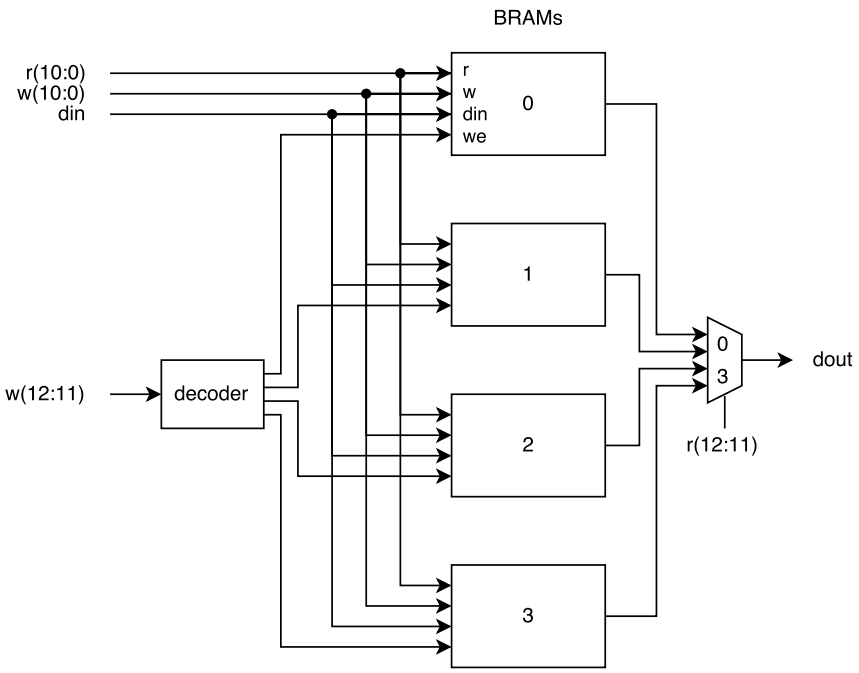


Figure 5.5: Block diagram for the hash table memory

5.2.2.E Match Calculator

This block compares the 32 bytes of the lookahead with those in the corresponding positions coming out of the overlapper. The result of these comparisons is an array of 32 bits, each of which is set to 1 if the corresponding bytes are equal. The number of consecutive set bits, starting from the most significant bit, are then counted in order to obtain the match length.

5.2.2.F AXI Interface

The LZ77 IP was packaged into an AXI-Stream interface such that it can be easily integrated into a complete compressor design. The data bus is 32-bit wide, with little endian byte order. The diagram of the IP after packaging is shown in Figure 5.7. The protocol uses input and output "valid" signals to indicate that the data on the buses is valid. The interface also has "ready" signals to specify if the bus is ready to read new data. When no valid data is found at the input of the IP bubbles will form on the pipeline; and when the output bus is not ready the IP will stall. A "last" signal also exists both on the slave and the master interfaces. This signal allows to delimit data bursts by marking a word as the last of the current transmission.

Regarding the "last" signal, the proposed IP does not need to know when a burst ends. The valid and ready signals suffice to control the reads and writes of the data buses. However, for full AXI-Stream compatibility, the IP was designed such that the `tlast` signal at its output is generated once all the input data from the last transfer has been processed. This way, the transfer size at the input interface defines the size of the output.

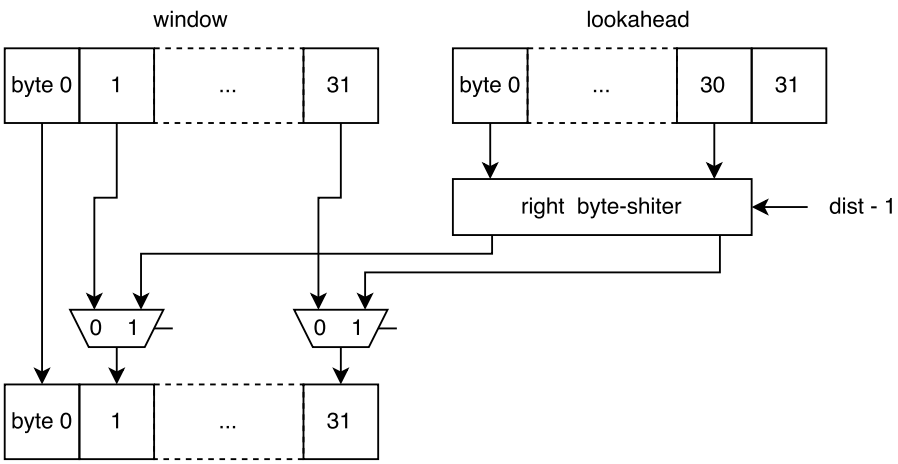


Figure 5.6: Block diagram for the overlapper circuit. The control signals for the multiplexers are generated based on the match distance.

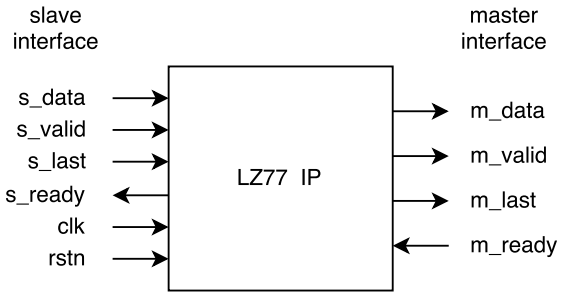


Figure 5.7: AXI-Stream interface of the LZ77 IP

5.2.2.G Datapath

In order to improve the throughput of the architecture its datapath was pipelined. The pipeline has seven stages, as shown in Figure 5.8. The registers from the lookahead and `strstart` are the first in the pipeline. The hash table uses two pipeline stages, which are implemented using the built-in registers of the BRAMs. For the window three stages are used: two of them are implemented with the registers from the BRAMs, while the last pipeline registers are placed immediately after the window's byte-rotator. The last pipeline registers are placed between the overlapper and the match calculator.

