POLITEHNICA UNIVERSITY OF BUCHAREST

Doctoral School of Electronics, Telecommunications and Information Technology

Decision No. 937 **from** 21-10-2022

Ph.D. THESIS SUMMARY

Ing. Sabin BELU

METODE AVANSATE DE COMPRESIE IN AUTOMOTIVE

ADVANCED DATA COMPRESSION METHODS IN AUTOMOTIVE

THESIS COMMITTEE

Prof. Dr. Ing. Gheorghe BREZEANU Politehnica Univ. of Bucharest	President
Prof. Dr. Ing. Daniela COLŢUC Politehnica Univ. of Bucharest	PhD Supervisor
Prof.Dr.Ing. Ioan TĂBUŞ Tampere University, Finland	Referee
Prof.Dr.Ing. Daniela TĂRNICERIU Technical Univ. “Gh. Asachi”, Iaşi	Referee
Conf.Dr.Ing. Daniela FAUR Politehnica Univ. of Bucharest	Referee

BUCHAREST 2022

Table of contents

Chapter 1

Introduction

Data Storage has a story of its own. Since its creation in 1956, hard disk drives capacity has increased 50-million times. The first hard drive weighted 1 ton and only hold 5Mbytes of data. After 26 years, the first 1 GB (GigaByte) hard drive has been produced and during 2007 to 2011, hard drive capacities quadrupled from 1 TeraByte (TB) to 4 TB. Within the next ten years, 20 TB hard drives became as common as digital cameras. Notions like PetaBytes, ExaBytes, ZettaBytes and YottaBytes are common these day. They represents 1 Million Giga Bytes, 1 Billion Giga bytes, etc.

Our lives are now surrounded by Big Data. This is a concept that we are still trying to grasp these days, since there is no standard definition of this term. We only know that Big Data is defined by some characteristics. Big Data is a combination of structured, semi-structured and unstructured data collected by certain organizations, which can be exploited for information and can be used in machine learning projects, predictive modeling, and other applications, such as advanced analysis.

Systems that process and store big data have become a common component of data management architectures in organizations, combined with tools that support big data analytic usage. Big data is larger, more complex data sets that are so voluminous, that traditional data processing software can't manage them. The advantage of big data is that it can be used to address business problems that couldn't be approached before.

Even though big data does not quantify to a specific volume of data, implementations most often mean terabytes, petabytes, and even exabytes of information that is created and collected over time.

Big data contains greater variety, volume, velocity, veracity, and value, five keys to making big data a huge business.

1.1 Applying Data Compression on Big Data

In terms of compressing the Big Data, humanity realized that classic algorithms have long peaked. Sure enough, applying regular compression on Big Data has its benefits,

but if a compression rate of 90 or even 99 is taken into account, there is 1 left to deal with, store or transport on different communication channels. And this is a huge challenge. As the size of a compressed Big Data stream archive could be in the in the range of tens of petabytes, even if the compression ratio is higher than 98 or 99%.

These kind of compression 'savings' have zero relevance these days. That being said, it is necessary to apply a new compression techniques capable of providing an even greater compression rate than 99.998%.

This method is called differential compression or binary delta compression.

1.2 Automotive and Big Data

The Big Data 'syndrome' affects more and more industries, and Automotive is one of them. Car systems and technologies grew to a level of complexity that hasn't been seen before. The more complex a system is, the more files, folders and data it processes. Audio files, movies, maps, general automobile information, other infotainment data, they all tend to be in gigabytes of data. Processing such Big Data when it comes to flashing or installing the software on automobile Electronic Devices (ECUs) soon becomes a burden due to versioning.

Car systems have many versions as compilations are done on a daily basis and new software versions appear maybe weekly. Dealing with so many software versions and so much data, requires special algorithms, far beyond the regular classic compression algorithms.

Automotive has a very strong reason to use binary delta compression

Software UPDATE and why car systems have to stay up-to-date.

These days, car systems and the projects that run them grow at a level of complexity that has not been seen before. The more complex a system gets, the more prone to errors it is. Defects or Diagnosis Trouble Codes (DTCs, in Automotive terms) are events in which the system jumps from a known state of a finite state machine to another state that has not been taken into account by the architects, developers and test engineers of the entire system.

If this happens without a warning, a *Fault* is generated, and depending on its severity, the display cluster or Kombi, in Automotive terms meaning the Cluster Instrument, could display this as a Check-Engine lit-up icon.

Chapter 2

Dictionary Based Compression

2.1 Introduction

This chapter focuses on dictionary based compression techniques and algorithms. It starts with the presentation of several well known dictionary based compression algorithms like Lempel Ziv '77 [115], followed by its variants LZSS [117] and LZP [113]. The description of a brand new algorithm is also presented, called ROLZ (Reduced Offset Lempel Ziv), an undocumented, patent-free dictionary based compression algorithm reverse engineered from partial descriptions of LZP, LZRW-4 [142] and RKive [112] compression software.

The original contributions on the subject of dictionary based compression are:

- Back-trace an undocumented algorithm, the RoLZ algorithm, from notes and descriptions of LZP, LZRW-4 and RKive algorithms.
- Improvements of the RoLZ algorithm over the original version suggested by Charles Bloom and Malcolm Taylor.

This has been published in the following Conference Paper *RoLZ - The Reduced Offset LZ Data Compression Algorithm* [128].

2.2 Dictionary Based Lossless Data Compression

We have written about this ROLZ algorithm as a tribute to the golden age of data compression. Its unravel was both time consuming in order to deduce the way the algorithm works and also trying to find if our findings are correct.

Unlike classical Lempel Ziv implementations, ROLZ uses a 'reduced' subset from which a possible 'match' set is chosen and also it minimizes the information needed to describe this match-length set. The big advantage of such approach is higher compression

ratio, at some decompression speed expense. Our three algorithm embodiment for LZSS[117], LZP[113] and ROLZ[129] are tested against five types of data, and in all our tests, the compression ratio of ROLZ is far superior to LZSS' or LZP's

The appearance of RKIVE[112] in the early 90's as de facto one the best data compression tools on the market, created a lot of commotion and rumours in the bbs/online communities. RKIVE was written by Malcolm Taylor [112], a New Zealand developer and researcher in data compression. It was soon adopted by researchers and engineers all over the world, as the algorithm was achieving compression ratios never seen before. This was a world that has been dominated for years by PkZIP/PkArc[38], LHa/LHarc[122] or Arj[119] (all trademarks are the property of their respective owners).

Also, during the '90s and mostly in the Dot-Com boom era, a lot of patents were issued in Data Compresion. ROLZ, the acronym from *Reduced Offset Lempel Ziv*, which was later discovered to be the underlying compression algorithm within RKIVE, is even today free of patents, to the best of the authors' knowledge. It is worth mentioning Robert Jung's US Patent 5,140,321 [119], also known as the *LZ77 Limited Search Patent*.

This idea played an important role in shaping ROLZ and it will be discussed later. Meanwhile, RKIVE was closed-source and its ROLZ algorithm remained undocumented for years; some clues emerged from the readme file associated with RKIVE compression package. Within the document, Malcolm [112] acknowledges Charles Bloom for his noticeable insights and helps on writing his program.

Going back one year, in 1995, Charles Bloom has just invented a new algorithm that he presented at The Data Compression Conference in Utah, in the following year [113]. This algorithm was called *Lempel Ziv Prediction* (LZP). In his paper describing LZP, appears from the first time the following line: *"This algorithm works by reducing the set of available window position for an LZ77 encoder to match from"*. Furthermore, comparing LZP with LZ77, it turns out that fewer bits are used to indicate a match-length pair, since only the length is transmitted to output location. Charles Bloom presented four versions for LZP algorithms: from LZP1, which used a fixed order-3 context [1]hash table lookup and a 16k byte LZ-window to LZP4, which used an order-5 context and no hashing.

However, not even the best LZP variant could come close to RKive results in compression ratios. It turned out that RKive was a successful combination of compression methods and heuristics, from solid compression to optimal LZ parsing.

But RKive's powerful compression algorithm will remained a mistery for many years to come. Our article's purpose is to depict the power that lies within this algorithm and maybe, once for all, re-enact ROLZ towards wider usage within software programs.

[1]Using a context in a data compression method is explained by I. Witten in his paper [134]. Ian Witten: *In a sophisticated compression system, where predictions are conditioned by a context of preceding characters or are otherwise drawn from more than one symbol distribution, a context number would also be passed to index an array of coders.*

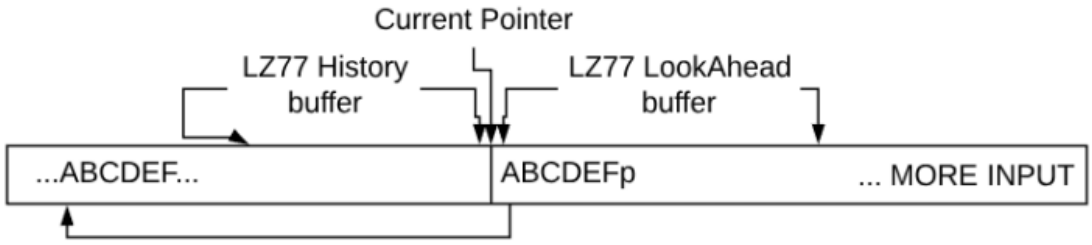

Fig. 2.1 Diagram of the *LZ77* algorithm.

2.3 LZ77 algorithm

Lempel-Ziv's 1977 algorithm [115] is a dictionary compression algorithm. The algorithm's core functionality is replacing sub-strings of commonly seen successions of symbols from the input stream into pairs of position and length. As shown in Fig. 2.1, LZ77 splits the input stream into history and look-ahead buffer; any sub-string portion of a <match, length > pair points to a copy of it in the history part of the buffer. The output encoding consists of a triple ⟨d, l, s⟩ meaning *distance*, *length*, *symbol*, where *symbol* is the first literal or unmatched symbol following the match-length pair ⟨d, l⟩.

Suppose there is a string S, which starts at i^{th} position of the input stream, called current pointer, see Fig. 2.1. Symbols at current pointer are compared with similar symbols from the *LZ History* or symbols that have the same hash value when combined in a *"string"*, a collection of symbols from the used alphabet. There is no context involved in choosing the *LZ-History "string"* to be matched against. Only the current pointer makes the separation between the *LZ-History* and *LZ-Lookahead* partitions, within the input stream.

- The string $S_{i\cdots i+l}$ of length l has another occurrence P, which starts d positions earlier in the text, $S_{i-d\cdots i-d+l}$
- This earlier occurrence of S, should be always less than l_{max} and should start within a window $S_{i-dmax\cdots i-1}$
- The values d and l must satisfy the following constraints: $d \leq d_{max}$ and $l \leq l_{max}$
- Strings $S_{i\cdots i+l}$ and $S_{id\cdots id+l}$ overlap only if $d \leq l$, which makes *LZ77 a self-compressible algorithm*
- When in greedy mode, LZ77 always try to maximize l from all the possible occurrences of P_j in history
- The triple $\langle d,l,s\rangle$ could be further encoded using $log_2(d_{max}) + log_2(l_{max}$ - *LZ_MIN_LEN*$)$ bits

LZ_MIN_LEN is the minimum encoding length beyond which a match is accepted.

It was until Storer and Szymansky made their LZ77 version famous, with a variant called LZSS[117]. With LZSS [117], the authors introduced the concept of $coding flags$, used to differentiate a literal from a distance/length pair. This *1-bit* flag eliminates the need to output a $triple$ at each iteration of the algorithm, thus saving a literal for a new search and possibly, a new match.

