Master Thesis
Software Engineering
Thesis no: MSE-2010-21
May 2010

An Introduction and Evaluation of a Lossless Fuzzy Binary AND/OR Compressor

Philip Baback Alipour and Muhammad Ali

School of Computing
Blekinge Institute of
Technology Box 520
SE – 372 25 Ronneby
Sweden

Contact Information:
Author(s): Philip B. Alipour
Email: philipbaback_orbsix@msn.com

Muhammad Ali
Email: moh.ali@live.se

University advisor(s): Dr. Niklas Lavesson
School of Computing, BTH

School of Computing
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/com
Phone : +46 457 38 50 00
Fax : +46 457 271 25

ABSTRACT

Context. The currently known compression coding algorithms on x86 machines do not include techniques for generating fixed values of compression ratios as reliable as possible for maximum lossless data compressions (LDCs). However, a '4-dimensional self-embedded bit flag model' is evaluated to serve multidimensional LDCs with fixed value generations, contrasting the popular methods used in probabilistic LDCs, such as Shannon entropy. In context, we use this model to implement a new way of predictably-fixed compression values compared to other algorithms.

Objectives. We investigate a new type of logic in design and concept, governing issues directed to one LDC algorithm. The main objectives were: the applicability of FBAR logic on data structures for a new LDC; design and requirements of the algorithm parallel to limitations versus strong points of the implementation; tackling the limitations mostly observed on x86 compilers compared to the simulated version of quantum compilers; the evaluation of the algorithm based on output data, once logic is applied; investigating whether the algorithm is lossless or not. Finally, comparing its performance with other LDCs used today. We introduce this logic in its respective model via implementation and simulation for general use on x86 machines, and its future use on quantum computers.

Methods. We implement the FBAR algorithm in form of a prototype using certain software development environments and program code samples in C language. The prototype presents the encoding/decoding techniques for low-level compressions. From there, a high-level compression is conducted. It compressed data as expected by loading a document sample using a memory grid file which is a portable file containing single bit flags in 65,536 rows or addresses. The translation of addresses for original characters is given in a translation table with a static size ≈ 8MB, for any amount of input data. By the program's interpreter, once flags are compared with the compression result, we begin decompression. The decompression uses these flags to manipulate the compressed data to reconstruct new data identical to the contents once loaded to the program i.e. the original document.

Results. We covered spatial results as minimally as 37.5% data compression, and a maximum of roughly 87.5% on x86 machines. The algorithm's compression and decompression simulation grade, performed bitrates averaging 475 kBps, to encode and decode data, respectively. Based on our analysis, we ranked the topmost algorithms used today, to have the lowest ranks in memory usage compared to FBAR, such as WinRK, on their dictionary coders. We also deduced by result that, FBAR performance at any level is uniformly fixed due to its logic and design implementation. We further used a nonparametric statistical test to compare our algorithm with other LDCs with mean rank sums, showing a significant difference between results. Results were realistically 50% for the fuzzy binary (FBAR) version, proving double-efficiency of 16 bit transmission via 8 quantum bits, and roughly 87.5% for the strongly fuzzy quantum (FQAR) type on x86 machines, which gives double-efficient hypothetical values ≥ 87.5% compression.

Conclusions. We observed that the current version of FBAR compresses data with fixed compression ratios, where other compressors do not. Almost every lossless compressor uses probabilistic Shannon entropy as its 'logic base' in conducting LDCs. FBAR achieves higher space savings, above 50% as estimated, simulated and discussed in theory from its quantum state protocol. The LDD simulation, allowed us to study FBAR products from our experiment, yielding a double-efficient data compression > 87.5% or a negentropy < 0 bits/byte. We thus conclude that, our algorithm contains predictable values for every double-character input. The predictable fixed value, allows a user to know how much physical space is available within a reasonable time, before and after compression. This confidence in predictability makes FBAR a reliable version compared to the probabilistic LDCs available on the market.

Keywords: Fuzzy Binary AND/OR, data compression/ decompression, pairwise bits, double efficiency.

CONTENTS

Paper

Appendix A

Appendix B

ACKNOWLEGMENTS

We would like to thank our supervisor Dr. Niklas Lavesson for his support, guidance and insightful ideas throughout this work. This thesis would not have been possible without his valuable time and input. We are thankful for the constructive meetings and discussions that we had, which were a great inspiration and helped us to focus on the subject and most important issues.

