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

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.

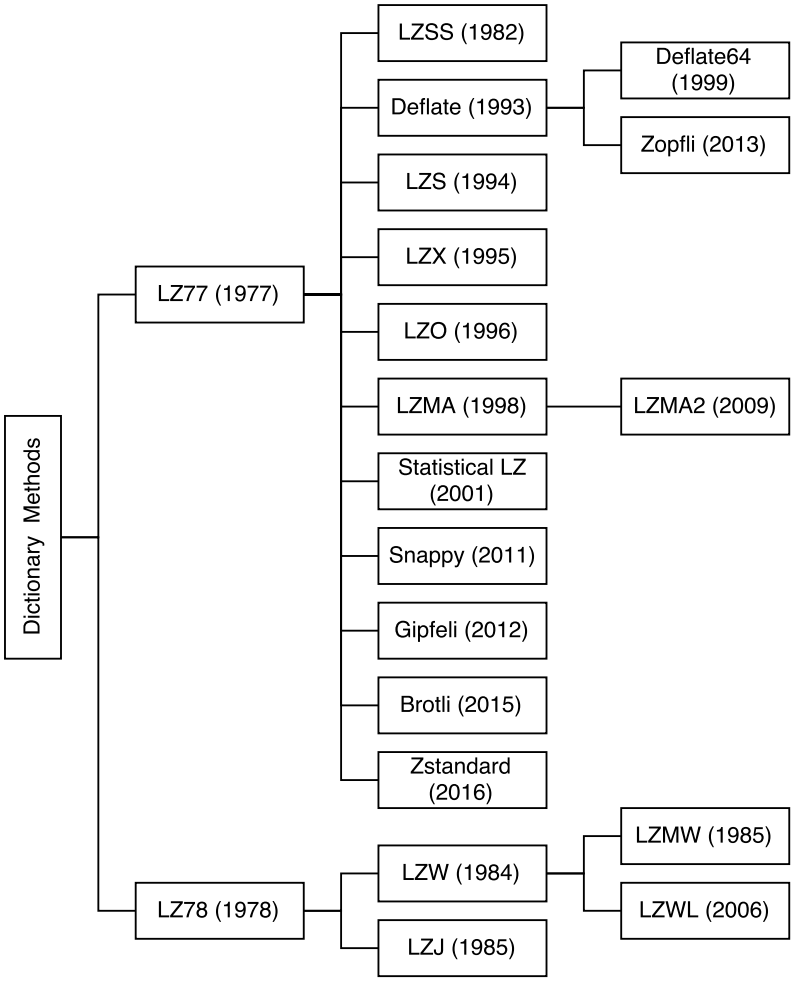

Figure 2.1: LZ class of algorithms

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.

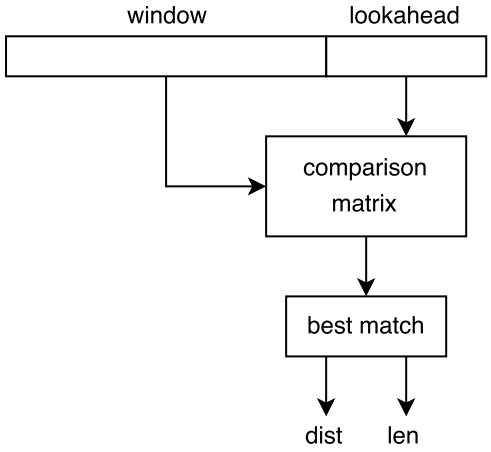

Figure 2.2: General hashless architecture

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.

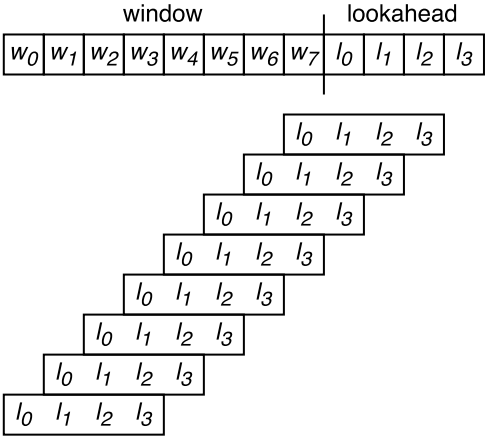

Figure 2.3: Representation of the comparisons that the comparison matrix must perform for $W = 8$ and $L = 4$