The ROLZ algorithm is implemented into 7 distinct steps, of which the first 5 are derived from LZP:

- 1) In the first step, ROLZ calculates the context value *hashIndex* for the current position i, by inserting the previously seen *cntxN* symbols into a hash function hF.
- 2) In the 2^{nd} step, ROLZ checks this value against a hash table. In ROLZ case, this value is a pointer to a collision nodes list; if the pointer is null or the list is empty, ROLZ goes straight to step 4. If the pointer is not null or the collision nodes list is not empty, ROLZ steps into the 3rd step.
- 3) In the 3^{rd} step, ROLZ performs a match trying to find the longest length of a possible match between the string at current position i and the string at position pointed out by collision nodes list's first node; the length l node1 specifying the longest match found during this current search is stored.
- 4) In the 4^{th} step, ROLZ tries to maximize the longest match by advancing to the next node from this collision nodes list; so step 3 is repeated for each node until the list has depleted or max nodes searched is reached.
- 5) In the 5^{th} step, ROLZ calculates the longest length out of all matches, $l_{max} = max(l\_node_N, l\_node_{N-1}, \cdots$,l_node$_1)$; this l_{max} is also stored, along with the index where it was found, named herein n_{max};
- 6) In the 6^{th} step, ROLZ inserts the current position, i into the current collision nodes list;
- 7) In the final step, ROLZ writes l_{max} to the output location. if l_{max} is not null, n_{max} is also sent to output location. The pseudo code of the complete algorithm with an order-4 context is in [**?**].

In the LZP algorithm, the string at the current pointer from *LZ-Lookahead* buffer's i_{th} position is matched only with strings following the same context from the *LZ-History*. The longest match is encoded only by its length, $_l$, not by a pair <d,l>. The current pointer is always written in the hash table. If no match is found, the current symbol, S_i, is written into output location and the current pointer advances to the next location. If a match of length l is found, l is written into the output stream and the current pointer advances $l + 1$ locations.

Table 2.1 LZSS, LZP AND ROLZ COMPRESSION RATES

	ORIGINAL SIZE (MiB[2])	LZSS (%)	LZP (%)	ROLZ (%)
Executable	701	68.63	65.04	**64.43**
Formatted Text	135	25.15	23.00	**21.87**
Object Files	526	43.68	37.87	**37.33**
Text Files	86	58.95	58.17	**54.68**
Misc. Binaries	555	56.46	54.84	**53.92**

Table 2.2 DECOMPRESSION TIMES FOR ROLZ VS LZSS AND LZP

	ROLZ (ms)	ROLZ vs LZSS (times)	ROLZ vs LZP (times)
Executable	13,312	1.03	2.55
Formatted Text	1,037	3.02	2.46
Object Files	4,413	2.26	2.08
Text Files	954	2.04	2.26
Misc. Binaries	7,933	3.58	2.40

In the ROLZ algorithm, the string at the current pointer from *LZ-Lookahead* buffer's i_{th} position is matched only with strings following the same context from the *LZ-History*. All matches l_p are stored and the maximum length l_{max} is computed. The current pointer is always added to the collision nodes list. If l_{max} is zero, no match is found, the current symbol Si is sent to output location and the current pointer advances to the next location. If l_{max} is not zero, it is sent to output and the current pointer advances l_{max+1} locations.

2.4 Experimental results on LZSS, LZP and ROLZ

We have implemented LZSS [118], LZP [113] and ROLZ [142] variant. As test results, we provide data compression ratios and decompression speed timing which are available from tests performed with applications we have implemented for this article; these applications are embodiment of LZP [113], LZSS [117] and ROLZ [142] algorithms. We have used test data sets which we believe to be generically applicable for any data compression application, as explained in details:

2.5 ROLZ Conclusions

We find ROLZ to be an exceptional algorithm. As compared to LZSS, the output of *cntx* bytes allows us to preserve a certain degree of context in the output stream. This implies that ROLZ output is suitable for a $cntxN-order$ compression modelling, providing even better compression ratio, once the output is to be further compressed. While LZSS is usually bound to finite LZ window, LZP and ROLZ can work in an environment where

position and distance of matches are infinite in values, with ROLZ providing the best compression rate, among all three algorithms.

Chapter 3

A Fast Canonical Huffman Decoder

This chapter focuses on entropy coders, in particular, the Huffman encoding and its variant called Canonical Huffman encoding. It starts with the presentation of the ubiquitous Huffman algorithm, followed by the canonical rules that make up the formation of the canonical Huffman prefix codes, hence the name of the variant, Canonical.

The description of an original algorithm is also presented, called Fast Canonical Huffman Decoder (FCHD), which by design, focuses on the creation of a high throughput decoding algorithm which uses canonical Huffman codes.

The original contributions on the subject of canonical Huffman entropy coding are:

- A novel canonical Huffman decoding method, with the ability to process 8 to 12 symbols in a single decoding cycle.
- New and improved method for creating the classic Huffman tree.
- New and improved method for canonical Huffman codeword lengths storage.

This is an original algorithm not an improvement of any of the previous work art presented. Thoroughly tested, the algorithm was able to surpass the 2GiB/s decompression speed mark. The text of this article has been published in the following Conference Paper: *Fast Huffman Canonic Decoder* [131].

3.1 The Huffman Process

Huffman codes have been around since 1951 when they were invented by D.A.Huffman [81]. The problem D.A.Huffman was trying to solve was finding the most efficient method of representing a symbol to be encoded into a more compact binary form. D.A.Huffman solved this problem with a simple idea, an idea that brought him the highest Information Theory rank of underpinning technical achievers of all times.

In theory, Huffman[81] codes approach optimality but when it comes to implementation, the method is much less optimal especially when fewer symbols are encoded. We will explain this below.

3.2 Huffman Encoding Optimality

Huffman coding and in particular, the decoding part, uses the exact same information in order to reconstruct the original input stream; in order to achieve this, step 1 is the hardest to implement in a optimal fashion due to storing or transmission of such mandatory information over an information channel.

This is where the classical Huffman coding is far less optimal in practice then theory. The method proves to be less scalable for larger alphabets, due to the additional information required by step 1 and 2. Apart from the set-notation of the left node to 0 and right node to 1, the classical Huffman tree and generated codes have no additional properties we can rely upon for an optimal storage or transmission, or the omission of such. New progressive codes and coding method had to be found in order to optimally solve this problem and a new set of Huffman codes that follow certain rules have been lately proposed.

Such codes are called the Canonic Huffman codes, and they possess unique properties that will allow us to solve step 1 and 2, the transmission or storage and retrieval of the information required for reconstruction of the input symbols, in an optimal way. Such properties define the canonical Huffman codes to be sequential. They will be discussed in more details during this article, but it is worth mentioning that the most important property of such codes has dramatically changes the paradigm of Huffman coding.This code property, called Consecutive Value property, allows most of the canonical Huffman codes to be automatically generated in sequential order when the number of bits per codeword is provided.

3.3 Our Solution for a Fast Decoding

Unlike any of the classical implementations that we have surveyed, our FCHD implements a proprietary algorithm that allows very fast decoding of multiple symbols in one decoding cycle.

3.3.1 The Canonical Huffman Algorithm

Let $s_1, s_2, \ldots, s_N$ be the following source alphabet symbols, sorted according to their frequency of appearance in the input steam. s_1 being the most frequent symbol and s_N being the least frequent. $l_{s_1} \leqslant l_{s_2} \leqslant \ldots \leqslant l_{s_N}$ the $bit-lengths$ generated by the classic Huffman tree.

The pseudo-code for the Canonical Huffman coding canonical rules could be described as following:

- **Step 1**. $codeword_{s1} = 0$

- **Step 2**: $codeword_{s2} = (codeword_{s1} + 1) \ll (l_{s2}\breve{} - l_{s1})$ [1]

The algorithm starts with $codeword_{s1}$ = zero, at Step 1. We iterate through all the input symbols and generate the next *codeword* by using the formula at Step 2.

3.3.2 Construction of FCHD decoding table

We propose a fast multi-symbol huffman canonic decoding solution implemented using a single lookup table. It requires minimum time for construction and it will be stored in the final output file. This lookup table will be further referred to as the FCHD decoding table.

In order to implement our FCHD algorithm, we partition the huffman binary output stream in segments that follow two rules: A. **The greedy rule** B. **The limitation rule**: their cumulated codeword length is less or equal to an arbitrary chosen N.

For simplicity of examples, we have chosen to present this algorithm for $N = 8$.

We use the encoding of the message $s_0 \ldots s_3$ with the canonical Huffman codes from Table 3.1.

We then use the segment value to generate the FCHD table index. This index is generated from a segment and a remainder of bits, if any. The remainder exists only if the cumulated codeword length is higher than N.

FCHD table index is always of length N bits.

Using FCHD table index, we then store the additional information required for fast decoding in a location of the decoding table where the FCHD table index points to.

The remainder is a left over from the next instantly decodable codeword which is not fully used because of rule A

The FCHD table index is formed out of a segment and a remainder of bits, if the current codeword length is less than N . FCHD table index is always of length N bits.

[1] left shifting is a bit-wise operation which requires two operands to work, a number and the $shift\_bits$ value; the operation result consists in the first operand being multiplied by 2 power shifted-bits (2^{shift_bits}).

[1] ASCII (/æski/ ASS-kee), abbreviated from American Standard Code for Information Interchange, is a character encoding standard for electronic communication.

Table 3.1 Symbol frequencies and associated codewords

Index	Symbol	Freq	Code Length	Canonic Code	Huffman Code
0	A	9	1	0	1
1	T	5	2	10	01
2	C	3	3	110	000
3	G	1	3	111	001

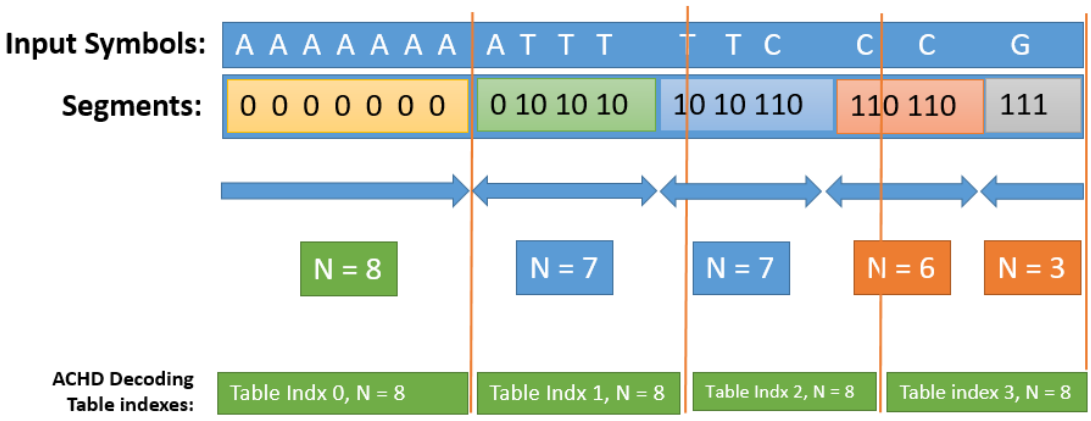

Fig. 3.1 Computing table indexes from C_{bv} and R_{bv}.