HMT FORMAT

The thesis is structured according to the ‘Hybrid Master Thesis’ (HMT) format, which was proposed in the summer 2007 by members of BTH. The idea of the HMT format, is to have a hybrid form between an IEEE/ACM paper and a traditional master thesis. In the current version, we combined two disciplines of Computer Science and Software Engineering relevant to the characteristics of the current topic which mostly rely on Computer Science experimentations, thus entailing aspects of Software Engineering, such as risk analyses and software marketing issues for the topic’s technique when demonstrated under a set of directive criteria. The directive criteria anchors into the implementation of hypotheses and thereby their conduction relative to the performance of technology, introduced earlier, in form of a thesis proposal. One of the reasons behind the HMT format is to increase the number of theses that can be published as papers. A further reason is to help students focus their writing and express themselves clearly.

According to the HMT format, the document should be divided into two major parts. The former part (Part A) follows the IEEE/ACM format and focuses on the most relevant areas of the thesis project. It should comprise a maximum of 15 pages. The latter part (Part B) consists of a number of Appendices that cover in detail different aspects, such as methodology, validation and experiment. Part B should typically have a length of 15-40 pages, whereas it can grow upwards if necessary.

An Introduction and Evaluation of a Lossless Fuzzy Binary AND/OR Compressor

Philip Baback Alipour and Muhammad Ali
School of Computing,
Blekinge Institute of Technology
SE-372 25 Ronneby, Sweden
Emails: philipbaback_orbsix@msn.com and moh.ali@live.se

Abstract

We report a new lossless data compression algorithm (LDC) for implementing predictably-fixed compression values. The fuzzy binary and-or algorithm (FBAR), primarily aims to introduce a new model for regular and superdense coding in classical and quantum information theory. Classical coding on x86 machines would not suffice techniques for maximum LDCs generating fixed values of $C_r \geq 2{:}1$. *However, the current model is evaluated to serve multidimensional LDCs with fixed value generations, contrasting the popular methods used in probabilistic LDCs, such as Shannon entropy. The currently introduced entropy is of 'fuzzy binary' in a 4D hypercube bit flag model, with a product value of at least 50% compression. We have implemented the compression and simulated the decompression phase for lossless versions of FBAR logic. We further compared our algorithm with the results obtained by other compressors. Our statistical test shows that, the presented algorithm mutably and significantly competes with other LDC algorithms on both, temporal and spatial factors of compression. The current algorithm is a steppingstone to quantum information models solving complex negative entropies, giving double-efficient LDCs > 87.5% space savings.*

1. Introduction

In the world and market of data compressors, developing a *lossless data compression* (LDC) *algorithm* satisfying compression values much greater than 50% compression, even greater than 98%, would itself be a novel approach. Why this is important, is answered in how we perceive data compressors today. Imagine the amount of *space savings* and *bitrate savings* performed by some new algorithm in a consistent manner, i.e., predictable data compression values regardless of content size and input. The LDC techniques used today are for compressing commonly available documents, and are reported as probabilistic-dependence compression techniques only. There are a number of LDC algorithms to choose from, and they vary in methodology, code size and complexity. Which one is chosen, depends primarily on the specific structure of the data, as well as the objectives of the particular application. Most applications are compatible with popular compression standards, such as PKZip, GZip, WinZip, 7Zip, or UNIX's compress programs. Whichever compression standard is chosen, chances are it will require a large amount of RAM. For example, the WinRK with different compression profiles, if set for a slow and maximum LDC, uses 800 MB of RAM to encode a 10 MB data. (See also [29].)

Furthermore, the complexity and size of software systems have increased in recent years especially when it comes to LDCs. There are diverse techniques that perform LDC based on the mere-chance probability as random process. In principle, these techniques benefit from e.g., Shannon entropy [12, 13, 15], to compute similarities between data objects and their recursive pattern recognitions. More specifically, the compression is based on repeated patterns of input data to bit sequences (frequently encountered), [13, 24], restricted to random variables. Therefore, the algorithm loading different types of information with the same size, its compressed output, based on these random variables, would vary in output size and content (an *uncertainty*).

In order not to fail on the market, it is important to also achieve a high quality with intact data integrity when studying the output data. Such a quality expectation could be realized from probabilistic techniques that calculate distances between strings of x and y, for similarity comparisons, such as algorithmic complexity (Kolmogorov). For instance, GZip by default uses Lempel-Ziv coding (LZ77). Such compressors are used to foremost increase space savings, and in general, via e.g., algorithmic complexity, increase quality and quantity, data integrity, clustering, inheritance and grouping for their redundancy checks. In view of such factors, we evaluate a data compressor by its compression rate, and thereby decompression (LDD) based on char identification and address checks. The question is, whether a technique exists in using a new combinatorial logic that performs LDC with the least probabilistic factors. In other words, we search for a technique which *contrasts* symbol frequency and its firm dependency on Shannon entropy for repeated patterns of characters.