A validity bit accompanies each stage of the pipeline, indicating whether the data currently in the stage is valid. When no valid data exists at the input of the datapath "bubbles" with invalid data will form. On the other hand, when the bus at the output of the datapath is not ready the pipeline must be stalled. Therefore, the validity bits and the "output ready" signals allow to control the registers of the pipeline stages.

The hash table and window memories can only be written to when the data at their inputs is valid. In order for the writes to be successful both the write enable and the enable signals must be set. On the other hand, if the output of the memories is valid and a stall occurs, the enable signal of the BRAMs must be set to 0, so that no new value is read.

Before the pipeline registers were added, the datapath had three stages, due to the latency introduced by the BRAMs, lookahead and `strstart`, and its implementation frequency was 70 MHz.

After including the described pipeline registers the maximum frequency rose to 130 MHz.

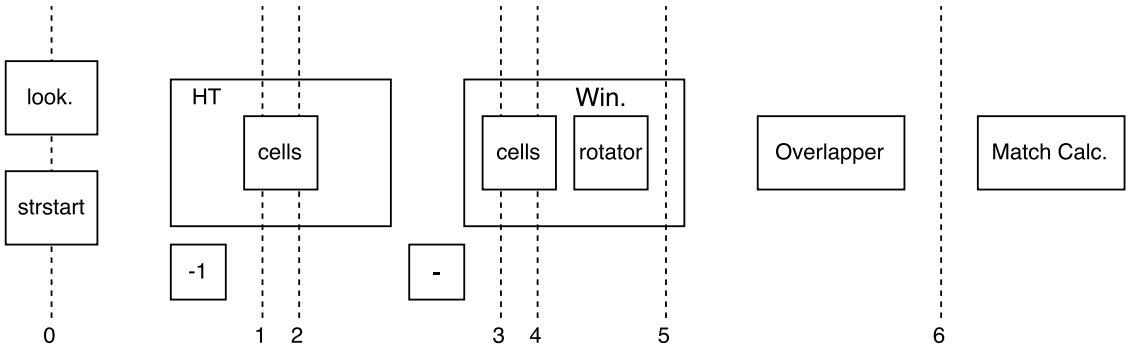


Figure 5.8: Representation of the locations of the pipeline registers in the datapath

5.2.2.H Control Unit

The control unit uses a finite-state machine to define the datapath control signals and control the AXI-Stream interface. Figure 5.9 shows a flowchart with the three states of the machine and the transition conditions. In the `initial` state the IP is simply waiting for data to process.

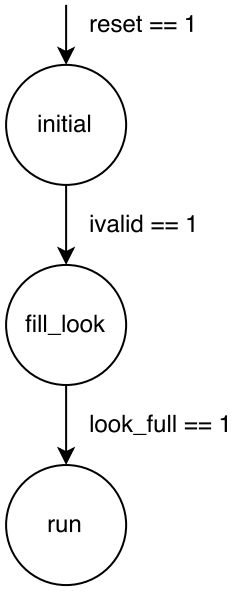


Figure 5.9: Finite-state machine for the proposed architecture

When valid data is found at the input of the IP it transitions to the `fill_look` state. As the name implies, this is an initialisation state during which the lookahead is filled with valid data. In this state the lookahead enable signal is set when the input is valid. A counter keeps track of how many bytes of the lookahead have already been filled. During this state the datapath is not yet ready to start the execution of LZ77. Consequently, the LZ77 IP has nothing to output during this state. The state machine leaves the `fill_look` state when the counter indicates that the lookahead is full of valid data.

The state in which the processing actually occurs is the `run` state. In this state, when unprocessed data still exists at the input of the datapath the lookahead and the `strstart` counter are enabled.

Furthermore, the `tlast` signal for the output is generated when all the data received up to a `tlast` signal at the input has been processed (the reasoning behind this behaviour was explained in Section 5.2.2.F).

In the `run` state the actual processing of the data takes place and the outputs of the LZ77 algorithm are produced. Since the IP is not aware of which transfer is the final one, no "end" state exists, nor does the machine return to the `initial` state automatically. Instead, it only returns to the `initial` state when a reset signal is sent to the IP.

The IP receives four bytes from the AXI data bus in a single cycle, but it only processes one byte per cycle. Therefore, the four bytes are buffered by the IP and during the `fill_lookahead` and `run` states the state machine keeps track of the next unprocessed byte and feeds that byte into the datapath as appropriate.

5.3 Gzip Hardware/Software System

For evaluation purposes, a fully functional Gzip hardware/software system was prototyped in a Zybo development board. The main component of this board is a Zynq Z-7010 system on chip, which integrates an ARM Cortex-A9 processing system and programmable logic. The board also contains multiple peripherals, one of which is a 512 MB DDR3 memory.

The LZ77 IP was implemented in the programmable logic of the Zynq device, while the software components were executed in the ARM processor. The DDR3 RAM was divided into four regions: a region for the input file, one for the compressed file and two regions for the results of the LZ77 IP. An AXI DMA is used to directly read/write data from/to the DDR3 RAM to/from the LZ77 IP. The DMA transfers are configured by the software executing in the ARM, which initiates the transfers and verifies their completion.

A diagram for the system is shown in Figure 5.10, while Figure 5.11 is a simple flowchart of the code executing on the ARM, with emphasis on the DMA requests. The ARM starts by initialising the LZ77 IP accelerator, by requesting the DMA to transfer 32 bytes to fill the lookahead. Then — still part of the system's initialisation — one request is sent to the DMA to start transferring the first chunk of the file into the IP; and another request instructs it to write the output of the IP into the first region of the RAM reserved for IP results. The ARM then waits for the IP to finish before proceeding to the main processing loop.

The core functionality of the main loop is to produce the final compressed output for a chunk of the file, using the results from the IP. The results for a match are read from the RAM and the ARM verifies whether it should encode a match or a literal. The encoding is done using the static Huffman coding described in the Deflate specification. In order to improve compression in about 5 % Gzip's lazy match evaluation was used, which only encodes a match after it has been verified that no better match can be found for the next position of the input.