3.3.3 Experiments

Our Fast Canonical Huffman Decoder has been tested against different data types such as nucleotides, bitmap files, text and binary files. Other test files relate to an in-house LZFG data compression program we have developed, such as literals (uncompressed symbols), literal runs, lengths, length runs and distances (substring copies coded as back reference pairs of distance and length tokens). All together, they form some specific test files that we use in our tests. Short descriptions of LZFG-A markers and tokens

3.4 FCHD Conclusions

Our Fast Canonic Huffman Decoder, FCHD, is extremely suitable for canonic huffman codes with an average codeword bit length of up to 12 bit. It is an original solution which addresses the need for high-speed big throughput algorithms. The algorithm was designed to exhibit a small storage and memory footprint and we find it also suitable for low resource environments, such as various ECUs [2] from IoT or Automotive. Through empirical studies, we have learned that our FCHD method is able to decode symbols at speeds at about 2GB/s and above while processing highly redundant files.

[2]1

[2]ECU stands for Electronic Control Unit, a small electronic device responsible for controlling specific functions. In Automotive, Electronic Control Units (ECUs) are small devices located inside vehicles that control specific functions such as power steering control, windows, seats or even engine parameters.

Chapter 4

A Quasi-Static Arithmetic Encoding

This chapter also focuses on entropy coders, in particular, the arithmetic encoding. It starts with the presentation of the ubiquitous arithmetic data encoding algorithm, followed by the description of a static version that we have improved.

With this thesis, we are proposing a new architecture for arithmetic encoders called Quasi-Static. Unlike the classical implementations, the Quasi-Static Encoder buffers the input stream and uses a static model for encoding the data in the buffer. The big advantage of such approach is the higher encoding speed obtained however with the price of slight degraded compression rates.

The text of this chapter has been published in our conference paper *The Anatomy of a Quasi-Static Arithmetic Encoder* [129].

Since dictionary based - compression algorithms like Lempel-Ziv [115] contain a level of redundancy in the output, it is important to reduce that redundancy by using a different method, other than based on common sub-strings. Therefore, any complete data compression method must include also an entropy coder.

Arithmetic data compression, as a de facto entropy encoder, has gone through an ongoing improvement process since its first appearance in the famous article "Arithmetic Coding for Data Compression" written by Ian H. Witten, Radford M. Neal and John C. Cleary [109]. This version of arithmetic compression is known as CACM Arithmetic.

One of the utmost difference of an Arithmetic Encoder with respect to most of the entropy coders, such as Huffman (dynamic, also known as Gallager Entropy Coder or static variants), Splay encoding, Shannon–Fano etc. [**?**] is that it encodes the entire message (buffer or file) into a single big number, that is carefully chosen to be a fraction q within the initial interval [0.0, 1.0).

[0]American Standard Code for Information Interchange

4.1 Arithmetic Encoding

A message to be encoded is composed of symbols from a given alphabet. As stated above, arithmetic encoder takes the message and converts it into a fractional number q with the condition that q belongs to [0.0, 1.0).

4.1.1 Output Streaming

In practical implementations the *codeword* is built progressively. For each new encoded symbol, the encoder can release none, one or more bits of the *codeword*.

This process is based on the following observation: within the interval [0.0, 1.0) any number that lies below 0.5 has 0 as MSB (most significant bits) and any number that lies above 0.5, has 1 as MSB.

When LOW and HIGH endpoints lie on the same side of 0.5, c^ will certainly have the MSB of that side. Consequently, a bit is released and in order to reproduce this condition, the binary representations of LOW and $HIGH$ are left shifted.

This is equivalent with a scaling of the interval by 2. The encoding process is described by the following pseudo code:

4.1.2 Decoding Process

At the decoder, the bit-stream is cut into *codewords* and for each *codeword* is identified the interval that originated it. The decoder must be aware of the statistical model otherwise the decoding is not possible. After the decimal conversion of the *codeword*, the decoder deals with a fractional number.

4.2 Practical Consideration

Considering the interval [0.0, 1.0) and the fast way it shrinks, it looks completely impractical to use this interval for an implementation of arithmetic coding. Using [0.0, 1.0) gives out two major defects when implementing arithmetic coding.

Where LOW_s and $HIGH_s$ are the interval endpoints, Cumulated frequencies of symbols are calculated as the sum of previous symbols frequencies and Value per Unit defines the ratio between the range and the sum of frequencies. Suppose that for our example, we want to re-scale the interval [0, 1) to [0, 20). The symbols frequencies and cumulated frequencies are given in the following table:

The cumulated frequencies are helping us to define the interval assigned to each symbol. The endpoints of the interval assigned to symbol Si are calculated as follows:

$$LOW[S_i] = LOW_0 + V_pU \times CumFreq[S_{i-1}] \tag{4.1}$$

$$HIGH[S_i] = LOW_i + V_pU \times CumFreq[S_i] \tag{4.2}$$

where V_pU is in our case 20/10 and LOW_0 is the left endpoint of the initial scaled interval. Here the index i specifies the symbols order in the interval. Without a good mapping, meaning without the use of V_pR, we would have a change of probabilities from the initial estimation.

The partition after the interval scaling is shown in Fig. **??**.

Following 4.1, the fractionally endpoints were mapped to integers: For symbol 'A', the interval [0, 0.5) is mapped to [0, 10), for symbol 'B', the interval [0.5, 0.8) is mapped to [10, 16), for symbol 'C', the interval [0.8, 1) is mapped to [16, 20)

When dealing with full 32 bit or 64 bit the upper limit of the re-scaled interval becomes 4,294,967,295 or $0xFFFFFFFFU$ (base 16).

The result is much larger intervals to work with: $[0x00000000, 0x80000000)$ for symbol 'A', $[0x80000000, 0xCCCCCCCC)$ for symbol 'B', $[0xCCCCCCCC, 0xFFFFFFFF)$ for symbol 'C'

4.3 Updating the Statistics in Quasi-Static Encoding

It is a known fact that arithmetic coding tends to optimality as long as the source model probabilities are equal to the probabilities of the symbols to be encoded.

Any difference reduces the compression ratio. In classic implementations the arithmetic encoder uses a dynamic model that adapts its statistics with each input symbol. All statistics are computed based on the previously encountered symbols.

Since all previous symbols are already known at the n^{th} encoding step, these statistics will be updated into the same way by the decoder as well; no overhead for statistics in needed.

The drawback is its complexity.

4.4 Experimental Results

Tests on the Quasi-Static Encoder has been performed on three types of data.

- The first test series is from Calgary Corpus [1], The size of the first test series is 6,285,846 bytes.
- The second series is a *formatted text file*. The file size is 11,924,676 bytes.
- The third test series is binary code and consists of a collection of *32bit/64bit windows PE/PE32+ executable files*. The size of this series is 21,742,646 bytes.

The larger the buffer becomes the lower the 'stored' frequency table size is as compared to the final output size (Fig. **??**). Thus, compression savings tend to overcome to output. This explains the best compression results for 64 and 128kB buffer sizes. The downside of using an unique frequency table for the entire buffer is a possible decay of the compression rate because of static model mismatch with the instant symbol probabilities. An input that is highly non-stationary will be in places poorly compressed. To evaluate this effect, we compared our Quasi-Static Coder with two dynamically adapting encoders, the former written by Diperstein [126]

4.5 Conclusions

We believe that this approach to data compression is a viable solution when an entropy coder is preferred or speed versus compression rate is a big factor. A drastic improvement on speed with a low impact on compression rate compared to two dynamic Arithmetic Encoders has been observed during tests, thus, making this approach a viable solution to any dynamic entropy coder, especially when arithmetic encoding is preferred.

[1]The corpus was created in the late 80s by Timothy Bell, John Cleary and Ian Witten, in order to be used in their research paper "MODELING FOR TEXT COMPRESSION" [132] at University of Calgary, Canada. ACM Computing Surveys published the research paper in 1989; in 1990 the corpus was used in their book "Text compression"

Chapter 5

Software Update in Automotive

Alongside improvements in functionality, a Software Update or a Patch, can provide fixes to major functionality issues related to various ECUs, ranging from ones that control non-essential functionalities like PIA (Personal Profile Information) such as individual chair settings and adjustments for passengers, to Infotainment glitches, e-Call or even City Stop or Brake Assist related issues. Automotive software is supposed to be 100% reliable in any situation. Software Update which installs new and improved software version is the only feature that makes this possible!

Taking out the human factor away from this equation, it could be possible that most of the issues could have been fixed by Automotive Software UPDATE (Wired or OTA).

Defects in automobiles have tremendous costs. The civil penalties, punitive damages alongside civil and legal costs exceeded 230M last year, in USA alone:

- During one recent year, 30,000 people died on US highways;
- Every year, it is estimated that traffic collisions cost this country a total of about $230 billion dollars in medical and insurance costs and lost worker productivity alone in US only, see Fig. 5.1.

Just like a smartphone, which has the capacity to download the newest OS model over the GSM network on the air or using wireless connectivity without being physically plugged-in a network, the same applied to any new automobile model or IoT device, where a remote management system to handle new software versions is a very demanded feature but also extremely popular due to a plethora of advantages it offers.

There is a harsh need to successfully download and install the newest software packages in mostly all the ECUs that make up the entire automobile system. In Automotive industry, this remote system that manages software update without a hard link connector to a network or storage device, is entitled Firmware Over-The-Air (FOTA) or simply, Over-The-Air (OTA) updates. The remote system that uses a hard wired link connector to a network or storage device, is called Wired (Secure) Software Update.

While we started considering a smartphone and we look at its software abilities to update the running operating system on the air, we need to consider the fact that the

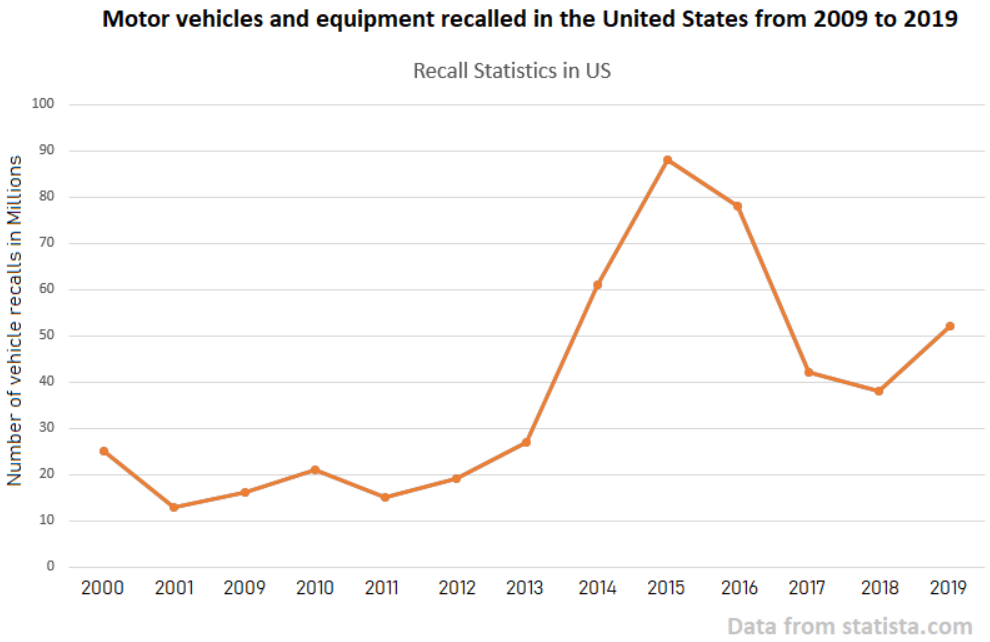

Fig. 5.1 Recall statistics in US.

criticality of these firmware or software updates is much higher in automotive ECUs. The impact of OTA software updates in automotive is extremely high. The user experience from the vehicle owner point of view is constantly upgraded when OTA is involved, since no dealership visits are required, no other expenses occur if the car is out of warranty, and when using an OTA, a new infotainment system could be available and ready to use is less than minutes.