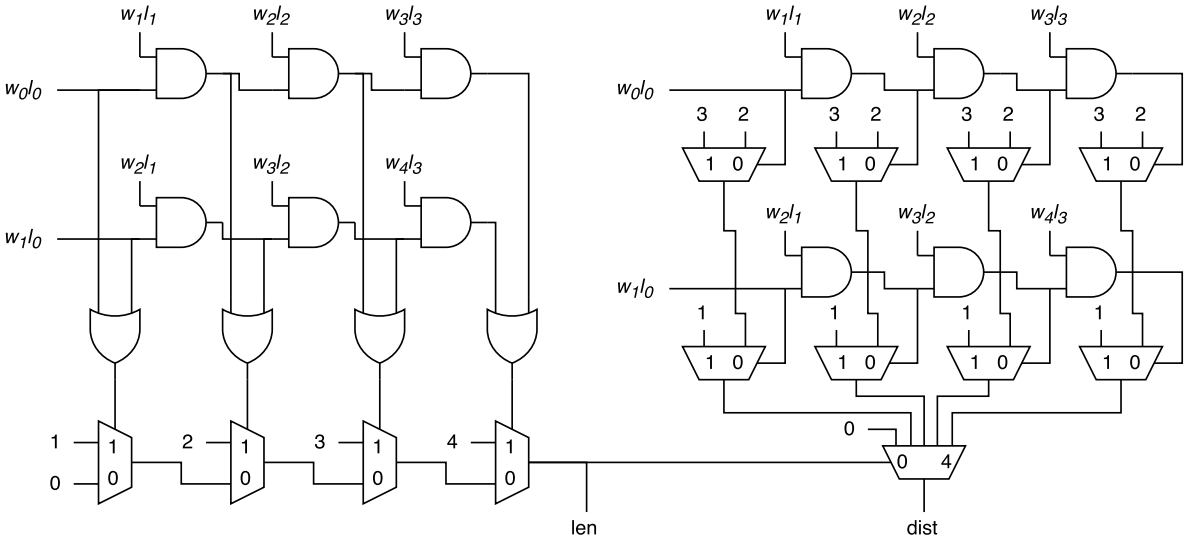

Figure 2.4: Architecture for a best match calculator with $W = 3$ and $L = 3$. The AND gates from the left and right halves of the diagram are the same.

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 $L_e = L$.

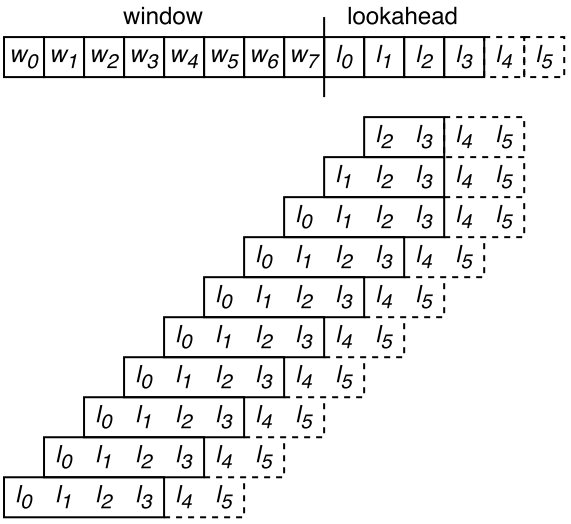

Figure 2.5: Unrolling of the comparison matrix to match $L_e = 2$ future lookaheads

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.

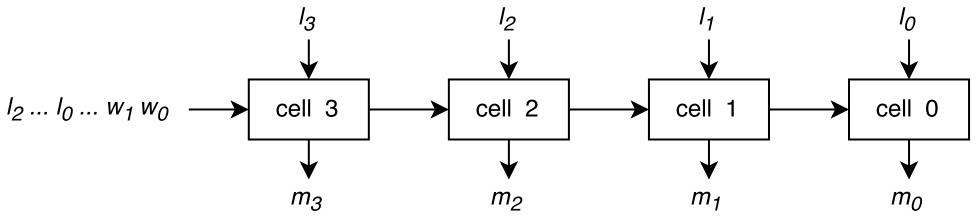

Figure 2.6: Systolic array architecture for $L = 4$

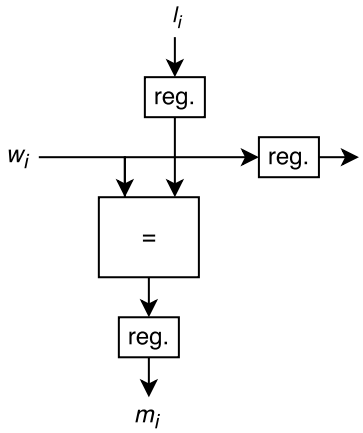

Figure 2.7: Architecture of a cell of the systolic array in Figure 2.6

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.