The remaining part of the loop controls the DMA transfers for chunks of the file. In order for the ARM not to have to wait for the processing of the IP, the chunks of the file that the ARM and the IP

process in each moment are offset by one unit. For example, when the IP is processing chunk n, the ARM is processing chunk $n+1$. This is the reason why two memory regions for the results of the IP are used: one region is read by the ARM while the other is being written by the DMA with new results from the IP.

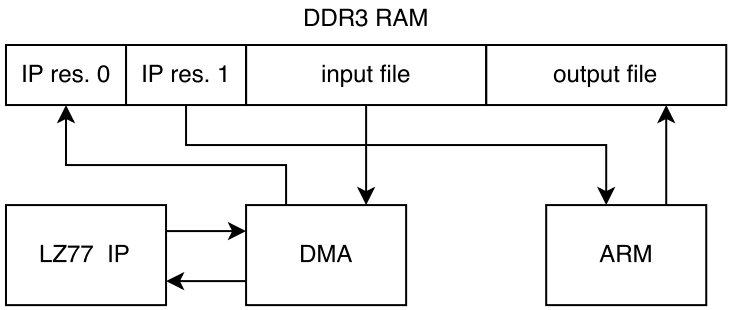


Figure 5.10: Diagram for the hardware/software system

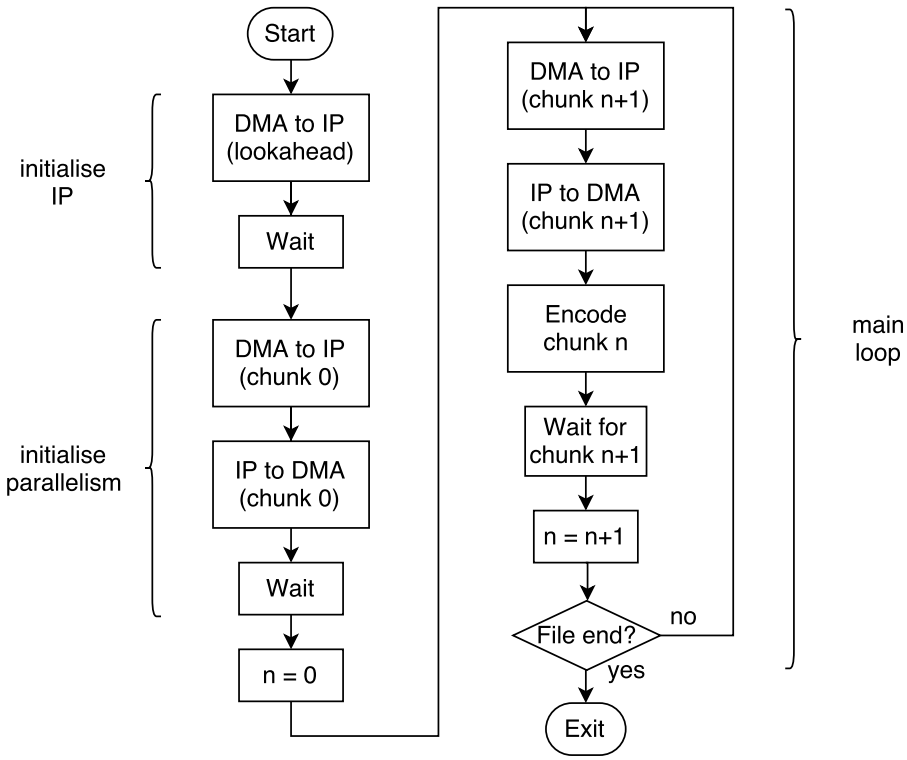


Figure 5.11: Flowchart for the software executing on the ARM, with emphasis on the DMA transfer requests

5.4 Conclusion

This chapter proposed a hardware architecture for a LZ77 IP. It also showed how this IP can be integrated into a hardware/software prototype implementing Gzip. Performance results are shown in the next chapter.

6

Results

Contents

This chapter presents performance and area results for the designed hardware LZ77 IP and for a hardware/software prototype implementing Gzip on the Zybo development board. The prototype uses depth 1 and static Huffman codes.

Throughout the chapter performance is compared to that of software-only Gzip running on the Zynq's ARM and running on a PC. The results for the ARM were obtained by adapting Gzip's source so that it could run on a Zynq's ARM Cortex-A9 core at 650 MHz with a predefined compression level. The source was compiled using the GNU Compiler ARM toolchain to produce a binary that could run on "bare metal." The optimisation level 3 (`-O3`) was used.

The results for the computer were measured using a computer with an Intel Core i5-2410M. This processor runs at 2.3 GHz but uses the Intel Turbo Boost technology that allows it to scale up to 2.9 GHz. Due to dynamic frequency scaling — aggravated by the Turbo feature, which only turns on when the CPU is under high loads — care was needed in order to collect consistent results. Small corpora were verified to be left at a disadvantage because the total execution time was not enough to make the frequency rise. Therefore, for each test, Gzip was run multiple times (from 8 to 128 depending on the size of the corpus) and the results average execution time rejecting the time for the first execution, which was the one that generally deviated considerably.

The computer's disk was benchmarked and its average read rate is 70 MB/s, which from the results can be seen not to limit Gzip. Its write rate is not important because in the tests Gzip's output was written to the `/dev/null` device, i.e. it was discarded. On the embedded system the file was read from the Zybo's DDR3 RAM which does not limit Gzip's performance either.

6.1 LZ77 IP

6.1.1 Performance

The LZ77 accelerator was implemented with a frequency of 130 MHz on a Zynq Z-7010 device. It can receive an uninterrupted stream of bytes and consume one byte in each cycle. Consequently, its maximum throughput is 123.9 MiB/s.

Feeding the IP through the AXI DMA might limit the throughput of the IP. In order to measure this difference, a simple test was designed which consisted of transferring chunks of 16 kB to the IP through the DMA (see Listing 6.1). The 16 kB correspond to the default maximum transfer size the DMA IP supports in a single transfer. The transfers were controlled by the Zynq's ARM and a total of 200 MB were transferred during the test.