In the new technique, however, AND/OR logic is used to operate on *bit pairs* promoted from one data compression level to another. The logic is explicit and empowered by mathematical rules of mapping and abstraction levels of logic bits, prior to their data type. The data type is given in terms of strings and chars with integer limits. The logical choice of strings is in being a very common input data for texts, and is frequently used in e.g., WinZip, GZip, WinRK and LZW with embedded compression switches, to perform arbitrary LDCs.

We report a set of data compression test cases used for maximum data compression ratios, and evaluated the losslessness of data when decompressed. In the presented study, we mainly focus on three aspects:

- How can FBAR logic be applied onto a data structure for a data compression?
- How does the output data from the algorithm evaluated for its losslessness and integrity?
- How different would be the performance of the algorithm, compared to other compressors?

This paper is structured as follows: Sect. 2 gives background information on FBAR and other compression algorithms. It also gives details on their implementation differences. Sect. 3 focuses on FBAR test and structure. Sect. 4 presents related work. Sect. 5 introduces the FBAR test data generation and its components. Sect. 6 describes the experiment, while Sect. 7 contains the results. Sects. 8-9, end the paper with discussion and conclusion, respectively.

2. Background

The motive of finding FBAR came about in its logic which gave the author a sense of unifying different types of logic, proving their interrelatedness of logic states in information theory. The relatedness for each character entry on binary construct is presented by the principles of the completeness theory [1] and logical consequence [3] from different models, e.g., fuzzy logic [3, 4, 7, 8, 9], quantum cryptography [2], binary, and quantum binary logic [17, 18]. By making this uniformity, FBAR logic is emerged. This logic is possible when packets of Boolean values per character are normalized (bit insertion and update), and abstracted into relative states of fuzzy and quantum logic.

The FBAR algorithm, primarily aims to introduce a model for regular and superdense coding. In coding theory and cryptography [2], *superdense coding* is used to attempt a 2-bit transmission via 1 quantum bit (*qubit*). Formally, it is barely achievable to transmit *double-efficient* binary via quantum states between two points. However, this issue is resolved in FBAR, since it gives absolute predictable states in its model structure for 2*n* bits via *n* fuzzy qubits. The logical interrelatedness and its consequence, as such, are given later by Eq. (1).

2.1. Notations and terminology

Notation	**Short definition**	**Example**
C_r	Data compression ratio.	2:1 compression
C	Compressed data; compression.	$C_{-1} > C_n$, $n \in \mathbb{N}$
C'	Decompressed data; decompression.	$C \xrightarrow{out \times ref} C'$
H	Entropy rate in e.g., Shannon systems.	$H_{\mathcal{A}} > H_{\wedge\vee(b)}$
m	String size e.g., English alphabet size.	$m = 26 + 1_{\text{space}}$
ch_i	A character, where $i \in \mathbb{N}$.	$\{ch_1 ch_2 ch_3 \ldots\}$
x	Array dimension for the residing bits in memory.	if $x_i = 0$ then $x = [000\ldots0]$
y	Array dimension for projected bits from upper memory to lower layers.	if $y_i = \neg x_i$, $x_i = 0$ then $y = [111\ldots1]$
$x \rightarrow y$	Bits from horizontal plane projected vertically onto a compressed binary.	if $y_i = 1$ and $x_i = 0$ then $y = [0101...]$
λ	Variable lengths function on e.g., string, char, time or binary.	$C(x) = \frac{\lambda(x)}{\lambda(t)}$
∞	Infinity; undefined, subject to removal via e.g., new characters.	$2ch - \infty = 2ch$
$\searrow$	Fuzzy state leaned to low level logic.	$1\searrow0 = \{0 , \approx0\} \equiv 0$
$\nearrow$	Fuzzy state leaned to high level logic.	$0\nearrow1 = \{1 , \approx1\} \equiv 1$
$\curvearrowleft$	Right bit-to-left bit selector.	$x_0 \curvearrowleft x_4$
$\curvearrowright$	Left bit-to-right bit selector.	$x_0 \curvearrowright x_4$
β	Binary vale or sequence, where $\forall \beta \in f(x) = x \rightarrow y = b$	$x_0 \curvearrowright x_4 x_5 \curvearrowright$ $x_1 = \beta$, if $\forall x_i = 0$, $x_5 = 1$, $\therefore \beta = 0001$
$\wedge$,&	Logical AND otherwise, bitwise AND	$0\wedge1=0$, $1\wedge1 = 1$
$\vee$, $\vert$	Logical OR otherwise, bitwise OR	$0\vee1=1$, $0\vee0 = 0$
$\leftrightarrow$	Bidirectional between states or logic	$x \leftrightarrow y \equiv x \rightarrow y \rightarrow x$
$\equiv$	Equivalence; identical to …	2 chars $\equiv$ 16 bits
$\therefore$	Logical deduction; therefore …	$\{ch_1 ch_2\} = \{\$\%\}$ $\therefore ch_1 = \$$, $ch_2 = \%$

