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

May 2017

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`

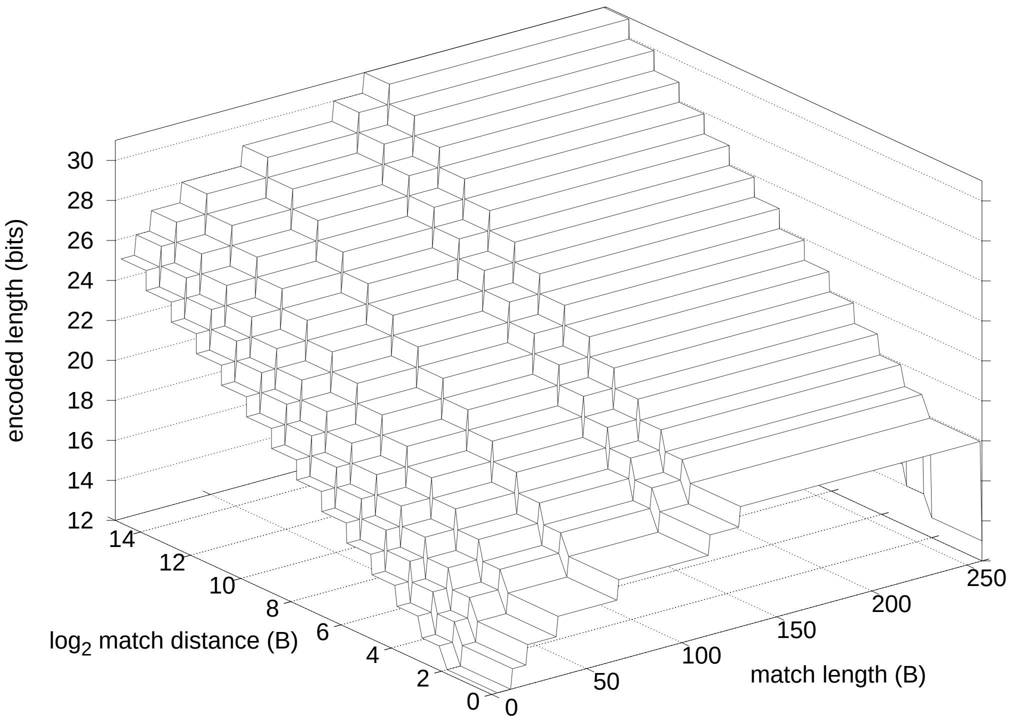

Figure 3.1: Number of encoded bits for distance-length pairs

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:

```
AAAAAAAAAAAAAAAAAAAAAA00000000000AAAAAAAAAAAAAAAAAAAA
```

LZ77 output:

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

Deflate output:

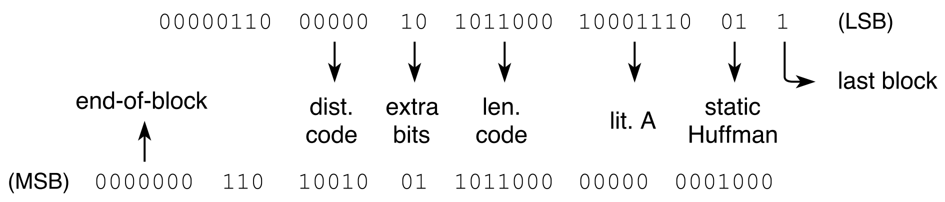

Figure 3.2: Example of encoding an input stream as a Deflate block

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.

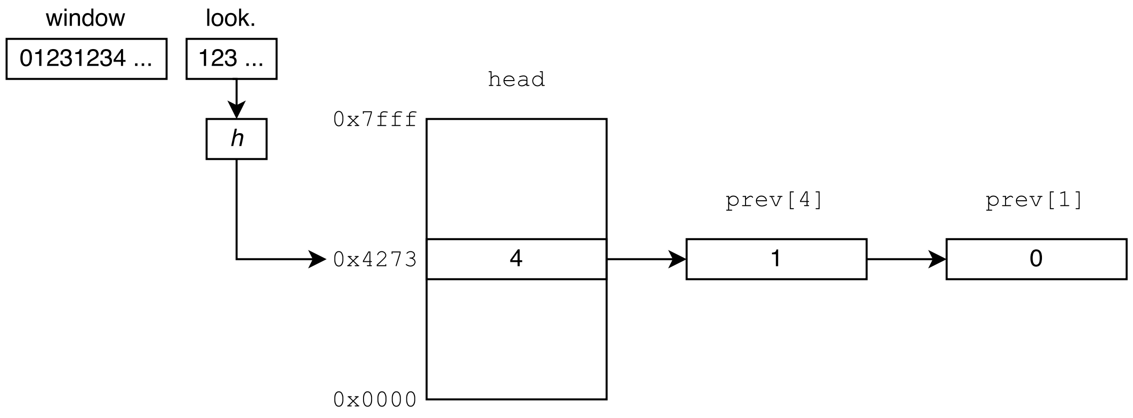

Figure 3.4: Example of a hash chain containing two positions with the same hash as the current lookahead

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.

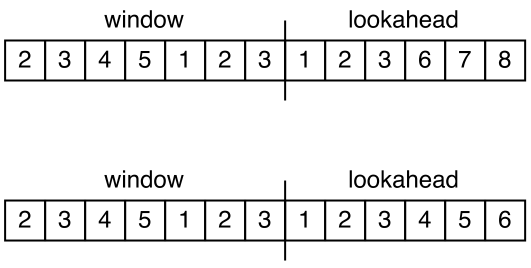

Figure 3.5: Example inputs to illustrate Gzip’s inter-lookahead selection. Top: a match for `123` will be encoded. Bottom: literal `1` will be encoded followed by a match for `2345`.

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.

[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