The obtained throughput was 108.7 MiB/s, which is 88 % of the maximum value. This difference should be due to the relatively small size of each transfer compared to the total transferred size. Using the provided AXI DMA software library, each new transfer request involves multiple accesses to the DMA's AXI-Lite interface in order to verify that the DMA is free and set it up. On the other hand, it was verified — using the Integrated Logic Analyzer (ILA) — that the AXI-Stream transfers are not completely continuous. In particular it seems that the first few words in each transfer are transmitted isolated from the remaining words. According to the AXI DMA IP manual, the IP benchmarks indicate

```
transfer first 32 bytes to IP (fill the lookahead)
nr_chunks = size_of_file / CHUNK_SIZE

while nr_chunks >= 0
    status = DMA_to_IP_request()
    if status == failure
        error
    end if

    status = IP_to_DMA_request()
    if status == failure
        error
    end if

    wait on IP to DMA transfer
    nr_chunks = nr_chunks - 1
end while
```

Listing 6.1: Pseudocode to test the maximum throughput of the LZ77 accelerator, fed through the AXI DMA IP

that for a frequency of 100 MHz it supports a throughput of about 400 MB/s for the MM2S interface and 300 MB/s for the S2MM interface. However, the fact that the DMA IP has faster throughputs than the LZ77 IP does not mean that it does not limit the overall throughput: for the DMA not to limit the throughput its operating frequency would have to be considerably higher than the frequency of the accelerator. In conclusion, in order to improve the loss of throughput through the DMA, larger transfer sizes should be used and the operating frequency of the DMA could be increased.

6.1.2 Hardware Utilisation

The hardware utilisation results for the LZ77 accelerator after implementation are shown in Table 6.1. The percentages on the table are relative to the total number of each resource available on the Zynq Z-7010 device. The relative utilisation reveals that the limiting factor for the accelerator's scalability is the amount of available block RAMs. One third of the available block RAMs are used by the window and hash table memories while only 12 % of the LUTs are used to implement the needed logic. This conclusion is still valid for other devices because the current BRAM to LUT ratios seem to vary from 1.5 to around 4 BRAMs per 1k LUTs, while the ratio for the accelerator is 9.5. The components using more LUTs are the rotator inside the custom dictionary memory and a shifter in the overlapper block.

Increasing the depth of the accelerator results in a linear increase of the resource utilisation. However, when increasing the depth the relative usage of BRAMs can be reduced by duplicating the frequency of the window memory: for a depth of 3 only 28 BRAMs would be needed, against 60 without the frequency increase, resulting in a BRAM-LUT ratio of 4.4.

Table 6.1: Global and partial hardware utilisation for the LZ77 IP with depth 1. Relative utilisations are for a Zynq Z-7010.

		Utilisation	
Component	Resource	Absolute	Relative (%)
LZ77 IP	Slice LUTs	2171	12.3
	LUT as logic	2042	11.6
	LUT as memory	129	2.2
	Slice registers	1536	4.4
	Slices	671	15.3
	RAMB36E1	20	33.3
LZ77 Unit	Slice LUTs	2122	12.0
	LUT as logic	1994	11.3
	LUT as memory	128	2.1
	Slice registers	1189	3.3
	Slices	633	14.3
	RAMB36E1	20	33.3
Hash Table	Slice LUTs	350	1.9
	LUT as logic	350	1.9
	LUT as memory	0	0.0
	Slice registers	4	0.0
	Slices	135	3.0
	RAMB36E1	4	6.6
Dictionary	Slice LUTs	949	5.3
	LUT as logic	949	5.3
	LUT as memory	0	0.0
	Slice registers	269	0.7
	Slices	367	8.3
	RAMB36E1	16	26.6
Overlapper	Slice LUTs	490	2.7
	LUT as logic	490	2.7
	LUT as memory	0	0.0
	Slice registers	0	0.0
	Slices	211	4.8
	RAMB36E1	0	0.0

6.2 Gzip Hardware/Software System

The implemented Gzip prototype is a hardware/software system that uses the LZ77 IP to perform the matches and the ARM processor to select the matches, encode them using Deflate's static-Huffman code and calculate a checksum for the input. This section outlines the results for the complete prototype.

6.2.1 Performance

The full hardware/software system was implemented in a Zynq Z-7010. Throughput and compression ratio were measured for the test corpora. These results are shown in Table 6.2.

The table shows that throughputs from 8.2 to 10 MB/s were obtained, while the compression ratio varies from 1.8 to 2.4. The compression ratios are in accordance with what would be expected for a

system using a 32 kB dictionary, 32 bytes of lookahead, depth 1 and static Huffman encoding.

However, the obtained throughputs are much lower than the measured throughput of the IP through the DMA — 108 MB/s. This is due to the fact that the ARM cannot produce the final encoding of the stream at the same rhythm as the LZ77 IP produces results. Section 6.3.1 shows that this throughput is expected from the prototype, while Section 6.4 discusses how to best use the IP to maximise performance.

Table 6.2: Throughput and compression ratio for the Gzip hardware/software system

Corpus	Throughput (MB/s)	Comp. ratio
Calgary	8.77	1.98
Canterbury	10.08	2.43
Enwik7	8.19	1.81
Enwik8	8.26	1.83
Silesia	9.28	2.14

6.2.2 Hardware Utilisation

The complete hardware/software system has three main components: the LZ77 IP, the ARM processing system and the AXI DMA IP. In order to connect these components two AXI Interconnects are needed: one to connect the S2MM and MM2S interfaces of the DMA to a high performance port of the processing system; and another to connect a general purpose port to the DMA's AXI-Lite interface. The last needed component is a Processor System Reset Module, which generates a reset signal for the other components that is synchronised with the clock signal.

Table 6.3 shows the individual programmable logic utilisation of each major component of the system, as well as the total utilisation for the entire system, for a depth of 1. For this depth the logic utilisation of the LZ77 accelerator is below that of the remaining components. However, when increasing the depth of the accelerator, the utilisation of the remaining components will remain the same and the relative area of the LZ77 IP will be preponderant. The BRAM usage is still the limiting factor when scaling the architecture. Since the DMA uses 5 BRAMs the maximum accelerator depth that can fit on the Zynq Z-7010 device is 2; or 6, if the window frequency is doubled.