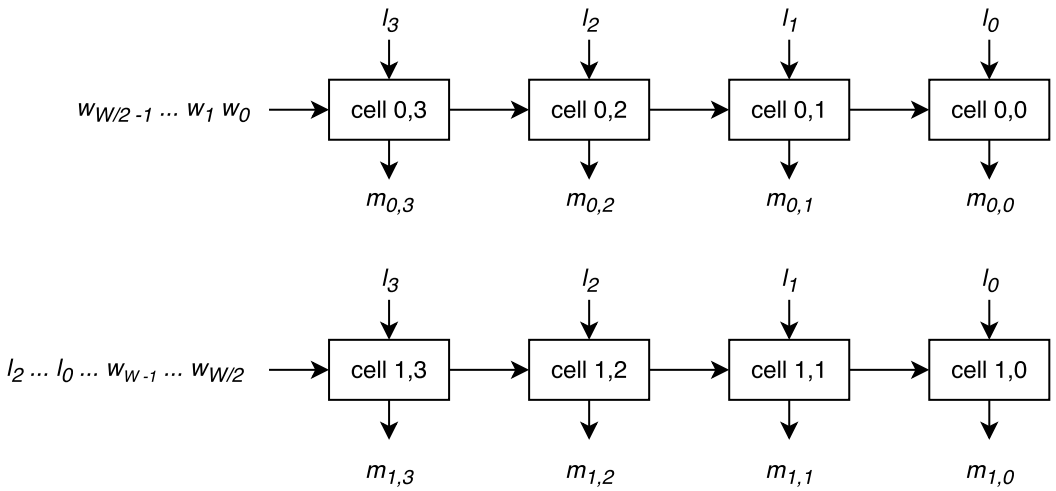

Figure 2.8: Using two systolic arrays to improve throughput, for $L = 4$. The upper array matches the first half of the window, while the lower matches the second half.

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.