2.2. Logic and Data Type

This study is focused on the presentation and evaluation of an FBAR LDC technique. Especially, the focus is on the FBAR data compression, and thereby, its successful lossless decompression. By implementing certain functions in a programming language, like C, with more efficiency, the FBAR logic and its LDC product is achievable. The motive to perform LDC with the least probabilistic frequency occurrence of characters, such as, from the English alphabet, is to conceive the logic behind each character-entry denoting a spatial size limit occupation. In modern machines, each standard ASCII character entry, excluding the extended type or, an entry $\geq 2^{7\text{-bit code}} = 128$ decimal, occupies 8 bits or more of space, in which each bit is either, a low-state or high-state logic. A set of logic states, in combination, according to ASCII 7-bit code pattern match, build up a character information or its corresponding symbol.

To perform the least probability of logic operations, there must be a definite relatedness between binary logic and its in-between states of low and high, relative to their corresponding characters for each 8-bit block. In FBAR logic, this could be recognized at the lowest levels of binary logic between AND and OR operations.

As we relate characters in their binary construct, fuzzy logic comes to our attention to include more states for further LDCs on the same char entry without losing the initial 0 and 1 binary. As we progress, fuzzy logic is too connected and related to quantum logic, no matter how many states of compressed data, still, 8 bits of 0's and 1's could be transmitted via minimally 4 fuzzy bits, and thereby, 2 quantum bits, interrelatedly; or

$$\begin{aligned}&binary\ \ states \leftrightarrow fuzzy\ \ states \leftrightarrow quantum\ \ states\\ &\equiv \{0,1\} \leftrightarrow \{1 \searrow 0, 0 \nearrow 1\} \leftrightarrow \{00,01,10,11\}\end{aligned} \tag{1}$$

2.3. Lossless data compression algorithms

Lossy and lossless data compression algorithms both have one purpose i.e. to compress data. However, there is a great difference in their specialty which entails both quality and quantity on a given I/O data. Lossy compressors do not concern the conservation of data in quantity, and it is just how to present data to the point of delivery without losing significant details i.e. decompression with acceptable quality or readably recognizable data. For example, in video technology, it suffices for a user to receive images with even low quality as far as details are not lost in the picture.

Lossless algorithms, however, maintain all details between the two points of data source and sink. It is extremely crucial for textual data I/O, e.g., a dictionary, to maintain no data loss on a single character throughout the compression process, whatever LDC method is used.

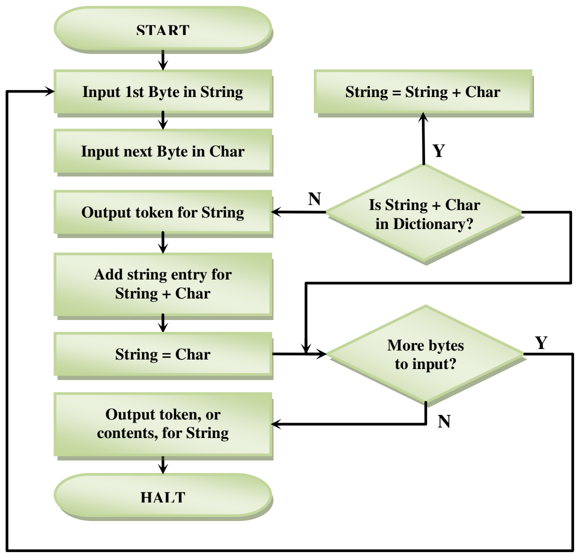


Fig. 1. Flowchart of a lossless type algorithm

Fig. 1 shows the flowchart of a simple lossless data compression algorithm [35]. As shown in this flowchart, the LDC algorithm is designed to input data, accumulate it, generate a dictionary that assigns tokens, and outputs them into a compressed format. An example of this is Lempel "Ziv" (LZW), a lossless data compression technique as an improvement to the popular LZ77 compression algorithm [4, 5]. We study current LDCs in their structure with the newcomer FBAR LDC algorithm, and thus highlight their explicit differences in logic, method, design and performance with FBAR.