6.3 Hardware/Software vs. Software-only Performance

This section compares the performance — both in terms of throughput and compression ratio — of the Gzip hardware/software prototype to that achieved by a version of Gzip adapted to run on the bare metal ARM Cortex-A9.

6.3.1 Throughput

Table 6.4 shows — for the test corpora — the throughputs of the hardware/software prototype and for the ARM running `gzip -1`, as well as the respective speedups. The results show that the hardware/software system is consistently faster than the software, by a factor of about 1.5. The lowest

Table 6.3: Hardware utilisation of the programmable logic for the hardware/software system, for a hash table depth of 1. Relative utilisation is for a Zynq Z-7010.

		Utilisation	
Component	Resource	Absolute	Relative (%)
AXI DMA IP	Slice LUTs	1362	7.7
	Slice registers	1868	5.3
	Slices	590	13.4
	RAMB36E1	5	8.3
AXI Mem. Intercon.	Slice LUTs	1149	6.5
	Slice registers	1312	3.7
	Slices	412	9.4
	RAMB36E1	0	0.0
LZ77 IP	Slice LUTs	2171	12.3
	Slice registers	1536	4.4
	Slices	671	15.3
	RAMB36E1	20	33.3
Others	Slice LUTs	392	2.2
	Slice registers	518	1.4
	Slices	196	4.5
	RAMB36E1	0	0.0
Complete System	Slice LUTs	5073	28.8
	Slice registers	5234	14.9
	Slices	1753	39.8
	RAMB36E1	25	41.7

obtained speedup is 1.44 and it is for the Canterbury corpus, which is the best compressing test. The slight drop in speedup for this file should be due to the fact that `gzip -1` does not perform lazy matching: it falls back to a simpler match selection, implemented by the `deflate_fast()` function. On the other hand, our prototype always uses lazy matching and therefore is at a disadvantage mainly for the best compressing files, for which more matches are analysed.

Table 6.4: Throughputs for `gzip -1` running on the ARM and for the hardware/software system, and respective speedups

	Throughput (MB/s)		
Corpus	Software (ARM)	HW/SW	Speedup
Calgary	5.91	8.77	1.48
Canterbury	6.99	10.08	1.44
Enwik7	5.45	8.19	1.50
Enwik8	5.49	8.26	1.50
Silesia	6.25	9.28	1.49

In order to understand how the hardware/software system improves the execution time Gzip's execution was profiled. Table 6.5 shows a portion of the results of `gprof` when compressing the Silesia corpus with Gzip's level 1.

From the profiling results it is possible to identify which percentage of the time should be saved by the hardware accelerator. The time spent in the `longest_match()` function — which is the one the program spends the most time in — is almost fully saved by the LZ77 IP since it processes in parallel

with the ARM (the software will still spend some time sending requests to the DMA). The IP also saves most of the time spent in `fill_window()` as most of the tasks of this function are unnecessary when using the IP. The function is called once for roughly every 32 kB of input: it moves the second half of the window to the first; and either zeroes or subtracts the window size from each of the positions in the `head` and `prev` arrays, which amount to 64k memory positions (recall Section 3.2.2). Finally, the prototype has no functionality equivalent to the `ct_tally()` function. This function is responsible for updating the frequencies needed for the dynamic Huffman code. Since the prototype uses static Huffman this is not needed.

Table 6.5: Profiling results when compressing the Silesia corpus with Gzip's fastest level. The name `deflate` actually refers to `deflate_fast()`, which seems to be embedded into `deflate()` by the compiler optimisations.

Time (%)	Function name
28.16	`longest_match`
16.51	`updcrc`
14.81	`deflate`
12.38	`compress_block`
10.44	`fill_window`
8.13	`ct_tally`
7.53	`send_bits`
0.97	`bi_init`
0.49	`pqdownheap`

The fact that the prototype always uses lazy matching while Gzip's fastest mode does not must also be taken into account. By forcing Gzip to use lazy matching it was found that its execution time increased by about 10 %. Therefore, for this example the use of the LZ77 IP is expected to reduce the execution time by about $28 + 10 + 8 - 10 = 36\,\%$. The 1.49 speedup obtained for this corpus implies that the percentage of saved time compared to the software implementation is 33 %.

In conclusion, the measured speedup of the hardware/software prototype versus the software corresponds to the expected speedup for the system.

6.3.2 Compression Ratio

Gzip with level 1 uses at most depth 4 and uses dynamic Huffman codes. Therefore the compression ratio of the prototype is expected to be inferior to that of Gzip. Table 6.6 shows the compression ratio attained by the prototype and `gzip -1` for several corpora and reveals that the prototype's compression ratio is bellow that of the software by 22 to 25 %.

The compression ratio results for the prototype exactly agree with those obtained with the simulator described in Section 4.1. Therefore, we can predict that if the prototype's depth was increased to 2, for example, the compression ratio gap would be reduced to 16.5–20 %, while maintaining the speedup of 1.5. However, the Gzip software will always have an advantage while static Huffman is used by the system. Consequently, dynamic Huffman should be used to aid in increasing compression, as mentioned in Section 6.4.

Table 6.6: Compression ratios for `gzip -1` and for the hardware/software prototype, with depth 1

	Comp. Ratio		
Corpus	Software	HW/SW	Difference (%)
Calgary	2.63	1.97	-24.8
Canterbury	3.23	2.43	-24.7
Enwik7	2.34	1.80	-22.8
Enwik8	2.36	1.82	-22.8
Silesia	2.73	2.13	-21.9

6.4 Maximising Performance

The previous results show that, in order to maximise throughput performance, the LZ77 IP should be integrated into hardware-only implementations. The IP's AXI-Stream interface makes it easy to combine with other AXI IPs. As an example, it could be combined with Xilink's Ethernet Subsystem IP and a static Huffman IP in order to compress an Ethernet stream at a theoretical rate of 1 Gbps (the LZ77 IP can compress at 1.04 Gbps).