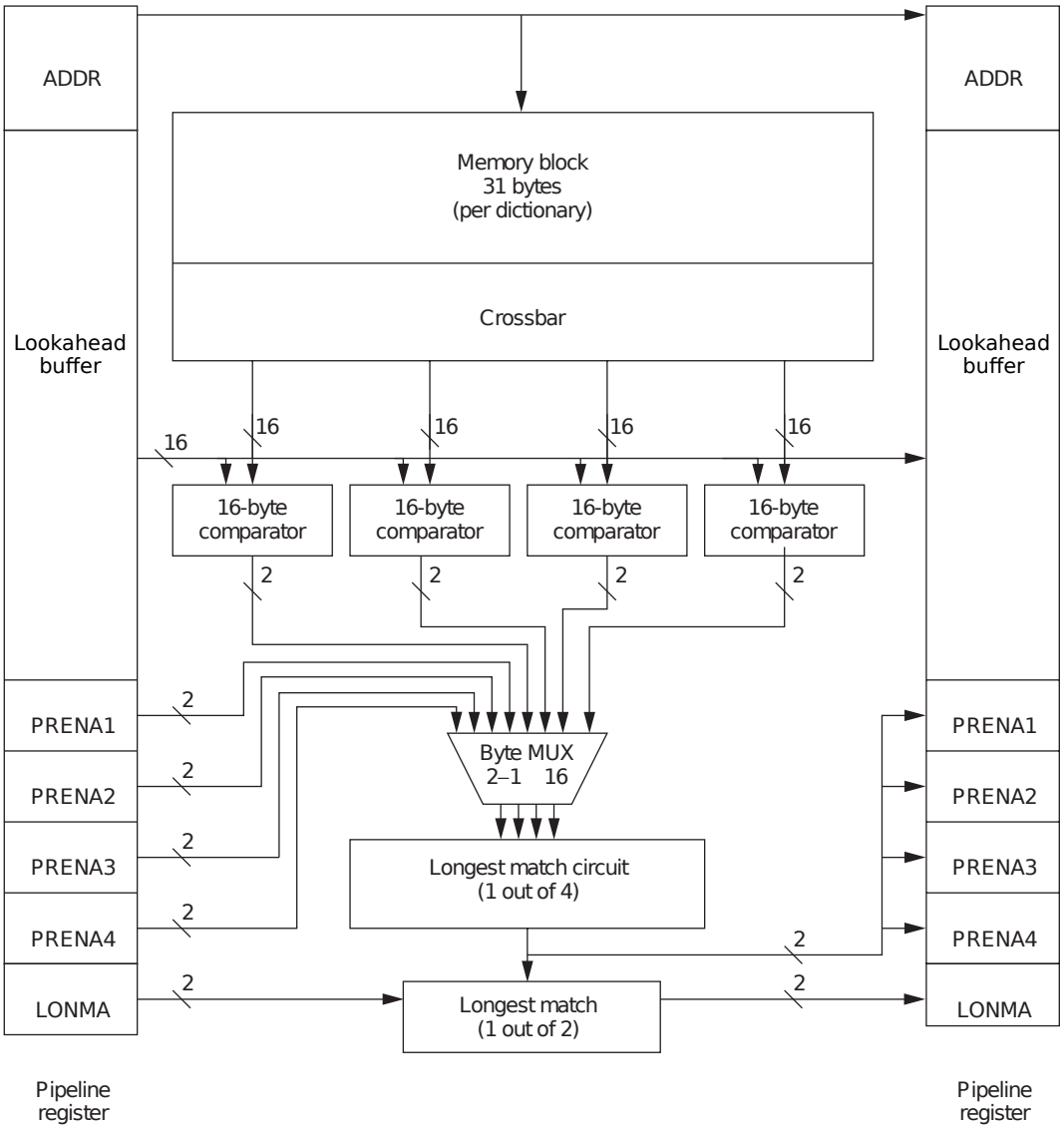

Figure 2.9: Pipelined dictionary architecture [66]. Bus widths are in bytes.

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

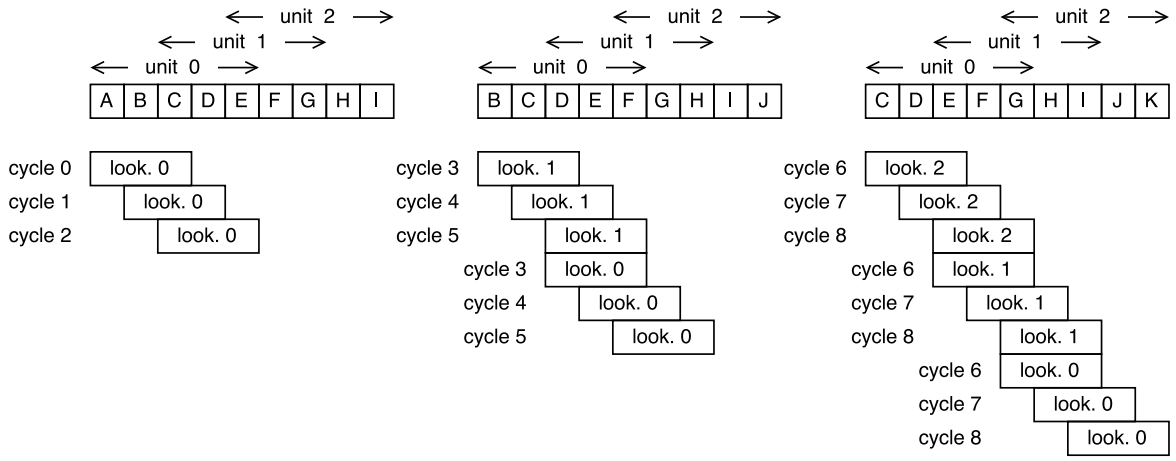

Figure 2.10: Scheduling of the comparisons in each processing unit in order to use a non-static dictionary, for $L = 3$. Note the dictionary moves one position from right to left for every 3 cycles.

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

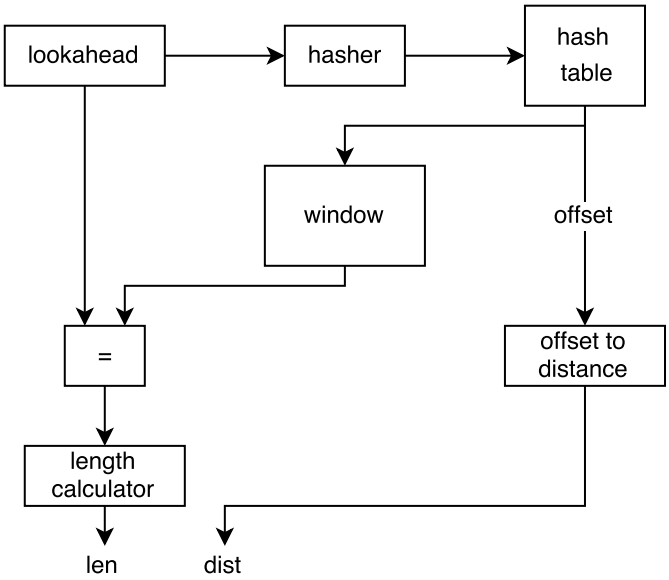

Figure 2.11: General hashful architecture

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.