The impact from the absence of OTA is much higher. Critical software updates or glitches in the system remain unfixed until a new dealership visit take place, following a prior appointment, which may put a stress on the system and affect the security of the car occupants. A few examples: in 2015, the Jeep automotive OEM (Original Equipment Manufacturers) had faced a problem when a hacker infiltrated the car electronic system, while the car was on the road. The hackers nearly paralyze the cars by disabling brakes when the car was running on a high speedway interstate road. An analysis of the issue turned out frightening numbers: the glitch affected 1.4 million vehicles, which had to be immediately recalled at numerous dealerships around the world. At that moment, the OEM faced their major crisis and the only cost-optimal solution was given by a masive OTA campaign which *patched* the affected automobile software by properly installing OTA updates into thousands of cars to alleviate the issue.

Another OEM additionally experienced an issue when their vehicle electronic scheme was hacked by some computer criminals in September 2016. As the OEM has been working on cryptographic validation of OTA updates for months and also on software

update encryption and signing, they had to speed up the roll-out of these options after the attack.

It could also be that sometimes, OTA updates are not fully tested before they are rolled out to customers. It happened with another pro-eminent OEM which started an update campaign affecting millions of car, but the newly installed software started to reboot the Infotainment's system (the car head unit display) every minute or so. Sometimes bad design affects the end users more than the absence of software update, since in this case, losing the car information which displays on the car cluster instruments turned out to be a very big problem than an outdated software system, which worked.

From the cost perspectives, a missing OTA update system with a newly launched car model could be very costly for an OEM, which has to pay an hourly rate for each dealership which manually installs updates on a car head unit fees.

5.1 Manual ECU Firmware Updates in Automotive

When it comes to a manual firmware update of ECUs in the car, we rely on the fact that mostly all the ECUs present in a particular automobile, are not interconnected thorough various network protocols and bus links such as CAN, Ethernet, LIN, FlexRay or MOST.

A manual firmware update also relies on a module, usually called Diagnosis Module, which interconnects and communicates with almost all present ECUs. The diagnosis module exports different functionality functions such as Transfer Data, Write Data, Read Data, Read DTCs (Diagnosis Trouble Codes), Remove DTCs etc. The Diagnosis protocols used are KWP-2000, OBD or UDS (Unified Diagnostic Services). The UDS protocol, described by ISO-14229 is now a standard across manufacturers and is used today in ECUs across most Tier-1 OEMs.

5.2 Firmware Over-the-Air in Automotive

Firmware Over-the-Air is made up of three steps. They combine server-side software which is meant to create the software update package along with client-side software that eventually installs the update on the client's machine. There is also an additional step which is different from the manual update process, named deployment. Based on the type of the software update, which could use encryption and/or signature validation, the software update payload may be deployed on secure or non-secure communication channels.

First FOTA Step

The server running on the server side is the first step in the FOTA Software Update, because this is the step where and when it is created the software update package. There are two software versions that are generated in this step:

- The binary difference software that contains the bug fixes and solves any other issues that may have risen during the software patch development and testing.
- The new software version in a full packaged software container that contains the bug fix and/or the new software features.

Second FOTA Step

The deployment part comes in as a second step in this process. The secure software update package containing the binary delta payload is stored into a distribution platform. The OEM or a specific vendor or supplier controls this entire platform, which has the role of sending push notifications to all affected head units from the automobiles, to notify them on the existence of an update targeting their car.

Final FOTA Step

In this final step, the secure software update decoder is running on the client's head unit in the automobile and performs the actual new software installation. The software update package has been already deployed in the cars' head unit hardware and the connection with the Diagnosis Module that handles the installation portion of the software has been initiated.

5.3 FOTA Update Containers

A FOTA process relies on specific packages called Containers. A Software Update Unit, or Container, in short SWUP or SWFK, from Software Firmware, are required to be created in order to deliver the SW to any ECU in the automobile.

Each logical Container consists in a pair of configuration files and binary files, the actual payload. The files are usually called PDX or ODX (open diagnostic data exchange format or package), but their naming convention depends from one OEM to another. There is no standard in place for this.

5.4 Data Chuncking Concept

A fresh concept is adopted these days and it is related to the ability of software update managers to perform the installation in parallel with the download of new data. For this, to be able to implement in practice, the idea of chunking began to take shape.

These installation data chunks could transfer payload data containing either the full version of the software or the binary delta data or configuration files, and they can span few megabytes in size. Each chunk must be equipped with its own CRC and Signature values such that the secure delta decoder or manager verify each individual chunk.

Installation Verification

There are multiple signature and CRC verifications during the update process.

- Signature and CRC32 verification for SWDL (Software Download) Container
- Signature and CRC32 verification for each transferred Chunk
- CRC32 verification for transferred Container modules.

Selective Software Update

The final Container configuration files are generated during the build process based on the build session type (either Development or Production Build type) and are based on a configuration template.

A selective update is based on the Head Unit Model type (the M-type) and also on the hardware sample levels, which are the various hardware types that are shipped by the Tier 1 supplier. For instance an M625 Variant 3 could be designed for ECE and US Markets only and comes with a TV-Tuner hardware module (the variant 1 and 2 may not contain this TV Tuner ECU).

Containers Data Streaming

Some OEMs or Tier 1 suppliers have implemented software update system capable of streaming data in Chunks for parallel installation and processing.

The Diagnosis Module Controller device send the data to the ECU and the data is received in chunks and installed on the inactive partition in parallel by the secure software installer application binary.

Chapter 6

Data Differencing using Referential Compression

This chapter focuses on binary data differencing and, in particular on referential compression (RC) and NCD-based algorithm. It starts with the generic presentation of the referential data compression, followed by the description of suitable data and use cases for this technique, the construction of a referential dictionary and possible hack-attack scenarios.

My contribution to the subject consists in an original algorithm for data differencing based on the differential compression that plays also the role of a generalized distance similar to the Normalized Compression Distance (NCD).

By design, this algorithm may be very suitable for low resources environments and could very well be used in industrial areas such as the Automotive Industry.

The proposed algorithm is innovative under the following aspects:

- It is an generalized NCD-based binary differencing algorithm designed for low resources environments, suitable for IoT or Automotive industries.
- It implements a new version of LZ77 algorithm, modified such to accommodate different block sizes and parameterized dimensions of the input stream.
- Features the ability to output an already compressed output stream as the binary difference between two blocks of data belonging to the two different input streams.

Part of this chapter has been published in the conference paper *An innovative algorithm for data differencing* [130].

When software does not work according to specifications or when new features are required and the effort to re-download and re-install an entire application is too high, a software update is the right solution. A software update is a compact instruction manual in the form of a downloadable file interpreted by the application installing the update. It contains simple commands such as ADD / CUT / $INSERT$ / $COPY$. They tell the

decoder how to reconstruct the original file by starting from the initial file and performing specific inserts, adds, cuts or copies on the initial byte stream.

6.1 Software Updates and Data differencing

Applying an update improves software not only by updating version information but by changing its data. Therefore, software updates or patches consists in the sum of all differences between the old and the improved version of the same software, noted also as *source* and *target* versions. The algorithms that produce these differences are data-differencing algorithms.

The referential compression is one of the techniques that can be used for binary data differencing. It has two essential properties that make it suitable for such application. It relates two different binary sources and outputs the *differences* in a compressed form.

6.2 Referential Compression

Referential compression is a class of dictionary based compression. A dictionary based compression 'core' algorithm functionality is to replace substrings of commonly seen successions of symbols from the input stream into pairs of position and length. It usually splits the input stream into history LZ_HISTORY_BUFFER and lookahead buffer LZ_LOOKAHEAD buffer; any substring portion of a <*match, length* > pair points to a copy of it in the history part of the buffer. The output encoding consists of a triple $\langle d, l, s \rangle$ meaning $distance$, $length$, $symbol$, where *symbol* is the first literal or unmatched symbol following the match-length pair $\langle d, l \rangle$.

6.2.1 Operating Modes of Referential Compression

Referential Compression involves two dictionaries and operates in two modes in both the encoding and the decoding process. These are the internal and external modes.

An internal dictionary is used when Referential Compression operates in internal mode. The internal dictionary which starts with no data (empty) builds up inside the LZ_HISTORY_BUFFER area while the Current LZ Pointer advances (CLZP).

In the external mode, Referential Compression employs a specialized dictionary. This specialized dictionary, in most of the cases, it is an external dictionary (see Fig. 5.2) which acts as a specialized substrings repository database, but also provides an encryption mechanism which will be explained below.

Since the decoder must emulate steps from the encoder, in this operation mode, however, an extra token is required by the decoder to distinguish to which dictionary a particular substring (match) is referred to. In this mode, the output encoding consists

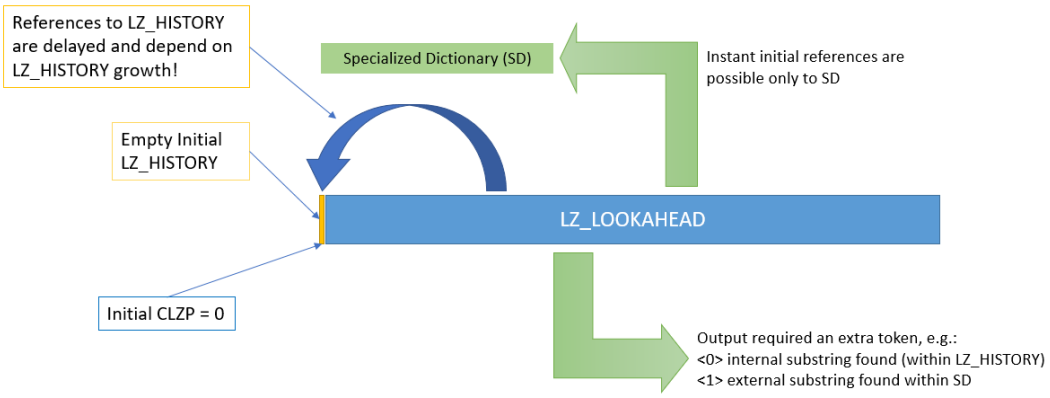

Fig. 6.1 Referential Compression with external Specialized Dictionary.

of a quadruple $\langle t,\ d,\ l,\ s \rangle$ meaning *token*, *distance*, *length* and *symbol*, where *token* represents the index of the dictionary which resulted in a successful match.

6.3 A New Data Diferencing Algorithm

Our new Data Differencing algorithm creates a data differencing file from two files given as parameters in the console command line. The two files are called *source* and *target*. $f1$ usually denotes the source file (old file). We also name $f1$ the dictionary file. $f2$ denotes the target file (new file).

We have designed our differencing algorithm to use the LZ77 sliding window algorithm. As presented above, the CURRENT_POINTER divides the input buffer into two portions, the LZ-Dictionary and the Lookahead buffer. Moving along with LZ77, initially, CURRENT_POINTER is zero, since nothing has been encoded in the input buffer.

Previously processed data, the input data buffer that lies below CURRENT_POINTER form the LZ_HISTORY. The next data to process that lie beyond form the LZ_LookAhead buffer. While the CURRENT_POINTER advances incrementally, new sub-strings are found and encoded.

6.3.1 Operating with our Differencing Algorithm

As noted above, we are devising an algorithm that works on limited resource environments. To achieve this main goal, the input stream is sliced into buffers or chunks. We work with two input streams, X and Y where X is the input stream represented by the source file $f1$, while Y, is the input stream represented by the target file $f2$. Furthermore, we need to define a similarity function which in our terms, computes the common sub-string count between X an Y. We call our function BEST.