Table 6.7 shows the throughputs of Gzip's fastest mode running on a Intel Core i5-2410M processor. It also shows the expected speedups of a hardware-only architecture, considering its maximum throughput (123 MB/s). The throughput of a hardware-only implementation of Gzip employing the proposed IP would be about 20 times faster than Gzip running on the Zynq's ARM, and from 1.7 to 2.4 times faster than on the Core i5.

Table 6.7: Speedup of a Gzip hardware implementation versus Gzip's fastest mode, running on the Zynq's ARM and on a Core i5

	Throughput (MB/s)		Speedup	
Corpus	ARM	Core i5	ARM	Core i5
Calgary	5.91	50.11	20.97	2.47
Canterbury	6.99	62.57	17.73	1.98
Enwik7	5.45	44.98	22.72	2.75
Enwik8	5.49	45.76	22.55	2.71
Silesia	6.25	51.96	19.83	2.38

The scalability of the LZ77 IP can be seen from two perspectives: the depth of a single IP can be increased in order to improve the compression ratio; or multiple accelerators can be implemented in a single device in order to compress multiple streams simultaneously.

In the largest Zynq device, Z-7100, up to 37 LZ77 units can be implemented. Using a depth of 2 combined with static Huffman results in a compression ratio gain of about 7 %, compared with depth 1. However, as the compression ratio increases, the difficulty to further improve is exponential: using a depth of 37 results in 22–28 % better compression than for depth 1, depending on the corpus. The best way to further increase compression is to use dynamic Huffman IP, which would allow to reach gains from 35 to 50 %, instead of increasing the depth even more.

On the other hand, scaling the system by using multiple instances of the LZ77 IP could result in a global throughput of 4.4 GB/s, distributed equally by 37 streams. Depending on the application it

might be useful to instantiate multiple accelerators with differing depths. This would allow to apply different compression ratios to multiple streams.

Table 6.8 shows performance metrics for the proposed architecture compared with the hashful architectures described in Section 2.5. In terms of throughput the proposed architecture is the third fastest, while the two faster architectures are 23 and 46 times faster. On the other hand, the proposed architecture has significantly lower hardware utilisation: it is estimated to use from 50 to 75 times less LUTs than the faster architectures. The reduced area makes it suitable for implementation in smaller FPGA devices and for implementing multiple cores in larger devices.

Table 6.8: Comparison metrics for the proposed architecture and other architectures in the literature

Ref.	Architecture type	Win. size	Look. len.	Thr. (MB/s)	Area	
[68]	Hashful (linked list)	32768	258	?	2077	4-input LUT
[67]	Hashful (linked list)	32768	258	46	2620	6-input LUT
[69]	Hashful (multi-match)	16384	?	62	?	
	Hashful (multi-match)	32768	32	123	2171	6-input LUT
[70]	Hashful (multi-match)	?	16	2908	110000	ALMs
[71]	Hashful (multi-match)	65536	32	5734	108000	ALMs

6.5 Conclusion

This chapter has shown that the proposed LZ77 IP has a maximum throughput of 123.9 MiB/s. By integrating the IP into a hardware/software implementation of Gzip a speedup of 1.5 was obtained versus Gzip running on an ARM processor. It was found that the ARM constitutes a bottleneck for the throughput of the prototype and it was concluded that to maximise the throughout of the IP it should be integrated into hardware-only implementations. This way speedups from 2 to 2.7 could be attained compared to a Core i5 processor. To maximise the compression ratio dynamic Huffman should be combined with increasing the depth of the IP.

7

Conclusion

Data compression has received and continues to receive great attention due to its increasing importance as more digital data is produced every day. In the last half decade alone, companies such as Google, Facebook and Apple created and made freely available six new lossless compression algorithms. Among the objectives of these algorithms are increased storage efficiency, faster web page load times and improving the battery life of wireless devices.

In spite of its advantages, data compression is generally slow and resource-intensive. For these reasons, much research has been made concerning the hardware acceleration of compression algorithms and even proprietary compression coprocessors exist. These same reasons constitute the motivation which led to the proposal of this thesis.

In this work a streaming hardware architecture implementing the LZ77 compression algorithm was proposed. The resulting IP can be used easily in multiple compression applications. The results show that a maximum throughput of 123.9 MiB/s can be achieved by the IP. This allows to use it in fast applications such as compressing Ethernet streams at 1 Gbps.

The functionality of the LZ77 IP was demonstrated by integrating it into a hardware/software system implementing Gzip. The results for this system showed that for this type of application the ARM processor constitutes a bottleneck for the performance of the system and thus, in order to maximise throughput, the proposed IP should be used in hardware-only implementations.

Other contributions of this work are a summary of some of the most important aspects of Deflate and its implementation in Gzip; a compression ratio simulator for the study of the LZ77 and Deflate algorithms; and a study of the impact implementation parameters have on the compression ratio of static-Huffman Deflate.

Future Work

Several aspects of this work can be further improved or explored in the future: a dynamic Huffman IP can be designed and integrated with the proposed LZ77 IP so that a full hardware Gzip accelerator can be realised with the best possible compression ratios; the pipeline of the LZ77 accelerator can be further improved in order to increase its throughput; the architecture can be adapted in order to process multiple lookaheads, for a greater throughput increase; a more efficient memory organisation for the window and hash tables can be explored; the communication with the accelerator can be made through streaming interfaces such as Ethernet, USB or PCI; and lazy match selection techniques can be further explored, namely by applying a shortest path algorithm to select the best combinations of matches for chunks of the input data.

References

[1] B. Tiwari and A. Kumar, “Aggregated Deflate-RLE compression technique for body sensor network,” in *Software Engineering (CONSEG), 2012 CSI Sixth International Conference on*. IEEE, 2012, pp. 1–6.