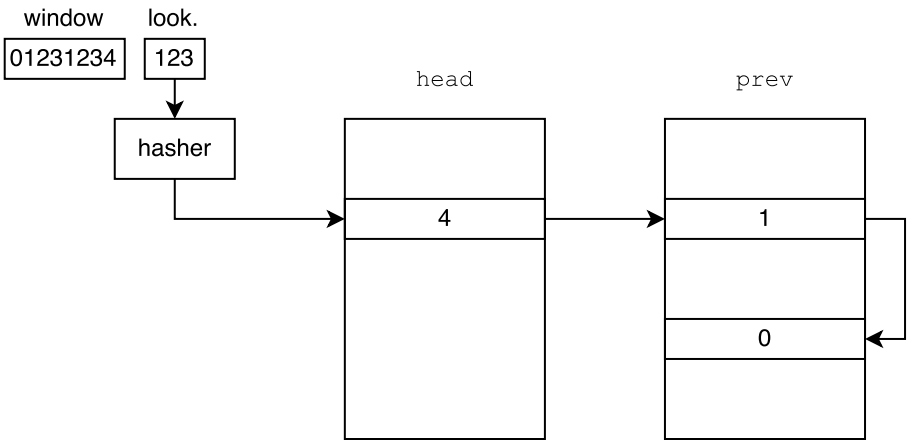

Figure 2.12: Example of addressing a linked list hash table

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.

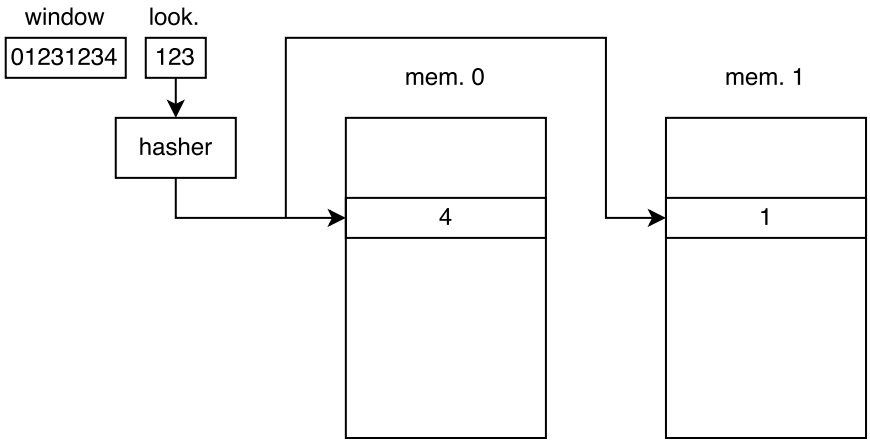

Figure 2.13: Example of addressing a multi-slot hash table implemented with two memories

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

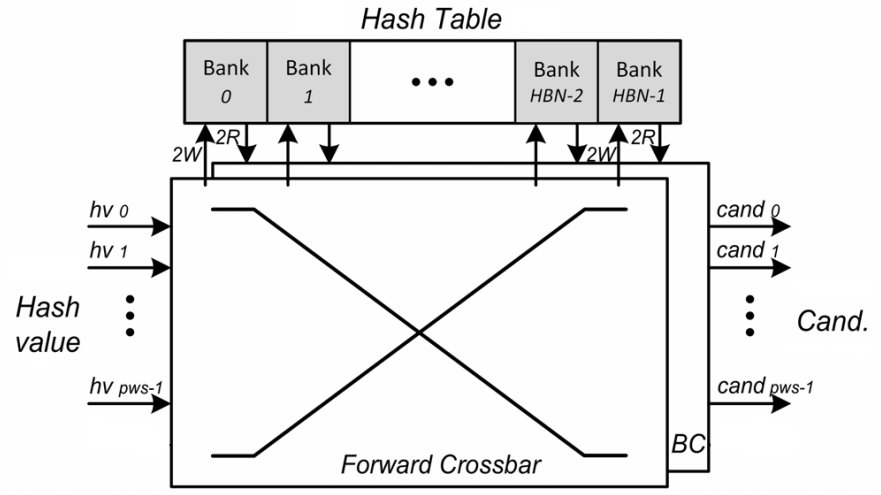

Figure 2.14: Multi-bank hash table architecture [71]

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.

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.