We define input I, such as

$$I = xy : x \in X, y \in Y \tag{6.1}$$

and $BEST(I)$ returns the compression rate assuming that buffers x^{th} and y^{th} are concatenated and encoded as a single buffer. We rewrite the function BEST, such as $BEST(x)$ returns an index k where $y = k^{th}$ buffer and $BEST(xy)$ returns the highest compression rates in the above condition.

We chose LZ77 as the main working algorithm, noted under Z, which further applies a sliding window mechanism. We replace $Z(x|y)$ with our BEST function, which becomes thus our similarity function.

We denote $BEST(data\_buffer\_k)$ = i, that is for the k^{th} buffer of file $f2$, the most similar buffer is buffer i^{th}. The higher the similarity of the two buffers, the higher the compression ratio when compressing the two buffers.

The algorithm proceeds as follows:

- **Step 1**: Define N and K values;
- **Step 2**: Split initial stream X into buffers of size K maximum;
- **Step 3**: Split target buffer into chunks and read the first chunk from Y;
- **Step 4**: Calculate $BEST(data\_bufer\_1)$ which outputs an index, i, and read the i^{th} chunk from X;
- **Step 5**: Start processing buffer 1 of X;
- **Step 6**: Finalize encoding of buffer k^{th} from X with buffer i^{th} from file Y, given by the function BEST(k) = i;
- **Step 7**: Repeat Step 6 until Y is depleted;

We have devised a strategy for our BEST function, and in this strategy, we compute all possible values of similarities between buffer x and y. This is proven by design to achieve the best possible result, since for any buffer k of file $f2$, we try out all possibilities by applying the similarity function for all chunks of file $f1$. In this strategy we analyze all buffers from file $f2$ against other buffers from the same file and also against buffers from file $f1$.

We compare our NCD Algorithm with commercial application including DeltaMAX (produced by Indigo Rose Corporation) and X3Delta.

6.3.2 Conclusions

Our Differencing algorithm is able to deliver compression rate results even 50% better than established commercial data differencing software. The results on windows software

Table 6.1 How our Differencing algorithm compares with on-the-market binary delta encoders.

	BEST(f1,f2)	DeltaMAX	X3 Delta
Log Files	0.41	0.35	0.12
Mingw Binaries	30.83	63.96	46.55
SW Packages	54.42	63.07	51.21

package show a 13.72% smaller delta file than Indigo Rose DeltaMAX software. When applied on *mingw* compiler binaries, our Differencing algorithm results are with more than 33% better than DeltaMAX.

Chapter 7

Data Differencing

This chapter also focuses on binary data differencing and in particular, on modified dictionary based compression algorithms suitable for data differencing software. It starts with the generic presentation of software update scenarios, followed by the description of server - side software that creates the binary delta file and also the client-side software that performs the software update installation.

The description of a new and original algorithm based on LZ77 is also presented. Three methods to create binary data differences are presented, mostly designed to accommodate different scenarios starting from faster processing to a scenarios solely designed to achieve best possible data compression rates.

The original contributions on the subject of binary differencing based on modified data compression algorithms are:

- An original algorithm for binary data differencing based on modified Lempel Ziv '77 [115].
- The algorithm has been designed for low-resources environments which make it suitable for industries like IoT or Automotive industries.
- Faster decoding algorithm in low resources algorithm.
- The development of three working strategies for this algorithm, to accommodate different use cases of this algorithm.

This is also an original algorithm not an improvement of any of the previous work art presented under Related Work.

The text of this article has been published in the following Conference Paper: *A Hybrid Data-Differencing and Compression Algorithm for the Automotive Industry* [131].

7.1 Why Data Differencing

The exponential increase in data, also known as big data, within the last decade has made once-popular data compression unable to fulfill its basic tasks. Compressing big data to achieve a workable or more feasible form for easier storage or transfer is now one of the challenges of the century. New technologies are needed to address it, and delta encoding seems to be one of them.

Most vehicle OEMs (Original Equipment Manufacturers) issue software updates periodically for a variety of reasons other than bug fixes. For instance, quality patches are regularly added to improve overall performance, ranging from updating the infotainment device(s) for a better user experience inside the vehicle, to security-related updates, such as air-bag deployments or gas -consumption check-up software.

These software updates are must-haves if vehicle computers—and, by default, automobiles—are meant to function at their expected qualities.

Few publicly available solutions for delta encoding and binary differencing are worth mentioning. The secure delta binary-differencing engines —developed by AgerSoftware[1] in collaboration with NetLUP Xtreme Technologies— and XtremeDELTA [5]—are two of them.

Among other binary differencing engines [51], there is an open-source solution provided by xDelta.org, although it seems to be part of an abandoned project today [6]. It employs the vcdiff format in RFC (3284) [11], briefly described in Section 7.2. The last project entry for the engine is dated 23 April 2016.

Our method is an innovative delta-encoding algorithm that embodies data compression as well. We call it Keops. In contrast with other delta algorithms, Keops creates binary-differencing fileswith the main advantage that it outputs a compressed stream even from an initial phase. This is achieved by using a compression distance for comparing the files [57].

7.2 Related Work

In recent years, different industries began using delta-encoding techniques on a large scale, from genome-information data storing, indexing and retrieving to source-code repositories for the automotive, gaming industries or even executable compression[56].

A binary code update based on binary differencing is also used in the automotive industry.

After carefully reviewing the most important articles, solutions and ideas, we found that the majority of the publicly available solutions do not address the fact that regardless of how good the internally deployed delta algorithm is, there will always be some sort of redundancy the delta algorithm is not addressing.

This is simply because by design, a delta algorithm is not a data-compression algorithm. It is a de-duplication algorithm at the best of its abilities.

7.3 Keops Algorithm

Our innovative Keops algorithm derives from the ubiquitous LZ77 [29] data-compression algorithm and uses it as a preferred compression method internally. LZ77 achieves compression by splitting a stream of data to be compressed, also known as an input stream, into two portions; the data are divided by using a current processing pointer called the current or compression pointer *cp*. The two sections are called LZ77 History and LZ77 Look Ahead, respectively (Fig. 6.1).

7.3.1 Delta File

Inside Keops, the LZ77 encoder acts the same within 'source' and 'target' windows, but unlike *vcdiff*, it instructs the decoder to reconstruct the content of the 'target' using the triplets $< d, l, c >$. Thus, it achieves compression in the binary delta file. The differences from the original LZ77 structure are that LZ77 History consists entirely of the old version of the file and the LZ77 Look Ahead is entirely made up of the newer version.

The delta file created by Keops is a sum of all the LZ77 compression operations applied to the LZ77 Look Ahead buffer—in a finite number of steps and various history-buffer combinations. The way the steps are controlled and how the two buffers are combined will be further explained.

7.3.2 Strategies for Buffer Pairing

LZ77 as applied in Keops is not a distance in the true sense of the word. It is not symmetrical, it is not null when the History buffer and the Look Ahead buffer are identical and it does not necessarily satisfy the triangle inequality. It is, however, greater than zero, disregarding the considered buffers. Consequently, we speak about a generalized distance.

In the following sections, we present three strategies for pairing the buffers. They are designed to optimize either the compression time or the compression rate or to balance them.

One-to-One Strategy (Time Optimized)

When a set of changes is designed to update or improve a software or its associated data file, it is usually called a patch. Called bug fixes or simply fixes, patches are usually designed to improve the functionality of a program or fix a coding flaw.

Brute-Force Strategy (Rate Optimized)

When differences between the old and the new file are numerous, the one-to-one strategy cannot offer a good compression anymore. Consider the case in Fig. 6.5, where in the new file, data are mostly new, but data from the old file are still present. It should be noted that because of code additions, modifications or substitutions, the one-to-one mapping is no longer preserved, since in this situation, the files are out of sync.

Flexy Strategy

Within updated code or data which may count numerous differences, the most similar buffer of the source is usually in the vicinity of the considered target buffer. This is explained by the fact that the de-synchronization is produced by both removing and adding code blocks; thus, the blocks' shift is not accumulated.

7.4 Experimental Results

We tested the Keops binary delta encoder on five types of data. The first test series comprised the minGW compiler binaries for Windows platform (*mingw*). We created binary delta files between versions 4.4 and 4.5 using various sizes for the History and Look Ahead buffers.

The second type consisted of formatted text files representing software performance logging files, or trace files, combined together into a single file. For the third test package, we chose two binary images from a collection of ECU binary images specifically used in one of our automotive projects. *Replay* was the fourth test package, which contains a specific media-related Windows software. We used real instances of this software: versions 31.2, 31.4 and 31.5.

For the last test package, we chose *silezia* corpus files, a well-known collection of English text and binary data files commonly used in data compression tests [35].

Three out of five test packages—*ALog*, $swfk$ and *Replay*—are specific to the automotive environment.

The test packages were chosen to be representative of various or homogeneous data, low- and high-redundancy files and structured and binary files. The tests were done for five buffer sizes: 2, 4, 8, 16 and 32 MB. We used equal History and Look-Ahead buffers. All tests were run on a Windows 10 PC running on an Intel i3-4130 CPU at 3.40GHz with 8 GB of RAM.

We tested Keops using the three strategies presented in Section 7.2: one to one, brute force and Flexy. For each of them, we recorded the delta rate, the encoding and decoding time and the memory requirements. The delta rate was the compression rate in percentage, expressed as the ratio of the sizes of the targets after the Keops compressions and before them.

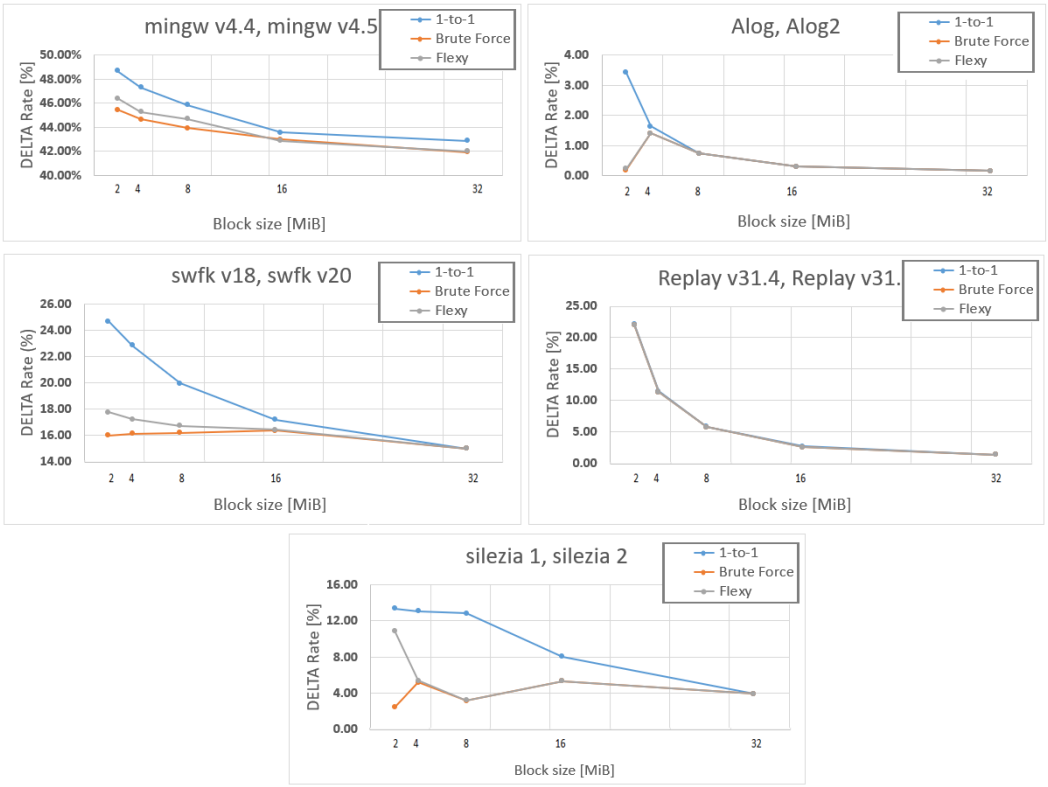

Fig. 7.1 Delta rate vs. block size.