[2] C. Shannon, “A mathematical theory of communication,” *Bell System Technical Journal*, vol. 27, pp. 379–423, 623–656, July, October 1948.

[3] R. M. Fano, “The transmission of information,” Research Laboratory of Electronics, Massachusetts Institute of Technology, Technical Report 65, March 17, 1949.

[4] D. A. Huffman, “A method for construction of minimum-redundancy codes,” *Proceedings IRE*, vol. 40, pp. 1098–1101, 1952.

[5] S. W. Golomb, “Run-length encodings,” *IEEE Transactions on Information Theory*, vol. 12, pp. 399–401, July 1966.

[6] C. E. Shannon, “Prediction and entropy of printed English,” *Bell System Technical Journal*, vol. 30, pp. 50–64, Jan. 1951.

[7] T. M. Cover and R. C. King, “A convergent gambling estimate of the entropy of English,” *IEEE Transactions on Information Theory*, vol. 24, no. 4, pp. 413–421, July 1978.

[8] W. J. Teahan and J. G. Cleary, “The entropy of English using PPM-based models,” in *Data Compression Conference, 1996. DCC’96. Proceedings*. IEEE, 1996, pp. 53–62.

[9] F. G. Guerrero, “A new look at the classical entropy of written English,” *CoRR*, vol. abs/0911.2284, 2009. [Online]. Available: http://arxiv.org/abs/0911.2284

[10] M. Crochemore and T. Lecroq, “Text data compression algorithms,” in *Algorithms and Theory of Computation Handbook*, M. J. Atallah and M. Blanton, Eds. CRC Press, 2010.

[11] N. Faller, “An adaptive system for data compression,” in *Record of the 7th Asilomar Conference on Circuits, Systems, and Computers*, 1973, pp. 593–597.

[12] R. G. Gallager, “Variations on a theme by Huffman,” *IEEE Trans. Information Theory*, vol. 24, no. 6, pp. 668–674, 1978.

[13] D. E. Knuth, “Dynamic Huffman coding,” *J. Algorithms*, vol. 6, no. 2, pp. 163–180, Jun. 1985.

[14] J. S. Vitter, “Design and analysis of dynamic Huffman codes,” *Journal of the ACM (JACM)*, vol. 34, no. 4, pp. 825–845, Oct. 1987.

[15] R. F. Rice, *Some Practical Universal Noiseless Coding Techniques*, ser. JPL Publication. National Aeronautics and Space Administration, Jet Propulsion Laboratory, California Institute of Technology, Mar. 1979.

[16] J. Rissanen, “Generalized Kraft inequality and arithmetic coding,” *IBM Journal of Research and Development*, vol. 20, no. 3, pp. 198–203, 1976.

[17] R. C. Pasco, “Source coding algorithms for fast data compression.” Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1976.

[18] J. Rissanen and G. G. Langdon, “Arithmetic coding,” *IBM Journal of Research and Development*, vol. 23, no. 2, pp. 149–162, 1979.

[19] I. H. Witten, R. M. Neal, and J. G. Cleary, “Arithmetic coding for data compression,” *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, Jun. 1987.

[20] P. G. Howard and J. S. Vitter, “Practical implementations of arithmetic coding,” in *Image and Text Compression*, J. Storer, Ed. Kluwer Academic Publishers, 1992, pp. 85–112.

[21] A. Moffat, R. M. Neal, and I. H. Witten, “Arithmetic coding revisited,” *ACM Transactions on Information Systems (TOIS)*, vol. 16, no. 3, pp. 256–294, Jul. 1998.

[22] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, Sep. 1977.

[23] ——, “Compression of individual sequences via variable-rate coding,” *IEEE transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, Sep. 1978.

[24] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 2nd ed., ser. Wiley Series in Telecommunications and Signal Processing. Wiley-Interscience, 2006.

[25] A. D. Wyner and J. Ziv, “The sliding-window Lempel–Ziv algorithm is asymptotically optimal,” *Proceedings of the IEEE*, vol. 82, no. 6, pp. 872–877, Jun 1994.

[26] T. A. Welch, “A technique for high-performance data compression,” *Computer*, vol. 17, no. 6, pp. 8–19, Jun. 1984.

[27] L. P. Deutsch, “DEFLATE Compressed Data Format Specification version 1.3,” RFC 1951, May 1996. [Online]. Available: https://tools.ietf.org/html/rfc1951

[28] ——, “GZIP file format specification version 4.3,” RFC 1952, May 1996. [Online]. Available: https://tools.ietf.org/html/rfc1952

[29] S. Hollenbeck, “Transport Layer Security Protocol Compression Methods,” RFC 3749, May 2004. [Online]. Available: https://tools.ietf.org/html/rfc3749

[30] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” RFC 5246, August 2008. [Online]. Available: https://tools.ietf.org/html/rfc5246

[31] E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3,” Working Draft, IETF Secretariat, Internet-Draft draft-ietf-tls-tls13-16, Sep. 2016. [Online]. Available: https://tools.ietf.org/html/draft-ietf-tls-tls13-16

[32] T. Berners-Lee, R. Fielding, and H. Frystyk, “Hypertext Transfer Protocol – HTTP/1.0,” RFC 1945, May 1996. [Online]. Available: https://tools.ietf.org/html/rfc1945

[33] R. Fielding and J. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing,” RFC 7230, June 2014. [Online]. Available: https://tools.ietf.org/html/rfc7230

[34] M. Belshe, R. Peon, and M. Thomson, “Hypertext Transfer Protocol Version 2 (HTTP/2),” RFC 7540, May 2015. [Online]. Available: https://tools.ietf.org/html/rfc7540

[35] R. Peon and H. Ruellan, “HPACK: Header Compression for HTTP/2,” RFC 7541, Dec. 2015. [Online]. Available: https://tools.ietf.org/html/rfc7541

[36] T. Ylonen and C. Lonvick, “The Secure Shell (SSH) Transport Layer Protocol,” RFC 4253, January 2006. [Online]. Available: https://tools.ietf.org/html/rfc4253

[37] L. P. Deutsch and J. Gailly, “ZLIB Compressed Data Format Specification version 3.3,” RFC 1950, May 1996. [Online]. Available: https://tools.ietf.org/html/rfc1950

[38] “Information technology — Computer graphics and image processing — Portable Network Graphics (PNG): Functional specification,” International Organization for Standardization, ISO 15948:2004, 2004.