For the Flexy strategy, the search of the most similar blocks in the source was conducted up and down with K positions relative to the current block in the target file, where K had a value of four by default. We chose this range after having observed the distribution of the gap of similar blocks in our experimental data (Fig. **??**). The distribution tended to be concentrated around zero, disregarding the file type. For *Alog* and *silezia*, where the newer versions were obtained by using removals, the distribution was bi-modal with a mode gap of two to six blocks, respectively.

7.4.1 Compression Rate

Concerning the strategies, the plots in Fig. 7.1 show that brute force is generally the best if the goal is a good compression. However, there are exceptions, e.g., for the *Replay* package, for which all strategies gave the same delta rate (the plots were superposed) or for the *Alog* package, for which brute force and Flexy behaved identically.

It should be noted that delta-rate curves are convergent as block sizes increase. Thus, if 32 MiB blocks are used, one should choose the one-to-one strategy, which is time optimized. There is no reason to do any block search and introduce delays as long as the compression is the same.

Generally, the difference between the delta rate and ZIP rate grew when large history buffers were used by Keops. For example, *Replay* shows an improvement in the delta rate from 22.15 to 1.33% as compared to a 75.78% ZIP rate.

We included the ZIP compression rate results of the new version (target file) in the tables as well since many software applications, even today, do not have software-update capabilities at the binary-differencing level. They simply allow users to download (possibly) a new compressed version of their application. There are many cases in which users download a ZIP file with an installer that handles a new version, i.e., uninstalls old versions and installs a newer version of the same software.

Concerning the strategies, the plots in Fig. 7.1 clearly show that brute force is the best if the goal is a good compression. However, Delta-rate curves are convergent as the block size increases.

As the results show, the delta ratios of *Alog* and *silezia* are very high, surpassing, in some cases, 99%, i.e., the Delta size (the binary differencing file size) is less than 1% in size when compared to the Target size.

7.4.2 Encoding Time

As expected, the encoding time depends on the strategy. Brute force and Flexy, which included a search for similar blocks, demanded longer encoding times. On average, brute force had four-times longer durations, while the Flexy strategy was only 2.5 more time consuming. Obviously, the highest encoding times correspond to encoding with small blocks.

7.4.3 Decoding Time

While the encoding times go from 3.75 s to 1253 s, the decoding is much faster. Depending on block size, file type and strategy, it may have values between 0.18 s and 10.46 s. Moreover, we noticed that usually, it goes lower once the block size increases, except with the 32 MiB blocks. This could be related to the fact that larger buffers tend to allow for more distant matches, and once matches (or common sub-strings) are further away, there is a higher probability cache misses and page faults will occur.

7.5 Conclusions

Keops is the best fit for low- to mid-memory environments that need to operate software updates at high speeds, since we provide, depending on the chosen strategy, the best possible solution for compression ratios and, ultimately, binary delta sizes.

Chapter 8

Conclusions

This thesis presents my research on data compression algorithms, designed for lossless compression in low resources environments and therefore suitable for the automotive industry. The research was conducted at two levels: coding algorithm improvements and design of a complete application for Big data compression by data differencing. In the first part of the thesis, starting from state of the art Entropy Coding algorithms, such as Static and Canonical Huffman or Static Arithmetic Compression, we proposed several original solutions for fast decoding of canonic Huffman codes and a fast solution for arithmetic encoding, called Quasi Static Arithmetic Encoder. In the second part of the thesis, we proposed a new method for data differencing that we believe it may be suitable for low resource environment. While designed for decompression speed as the first option, we believe this could be a great fit for the automotive industry, yielding faster decompression results than first two compared commercial products in the market.

The proposed solutions have been thoroughly examined by theoretical analysis as well as using dedicates software created to validate and produce the expected results. Each proposed solution was accompanied by a piece of software and exhaustive tests have been performed with such software. Different test cases with various test files were used.

The original contributions of this thesis are listed in the next section in details.

8.1 Original Contributions

The original contributions developed in this thesis, referred to the published research papers, are as follows:

- Backtrace of the undocumented algorithm RoLZ, from notes and descriptions of LZP and RKive algorithms; improvement of RoLZ algorithm over the original version suggested by Charles Bloom and Mark Taylor (Section 2.3, Conference Paper [128]).

- A novel canonical Huffman decoding method, with the ability to process 8 to 12 symbols in a single decoding cycle. It comprises new methods for creating the classic Huffman tree and for storage of code lengths. It is an original algorithm not an improvement of any of the previous work art presented (Section 3.2, Conference Paper [131]).

- An improved algorithm for fast arithmetic coding called Quasi-Static Arithmetic encoder. The algorithm was designed such to improve the encoding speed by minimizing the number of divisions needed to process a symbol. (Section 4.3, Conference Paper [128]).

- A binary differencing algorithm based on an NCD like generalized distance, designed for low resources environments, suitable for IoT or automotive industries. (Section 6.3, Conference Paper [130]).

- An original algorithm for binary data differencing for faster decoding in low resources environments. It includes three working strategies to accommodate different use cases in IoT and automotive industries (Section 7.3, Conference Paper [127]).

8.2 List of publications

8.2.1 Journal papers

1. S. Belu and D. Coltuc, "A Hybrid Data-Differencing and Compression Algorithm for the Automotive Industry", in *Entropy*, (IF 2.524, Q2), 24(5), 574, 2022, https://doi.org/10.3390/e24050574, WOS:000803286100001.

8.2.2 Conference Papers

1. S. Belu and D. Coltuc, "Fast Huffman Canonic Decoder". In *IEEE 14th International Conference on Communications COMM22*, Bucuresti, Romania, 2022.

2. S. Belu and D. Coltuc, "An innovative algorithm for data differencing". In *IEEE International Symposium on Electronics and Telecommunications (ISETC)*, Timisoara, Romania, 2020. DOI: 10.1109/ISETC50328.2020.9301053.WOS: 000612681000078.

3. S. Belu and D. Coltuc, "The Reduced Offset LZ Data Compression Algorithm". In *IEEE International Symposium on Signals, Circuits and Systems (ISSCS)*, Iasi, Romania, 2019. DOI: 10.1109/ISSCS.2019.8801741. WOS:000503459500013.

4. S. Belu and D. Coltuc, "The Anatomy of a Quasi-Static Arithmetic Encoder". In *IEEE 12th International Conference on Communications COMM18*, Bucuresti, Romania, 2018. DOI: 10.1109/ICComm.2018.8484263.WOS: 000449526000029.

8.2.3 Doctoral Research Reports

1. Doctoral Research Report No. 1/2018: "The Anatomy of a Quasi-Static Arithmetic Encoder", Conference Paper no. 4.

2. Doctoral Research Report No. 2/2018: "RoLZ - The Reduced Ofifsef LZ Data Compression Algorithm, Conference Paper no. 3.

3. Doctoral Research Report No. 3/2019: "ACHD - Advanced Canoniucal Huffman Decoder", Conference Paper no. 1.

4. Doctoral Research Report No. 4/2019: "An Innovative Data Differencing Algorithm", Conference Paper no. 2.

5. Doctoral Research Report No. 5/2020: "Analysis and Interpretation of Test Results for KEOPS, an Innovative Data Differencing Algorithm", 28 May 2020

8.3 Future Work

We identified several future research directions for the improvement of our data compression techniques and algorithms designed for the automotive industry. We shall continue on the path of further improving the stand alone data compression algorithms, both in speed and in compression rates.
Further and more sophisticated Software Update methods to be addressed with new data structures such as cache-aware hash algorithms and state of the art hash chaining mechanisms to allow for faster processing of Big Data at reasonable memory usage levels.
Another special interest will be given to heuristic methods used in our data compression implementations, specific streaming modes and more sophisticated tune up in the algorithms core, to further improve security and at the same time, compression rate by better rearranging the output streams.

References

[1] SecureDELTA SDK. Available online: https://agersoftware.com/securedelta_sdk.html (accessed on 13 January 2022).

[2] Guttman, L., A basis for scaling qualitative data, American Sociological Review, 9 (2), 1944, pp. 139–150, doi:10.2307/2086306, JSTOR 2086306.

[3] Jones, D.W, Application of Splay Trees to Data Compression, Communication of ACM, 18, 1988, pp. 996-1007

[4] Sleator, D.D. and Tarjan, R.E., Self-adjusting binary search tree, Communication of ACM, 32, 1985, pp. 652-686

[5] SecureDELTA Application with XtremeDELTA Engine. Available online: https://agersoftware.com/securedelta_app.html (accessed on 13 January 2022).

[6] xdelta org. Available online: http://xdelta.org/ (accessed on 13 January 2022).

[7] Richard C. Pasco, "Source coding algorithms for fast data compression", Stanford, CA 1976

[8] J. S. Vitter, Algorithm 673: Dynamic Huffman Coding, ACM Transactions on Mathematical Software, 15(2), June 1989, pp 158–167.

[9] Donald E. Knuth, "Dynamic Huffman Coding", Journal of Algorithm, 6(2), 1985, pp 163–180.

[10] Robert G. Gallager, "Variations on a Theme by Huffman", IEEE TRANSACTIONS ON INFORMATION THEORY, Vol. IT - 24, No. 6, Nov 1978

[11] RFC 3284—The VCDIFF Generic Differencing and Compression Data Format. Available online: https://tools.ietf.org/html/rfc3284 (accessed on 6 April 2022).

[12] Westerberg, E. *Efficient Delta Based Updates for Read-Only Filesystem Images: An Applied Study in How to Efficiently Update the Software of an ECU*; Degree Project in Computer Science and Engineering, KTH Royal Institute of Technology School of Electrical Engineering and Computer Science, Stockholm, Sweden, 2021.

[13] Falleri, J.R.; Morandat, F.; Blanc, X.; Martinez, M.; Monperrus, M. Fine-grained and accurate source code differencing. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, Vsters, Sweden, 15–19 September 2014.

[14] Gerardo, C.; Luigi, C.; Massimiliano, P. *Identifying Changed Source Code Lines from Version Repositories*; RCOST—Research Centre on Software Technology Department of Engineering—University of Sannio Viale, Viale Traiano - 82100, Benevento, Italy, 2007.

[15] Zimmermann, T.; Weisgerber, P.; Diehl, S.; Zeller, A. Mining version histories to guide software changes. In Proceedings of the 26th International Conference on Software Engineering, Washington, DC, USA, 23–28 May 2004; pp. 563–572.

[16] Ying, A.T.T.; Murphy, G.C.; Ng, R.; Chu-Carroll, M. C. Predicting source code changes by mining revision history. *IEEE Tr. Softw. Eng.* **2004**, *30*, 574–586.

[17] Li, B.; Tong, C.; Gao, Y.; Dong, W. S2: A Small Delta and Small Memory Differencing Algorithm for Reprogramming Resource-constrained IoT Devices. In Proceedings of the IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Vancouver, BC, Canada, 10 May 2021.

[18] Christley, S.; Lu, Y.; Li, C.; Xie, X. Human genomes as email attachments. *Bioinformatics* **2009**, *2*, 274–275.

[19] Pavlichin, D.S; Tsachy, W. The Human Genome Contracts Again *Bioinformatics* **2013**, *29*, 2199–2022.

[20] Ochoa, I.; Mikel H.; Tsachy W. iDoComp: A compression scheme for assembled genomes. *Bioinformatics* **2015**, *31*, 626–633.

[21] Kuruppu, S.; Beresford-Smith, B.; Conway, T.; Zobel, J. Iterative dictionary construction for compression of large DNA data sets. *IEEE/AMC Trans. Comput. Biol. Bioinform.* **2010**, *1*, 137–149.

[22] Deorowicz, S.; Grabowski, S. Robust relative compression of genomes with random access, *Bioinformatics* **2011**, *21*, 2979–2986.

[23] Kuruppu, S.; Puglisi, S.J.; Zobel, J. Optimized relative lempel-ziv compression of genomes. In Proceedings of the Thirty-Fourth Australasian Computer Science Conference, Perth, Australia, 17–20 January 2011.

[24] Pinho, A.J.; Diogo P.; Sara, P.G. GReEn: A tool for efficient compression of genome resequencing data. *Nucleic Acids Res.* **2012**, *40*, e27.

[25] Wang, C; Dabing Z. A novel compression tool for efficient storage of genome resequencing data. *Nucleic Acids Res.* **2011**, *39*, e45.

[26] Wandelt, S.; Ulf, L. FRESCO: Referential compression of highly similar sequences. *IEEE/ACM Trans. Comput. Biol. Bioinform (TCBB)* **2013**, *10*, 1275–1288.

[27] Brandon, M.C.; Wallace, D.C.; Baldi, P. Data structures and compression algorithms for genomic sequence data. *Bioinformatics* **2009**, *14*, 1731–1738.

[28] Chern, B.G.; Ochoa, I.; Manolakos, A.; No, A.; Venkat, K.; Weissman, T. Reference based genome compression. In Proceedings of the IEEE Information Theory Workshop (ITW), Visby, Sweden, 25–28 August 2012; pp. 427–431.

[29] Ziv, J.; Lempel, A. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. Inf. Theory* **1977**, *23*, 337–343.

[30] Yufei, T. Patricia Tries Lecture. KAIST. Available online: http://www.cse.cuhk.edu.hk/~taoyf/course/wst540/notes/lec10.pdf (accessed on 1 May 2013).

[31] Daelemans, W.; Bosh, A.V.D.; Antal, Weijters, T. IGTree: Using Trees for Compression and Classification. *Lazy Learn.* **1997**, 407–423

[32] Horspool, R.N. The Effect of Non-Greedy Parsing in Ziv-Lempel Compression Method, In Proceedings of the Data Compression Conference, Snowbird, UT, USA, 28–30 March 1995.

[33] Storer, J.A.; Szymanski, T.G. Data Compression via Textual Substitution. *J. ACM* **1982**, *29*, 928–951.

[34] Korn, D.G; MacDonald J; Mogul, J.C.; Vo, K.-P. The VCDIFF Generic Differencing and Compression Data Format. *RFC* **2002**, *3284*, 1–29.

[35] The Silesia Corpus. Available online: http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia (accessed on 10 January 2022).

[36] Zlib Compression Library. Available online: http://www.zlib.org/rfc1950.pdf (accessed on 13 January 2022).

[37] Chen, X.; Li, M.; Ma, B.; Tromp, J. DNACompress: fast and effective DNA sequence compression. *Bioinformatics* **2002**, *10*, 51–61.

[38] APPNOTE.TXT-ZIP File Format Specification. Available online: https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT(accessed on 13 January 2022).

[39] Constructing a Binary Difference File. Available online: https://agersoftware.com/docs/securedelta_app_v2.56/43Creatingabinarydiffdeltafile.html (accessed on 13 January 2022).

[40] Raghavan, S.; Rohana, R.; Leon, D.; Podgurski, A.; Augustine, V. Dex: A semantic-graph differencing tool for studying changes in large code bases. In Proceedings of the 20th IEEE International Conference on Software Maintenance, Chicago, IL, USA, 11 September 2004; pp. 188–197.

[41] Grumbach,S.; Tahi, F. A new challenge for compression Algorithms: Genetic sequences. *Inf. Process. Manag. Int. J.* **1994**, *6*, 875–886.

[42] Cao, M.D.; Dix, T.I.; Allison, L.; Mears, C. A simple statistical algorithm for biological sequence compression. In Proceedings of the IEEE Data Compression Conference (DCC'07), Snowbird, Utah, 27–29 March 2007.

[43] Deorowicz, S.; Grabowski, S. Data compression for sequencing data. *Algorithms Mol. Biol.*, **2013**, *8*, 1–13.

[44] Deorowicz, S.; Danek, A.; Grabowski, S. Genome compression: A novel approach for large collections. *Bioinformatics* **2013**, *29*, 2572–2578.

[45] Dotzler, G.; Michael, P. Move-optimized source code tree differencing. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, 3–7 September 2016; IEEE: Piscataway, NJ, USA 2016.

[46] Fluri, B.; Wursch, M.; PInzger, M.; Gall, H. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.* **2007**, *33*, 725–743.

[47] Canfora, G.; Luigi, C.; Massimiliano P. Ldiff: An enhanced line differencing tool. In Proceedings of the IEEE 31st International Conference on Software Engineering, Vancouver, BC, Canada, 16–24 May 2009; IEEE: Piscataway, NJ, USA 2009.

[48] Frick, V.; Grassauer, T.; Beck, F.; Pinzger, M. Generating accurate and compact edit scripts using tree differencing. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain, 23–29 September 2018; IEEE: Piscataway, NJ, USA 2018.

[49] Tsantalis, N.; Natalia N; Eleni S. Webdiff: A generic differencing service for software artifacts. In Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM), Williamsburg, VA, USA, 25–30 September 2011; IEEE: Piscataway, NJ, USA 2011.

[50] Maletic, J.I.; Michael L. C. Supporting source code difference analysis. In Proceedings of the 20th IEEE International Conference on Software Maintenance, Chicago, IL, USA, 11–14 September 2004; IEEE: Piscataway, NJ, USA, **2004**.

[51] Korn, D.G.; Vo, K.P. Engineering a Differencing and Compression Data Format. In Proceedings of the USENIX Annual Technical Conference, General Track, Berkeley, CA, USA, 10–15 June 2002.

[52] Nguyen, H.A.; Nguyen, T.T.; Nguyen, H.V.; Nguyen, T.N. Idiff: Interaction-based program differencing tool. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, 6–10 November 2011; IEEE: Piscataway, NJ, USA, 2011.

[53] Müller, K.; Bernhard R. User-driven adaptation of model differencing results. In *International Workshop on Comparison and Versioning of Software Models (CVSM'14)*; Köllen Druck+Verlag GmbH: Bonn, Germany, 2014.

[54] Onuma, Y.; Nozawa, M.; Terashima, Y.; Kiyohara, R. Improved software updating for automotive ECUs: Code compression. In Proceedings of the IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), Atlanta, Georgia, 10–14 June 2016.

[55] Ni, G.; Yan, Y.; Jiang, J.; Mei, J.; Chen, Z.; Long, J. Research on incremental updating. In Proceedings of the 2016 International Conference on Communications, Information Management and Network Security, Shanghai, China, 25–26 September 2016.

[56] Motta, G.; James G.; Samson C. Differential compression of executable code. In Proceedings of the Data Compression Conference (DCC'07), Snowbird, Utah, 27–29 March 2007.

[57] Belu, S.; Daniela C. An innovative algorithm for data differencing. In Proceedings of the 2020 International Symposium on Electronics and Telecommunications (ISETC), Timisoara, Romania, 5–6 November 2020.

[58] Deorowicz, S.; Agnieszka D.; Marcin N. GDC2: Compression of large collections of genomes. *Sci. Rep.* **2015**, *5*, 1–12.

[59] Kuruppu, S.; Simon J.P.; Justin Z. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In Proceedings of the International Symposium on String Processing and Information Retrieval, Berlin/Heidelberg, Germany, 13–15 October 2010.

[60] M. H. Mohammed, A. Dutta, T. Bose, S. Chadaram, and S. S. Mande, "DELIMINATE-a fast and efficient method for loss-less compression of genomic sequences," Bioinformatics, vol. 28, no. 19, pp. 2527–2529, 2012.

[61] Saha, S.; Sanguthevar, R. ERGC: An efficient referential genome compression algorithm. *Bioinformatics* **2015**, *31*, 3468–3475.

[62] Saha, S.; Sanguthevar, R. NRGC: A novel referential genome compression algorithm. *Bioinformatics* **2016**, *32*, 3405–3412.

[63] Liu, Y.; Peng, H.; Wong, L.; Li, J. High-speed and high-ratio referential genome compression. *Bioinformatics* **2017**, *33*, 3364–3372.

[64] Lempel Ziv Markov Algorithm. Available online: https://www.7-zip.org/sdk.html (accessed on 13 January 2022).

[65] Mary Shanthi Rani, "A New Referential Method for Compressing Genomes",International Journal of Computational Bioinformatics and In Silico Modeling, Vol. 4, No. 1 (2015): 592-596

[66] Alves F, Cogo V, Wandelt S, Leser U, Bessani A. On-Demand Indexing for Referential Compression of DNA Sequences. PLoS One. 2015 Jul 6;10(7):e0132460. doi: 10.1371/journal.pone.0132460. PMID: 26146838; PMCID: PMC4493149.

[67] Yao H, Ji Y, Li K, Liu S, He J, Wang R. HRCM: An Efficient Hybrid Referential Compression Method for Genomic Big Data. Biomed Res Int. 2019 Nov 16;2019:3108950. doi: 10.1155/2019/3108950. PMID: 31915686; PMCID: PMC6930768.

[68] Chern, B.G.; Ochoa, I.; Manolakos, A.; No, A.; Venkat, K.; Weissman, T. Reference based genome compression. In Proceedings of the IEEE Information Theory Workshop (ITW), Visby, Sweden, 25–28 August 2012; pp. 427–431.

[69] Kraft, Leon G. (1949), A device for quantizing, grouping, and coding amplitude modulated pulses, Cambridge, MA: MS Thesis, Electrical Engineering Department, Massachusetts Institute of Technology, hdl:1721.1/12390.

[70] McMillan, Brockway (1956), "Two inequalities implied by unique decipherability", IEEE Trans. Inf. Theory, 2 (4): 115–116, doi:10.1109/TIT.1956.1056818.

[71] Diffie, Whitfield; Hellman, Martin E. (November 1976). "New Directions in Cryptography" (PDF). IEEE Transactions on Information Theory. 22 (6): 644–654. CiteSeerX 10.1.1.37.9720. doi:10.1109/TIT.1976.1055638. Archived (PDF) from the original on 2014-11-29.

[72] Fritz, M.H.Y.; Leinonen, R.; Cochrane, G.; Birney, E. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Res.* **2011**, *21*, 734–740.

[73] Saha, S.; Sanguthevar, R. ERGC: An efficient referential genome compression algorithm. *Bioinformatics* **2015**, *31*, 3468–3475.

[74] Jarek Duda, "Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding", Center for Science of Information, Purdue University, W. Lafayette, IN 47907, 2013

[75] Garg, S 32/64 BIT RANGE CODING AND RITHMETIC CODING, https://sachingarg.com/compression/entropy$_c$*oding*/,*March*1979

[76] Robert F. Rice , Some Practical Universal Noiseless Coding Techniques, Jet Propulsion Laboratory, JPL Publication 79—22, March 1979

[77] Golomb, Solomon W., Run-length encodings, IEEE Transactions on Information Theory, IT–12(3), 1966, pp. 399–401

[78] J. H. Witten, R. M. Neal and J. G. Cleary, ARITHMETIC CODING FOR DATA COMPRESSION, Communication of the ACM (1987), 30, 1987, 520–540

[79] M. Nelson,Data Compression With Arithmetic Coding,Dr. Dobbs Journal,Nov 4, 2014

[80] P. G. Howard and J. S. Vitter,Practical Implementation of Arithmetic Coding,Kluwer Academic Publishers,1992,85–112

[81] D. A. Huffman, “A method for the construction of minimum-redundancy codes,” Proc. of the IRE, Vol. 40, pp. 1098-1101, Sep. 1952.