[39] P. Katz, “String searcher, and compressor using same,” U.S. Patent 5 051 745, Sep. 24, 1991.

[40] I. Pavlov. LZMA SDK (Software Development Kit). [Online]. Available: http://7-zip.org/sdk.html

[41] M. Nelson and J.-l. Gailly, *The Data Compression Book, 2nd Edition*. M&T Books, 1996.

[42] D. Salomon and G. Motta, *Handbook of Data Compression*, 5th ed. Springer Publishing Company, Incorporated, 2009.

[43] M. Mahoney. (2013, Apr.) Data Compression Explained. [Online]. Available: http://mattmahoney.net/dc/dce

[44] C. Bloom. (2010, Aug.) Deobfuscating LZMA. [Online]. Available: https://cbloomrants.blogspot.pt/2010/08/08-20-10-deobfuscating-lzma.html

[45] (2016, May) Snappy: a fast compressor/decompressor. Google, Inc. [Online]. Available: http://google.github.io/snappy/

[46] J. Alakuijala, E. Kliuchnikov, Z. Szabadka, and L. Vandevenne. (2015, Sep.) Comparison of Brotli, Deflate, Zopfli, LZMA, LZHAM and Bzip2 compression algorithms. [Online]. Available: https://www.gstatic.com/b/brotlidocs/brotli-2015-09-22.pdf

[47] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[48] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006.

[49] R. Lenhardt and J. Alakuijala, "Gipfeli - High speed compression algorithm," in *2012 Data Compression Conference, Snowbird, UT, USA, April 10-12, 2012*, 2012, pp. 109–118.

[50] J. Alakuijala and L. Vandevenne. (2013, Feb.) Data compression using Zopfli. [Online]. Available: https://web.archive.org/web/20130402101541/http://zopfli.googlecode.com/files/Data_compression_using_Zopfli.pdf

[51] L. Vandevenne. (2013, Feb.) Compress data more densely with Zopfli. [Online]. Available: https://developers.googleblog.com/2013/02/compress-data-more-densely-with-zopfli.html

[52] J. Alakuijala and Z. Szabadka, "Brotli Compressed Data Format," RFC 7932, July 2016. [Online]. Available: https://tools.ietf.org/html/rfc7932

[53] (2015, Mar.) WOFF File Format 2.0. [Online]. Available: https://www.w3.org/TR/2016/CR-WOFF2-20160315/

[54] (2016, Aug.) Hypertext Transfer Protocol (HTTP) Parameters. [Online]. Available: http://www.iana.org/assignments/http-parameters/http-parameters

[55] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," 1994.

[56] K. Sayood, *Introduction to Data Compression*, 3rd ed. Elsevier, Inc., 2006.

[57] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A locally adaptive data compression scheme," *Communications of the ACM*, vol. 29, no. 4, pp. 320–330, Apr. 1986.

[58] J. Cleary and I. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Transactions on Communications*, vol. 32, no. 4, pp. 396–402, 1984.

[59] I. H. Witten and T. C. Bell, "The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression," *IEEE Transactions on Information Theory*, vol. 37, no. 4, pp. 1085–1094, 1991.

[60] D. Shkarin, “PPM: One step to practicality,” in *Data Compression Conference, 2002. Proceedings. DCC 2002*. IEEE, 2002, pp. 202–211.

[61] P. G. Howard, “The design and analysis of efficient lossless data compression systems,” Ph.D. dissertation, Brown University, 1993.

[62] (2017, Mar.) Usage of Gzip Compression for websites. Q-Success. [Online]. Available: https://w3techs.com/technologies/details/ce-gzipcompression/all/all

[63] R. Mehboob, S. A. Khan, and Z. Ahmed, “High speed lossless data compression architecture,” in *2006 IEEE International Multitopic Conference*. IEEE, 2006, pp. 84–88.

[64] R. Mehboob, S. A. Khan, Z. Ahmed, H. Jamal, and M. Shahbaz, “Multigig lossless data compression device,” *IEEE Transactions on Consumer Electronics*, vol. 56, no. 3, pp. 1927–1932, 2010.

[65] M. A. A. Elghany, A. E. Salama, and A. H. Khalil, “Design and implementation of FPGA-based systolic array for LZ data compression,” in *2007 IEEE International Symposium on Circuits and Systems*, May 2007, pp. 3691–3695.

[66] K. Papadopoulos and I. Papaefstathiou, “Titan-R: A Reconfigurable hardware implementation of a high-speed compressor,” in *Field-Programmable Custom Computing Machines, 2008. FCCM’08. 16th International Symposium on*. IEEE, 2008, pp. 216–225.

[67] I. Shcherbakov, C. Weis, and N. Wehn, “A high-performance FPGA-based implementation of the LZSS compression algorithm,” in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 449–453.

[68] S. Rigler, “FPGA-Based lossless data compression using GNU Zip,” Master’s thesis, University of Waterloo, 2007.

[69] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, and K. Sheng, “FPGA implementation of Gzip compression and decompression for IDC services,” in *Field-Programmable Technology (FPT), 2010 International Conference on*. IEEE, 2010, pp. 265–268.

[70] M. S. Abdelfattah, A. Hagiescu, and D. Singh, “Gzip on a chip: High performance lossless data compression on FPGAs using OpenCL,” in *Proceedings of the International Workshop on OpenCL 2013 & 2014*. ACM, 2014, p. 4.

[71] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, “A scalable high-bandwidth architecture for lossless compression on FPGAs,” in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE, 2015, pp. 52–59.

[72] A. Martin, D. Jamsek, and K. Agarawal, “FPGA-based application acceleration: Case study with Gzip compression/decompression streaming engine,” *ICCAD Special Session C*, vol. 7, 2013.