[82] D. E. Knuth, "The Art of Computer Programming", Vol. 3, Sorting and Searching, Addison Wesley, Reading, MA, 1973.

[83] R.M. Fano. "The transmission of information", Technical Report 65, Research Laboratory of Electronics, M.I.T., Cambridge, Mass., 1949.

[84] C. E. Shannon. "A mathematical theory of communication", Bell System Technical Journal, 27:379–423, 623–656, July, October 1948.

[85] Alistair Moffat, 2019, "Huffman Coding", ACM Comput. Surv. 52, 4, Article 85, August 2019

[86] Sieminskly, A. “Fast Decoding of the Huffman codes.” Information Processing Letters 26 (1987/88) pp. 237-241 11 January 198

[87] Tanaka, H., “Data structure of Huffman codes and its application to efficient encoding and decoding, “ IEEE Trans. Inf. Theory 33, 1 (Jan 1987), 154-156

[88] Renato Pajarola “Fast Prefix Code Processing”, IEEE ITCC Conference, pages 206–211, 2003.

[89] Alistair Moffat and Andrew Turpin, “On the Implementation of Minimum Redundancy Prefix Codes”, IEEE Transactions on Communications, Vol. 45, No. 10, October 1997

[90] Daniel S. Hirschberg and Debra A. Lelewer, “Efficient Decoding of Prefix Codes”, Communications of the ACM, Vol.33, pp. 449-459, 1990

[91] Chung Wang, Yuan-Rung Yang, Chun-Liang Lee, Hung-Yi Chang “A memory-efficient Huffman decoding algorithm” Published in: 19th International Conference on Advanced Information Networking and Applications (AINA’05) Volume 1.

[92] Habib, A., Rahman, M.S. “Balancing decoding speed and memory usage for Huffman codes using quaternary tree”. Appl Inform 4, 5 (2017).

[93] Swapna R. and Ramesh P., “Design and Implementation of Huffman Decoder for Text data Compression”, International Journal of Current Engineering and Technology, Vol.5, No.3 (June 2015).

[94] Yann Collet, Huff-0,
https://github.com/Cyan4973/FiniteStateEntropy
Accessed on: May 20, 2022.

[95] Yann Collet, zHuff,
https://fastcompression.blogspot.com/p/zhuff.html
Accessed on: May 20, 2022.

[96] Yann Collet, Zstd,
https://facebook.github.io/zstd/, Accessed on: May 20, 2022.

[97] National Library of Medicine, National Center for Biotechnology Information, Bethesda, MD, USA, Available online at: https://www.ncbi.nlm.nih.gov/genomes/VirusVariation/Database/nph-select.cgi, Accessed on May 20, 2022.

[98] Edward R. Fiala and Daniel H. Greene, "Data Compression with Finite Windows", Communications of the ACM, Vol. 32, Issue 4, April 1989, pp. 490-505.

[99] Yann Collet, Range-0,
http://sd-1.archive-host.com/membres/up/182754578/Range0v07.zip
Accessed on: May 20, 2022.

[100] Yann Collet, Huff-X, https://fastcompression.blogspot.com/p/huff0-range0-entropy-coders.html, Accessed on: May 20, 2022.

[101] G. N. N. Martin, "Range encoding: An algorithm for removing redundancy from a digitized message", Video & Data Recording Conference, Southampton, UK, July 24–27, 1979.

[102] Jean-loup Gailly, Mark Adler, “Zlib library”, Available online at:
https://www.zlib.net/, Accessed on: May 20, 2022.

[103] Wikipedia. Adler-32. http://en.wikipedia.org/wiki/Adler-32

[104] J. G. Fletcher. An arithmetic checksum for serial transmissions. IEEE Transactions on Communications, COM30(1):247–252, Jan. 1982.

[105] Jarek Duda, "Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding", Center for Science of Information, Purdue University, W. Lafayette, IN 47907, 2013

[106] Garg, S 32/64 BIT RANGE CODING AND RITHMETIC CODING https://sachingarg.com/compression/entropy$_c oding/March$1979

[107] Robert F. Rice , Some Practical Universal Noiseless Coding Techniques, Jet Propulsion Laboratory, JPL Publication 79—22, March 1979

[108] Golomb, Solomon W., Run-length encodings, IEEE Transactions on Information Theory, IT–12(3), 1966, pp. 399–401

[109] J. H. Witten, R. M. Neal and J. G. Cleary, ARITHMETIC CODING FOR DATA COMPRESSION, Communication of the ACM (1987), 30, 1987, 520–540

[110] M. Nelson,Data Compression With Arithmetic Coding,Dr. Dobbs Journal,Nov 4, 2014

[111] P. G. Howard and J. S. Vitter,Practical Implementation of Arithmetic Coding,Kluwer Academic Publishers,1992,85–112

[112] Malcolm Taylor, ”RKIVE file archiver”, http://files.mpoli.fi/unpacked/software/dos/utils/diskfile/rkv190b1.zip/

[113] Bloom, Charles. ”LZP: A New Data Compression Algorithm.” Data Compression Conference. 1996.

[114] Kolmogorov, A. (1968). ”Logical basis for information theory and probability theory”. IEEE Transactions of Information Theory, IT-14:662–664, 1968

[115] Ziv, Jacob; Lempel, Abraham (May 1977). ”A Universal Algorithm for Sequential Data Compression”. IEEE Trans. on Info. Theory, 23:337–343, 1977.

[116] Jacob Ziv, Abraham Lempel, Compression of Individual Sequences via Variable-Rate Coding, IEEE Transactions on Information Theory, 24, 5, pp. 530-536, CiteSeerX 10.1.1.14.2892, doi:10.1109/TIT.1978.1055934, 1978

[117] Storer, James A.; Szymanski, Thomas G. (October 1982). ”Data Compression via Textual Substitution”. doi:10.1145/322344.322346.

[118] James A. Storer, 1992, ”Method and apparatus for data compression”, US Patent US5379036A, United States

[119] Robert K. Jung, ”Data compression/decompression method and apparatus”

[120] W.J.Cody et al., Aug. 1984, IEEE standards 754 and 854 for Floating-Point Arithmetic, IEEE Magazine MICRO, 84 – 100, IEEE

[121] Morrison, Donald R., title=PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric, journal=Journal of the Association for Computing Machinery, volume=15, number=4, year=October 1968

[122] Okumura, Haruhiko, History of Data Compression in Japan, edu.mie-u.ac.jp, https://oku.edu.mie-u.ac.jp/ okumura/compression/history.html, 1998

[123] R.Fiala, Edward; H. Greene, Daniel, Data Compression with Finite Windows, Communications of the ACM, 32, 4, April 1989, pp. 490-505, https://doi.org/10.1145/63334.63341

[124] Kak S., Generalized unary coding, Circuits Systems and Signal Processing, 35, 4, 1419 – 1426, doi:10.1007/s00034-015-0120-7, S2CID 27902257, 2015

[125] Golomb S. W.; Gordon Basil; Welch L. R., Comma-Free Codes, Canadian Journal of Mathematics, 10, 2, pp. 202 - 209, doi:10.4153/CJM-1958-023-9, 1958

[126] M. Dipperstein, “Arithmetic Code Discussion and Implementation”, http://michael.dipperstein.com/arithmetic/index.html, 23rd November, 2014.

[127] Sabin B. and Daniela C., A Hybrid Data-Differencing and Compression Algorithm for the Automotive Industry, Entropy, 24, 5, 2022

[128] Sabin B. and Daniela C., The Anatomy of a Quasi-Static Arithmetic Encoder,2018 International Conference on Communications (COMM),2018

[129] Sabin B. and Daniela C., RoLZ - The Reduced Offset LZ Data Compression Algorithm,019 International Symposium on Signals, Circuits and Systems (ISSCS),2019

[130] Sabin B. and Daniela C., An innovative algorithm for data differencing, 2020 International Symposium on Electronics and Telecommunications (ISETC),2020

[131] Sabin B. and Daniela C.,Fast Canonical Huffman Decoder, IEEE COMMS 2022,2022

[132] I. Witten, T. Bell and John Cleary, MODELING FOR TEXT COMPRESSION, ACM Computing Surveys, 21, 4, 557–591, December 1989, University of Calgary, CA

[133] Timothy Bell, John Cleary and Ian Witten, Text Compression, 0-13-911991-4, 1st edition, February 1, 1990, Prentice-Hall, Englewood, USA

[134] Alistair Moffat and Neil Sharman, AN EMPIRICAL EVALUATION OF CODING METHODS FOR MULTI-SYMBOL ALPHABETS, 0-13-911991-4, 30, 6, 791-804, Information Processing Management, 1994

[135] Calude, C.S. (1996). "Algorithmic information theory: Open problems"

[136] “Delta Algorithms: An Empirical Analysis”, James J. Hunt University of Karlsruhe, Karlsruhe, Germany and Kiem-Phong Vo ATT Laboratories, Florham Park, NJ, USA and Walter F. Tichy University of Karlsruhe, Karlsruhe, Germany.

[137] David G. Korn and Kiem-Phong Vo, “Engineering a Differencing and Compression Data Format” ATT Laboratories – Research 180 Park Avenue, Florham Park, NJ 07932, U.S.A. dgk,kpv@research.att.com.

[138] James W. Hunt and M.D. McIllroy, “An algorithm for differential file comparison” Technical Report Computing Science Technical Report 41, Bell Laboratories, June 1976.

[139] Webb Miller and Eugene W. Meyers, “A file comparison program. Software|Practice and Experience” 15(11):1025, 1039, November 1985.

[140] C.H. Bennett, P. Gacs, M. Li, P.M.B. Vitanyi, and W. Zurek, Information Distance, IEEE Trans. Inform. Theory, IT-44:4(1998) 1407–1423

[141] M. Li, X. Chen, X. Li, B. Ma, P.M.B. Vitanyi, "The similarity metric", IEEE Trans. Inform. Th., 50:12(2004), 3250–3264". IEEE Transactions on Information Theory. 50 (12): 3250–3264. doi:10.1109/TIT.2004.83810

[142] Williams R.N., Adaptive Data Compression, Kluwer Academic Publishers, 1991

[143] Fraenkel, Aviezri S.; Klein, Shmuel T., Robust universal complete codes for transmission and compression, Discrete Applied Mathematics, Markov Chains, 64, 1996, ISSN=0166-218X, CiteSeerX 10.1.1.37.3064, 10.1016/0166-218X(93)00116-H

[144] Markov, Andreĭ Andreevich, Extension of the limit theorems of probability theory to a sum of variables connected in a chain, Dynamic Probabilistic Systems, reprinted in Appendix B of: R. Howard, Markov Chains, 1, 1971, John Wiley and Sons

[145] Elias, Peter, Universal codeword sets and representations of the integers, IEEE Transactions on Information Theory, Markov Chains, 21, 2, 194-203, 1975, 10.1109/tit.1975.1055349

[146] Fraenkel, Aviezri S. and Klein, Shmuel T., Robust universal complete codes for transmission and compression, Discrete Applied Mathematics, 64, 1, 31 – 55, 1996, 10.1109/tit.1975.1055349, 0166-218X