Fig. 2, however, is a circular process representing FBAR. Both LDC algorithms must possess similar properties like LZ compressors that avoid string character misplacements, distance redundancy or token confusions on erroneous data reconstruction during the decompression phase. Similarly, anticipating character misplacements or symbol confusions in a document during decompression have been considered in the FBAR algorithm from a variant size to fixed size limits of memory space. Whichever compression standard is chosen, chances are, it will require RAM space. The more space dedicated to the compression program, the higher the compression ratio [29]. This yields in larger reference tables built by the LDC program.

The current challenge is to find some software that can achieve acceptable efficiencies within a small memory footprint. This article describes a lossless compression algorithm based on FBAR with a premise of a $2n$-dimensional dictionary of 'bit fields', strictly avoiding the concept of 'bit array' usage in its implementation. The reason is that in the latter, we would just encode rather than compress data, since bit arrays consume at least 1 full byte of memory for a single Boolean variable, i.e., a1-bit flag.

With appropriate bit-flag referencing upon compressed characters, FBAR achieves respectable fixed size compression ratios, typically on the order of 34-to-50%, while consuming about 64K of RAM. By extending the size of its flag reference table (Table 1), $n\times$two-dimensionally, the dictionary constructed out of it permits the order of 87.5% LDCs, and with future quantum inclusions, 98% LDCs are achievable i.e. an order of $2n$:1 ratios, where n is the number of bits. These LDC ratios are Eqs. (1) and (5) dependent, which solely means *fuzzy quantum binary computation*, rather than char frequency pattern match and occurrence.

On the other hand, LZW is capable of achieving respectable compression ratios, typically, on the order of 50 to 60%, while consuming about 2K of RAM. In larger RAM memory sizes, 8K or 16K, it is possible to achieve 80% efficiency or more [29].

As we shall later illustrate, an FBAR dictionary consists of a *translation table* and a *reference table*, both building a static size of flag information, later used by the program's interpreter for char comparisons. Fig. 2, shows a circular process of an FBAR LDC with dictionary, a combination of the algorithmic design and program's process model. The process comprises of program design and memory transactions with the usage of relevant functions and methods coded in C.

To conduct a successful data decompression, we renounce bit values based on a predictive pattern of bits in memory. This occurs subsequent to the double-dashed circle component in Fig. 2.

We constructed a 'char and binary' LDC reference table to satisfy these conditions during the compression phase of the algorithm. The conditional output per input char, subsists on relevant bit-flags and extended bits that are allocated in the memory. The allocation, read/write and reference process is shown in Fig. 3, denoting three major procedures to reconstruct data during an LDD. This makes compression values predictable regardless of content size and input, since a reference table is already constructed with unique bit values for every compression layer, starting with the 4th layer, upwards. The reference table is based on binary decisions, and is the core component of the FBAR algorithm aiming to reconstruct data at the decompression phase.

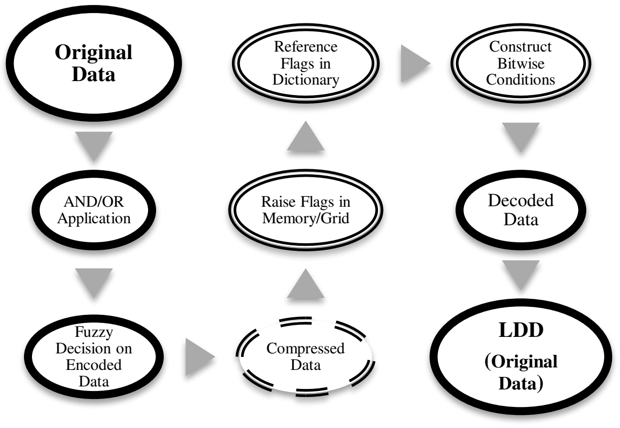

Original Data
AND/OR Application
Fuzzy Decision on Encoded Data
Compressed Data
Raise Flags in Memory/Grid
Reference Flags in Dictionary
Construct Bitwise Conditions
Decoded Data
LDD (Original Data)

Fig. 2. The circular process of an FBAR LDC

We further introduce the algorithm structure and test it for each single bit entry as follows:

3. FBAR structure and test

FBAR structure is a direct measure of test quantity, preparing the grounds to indirectly measure test quality. Its structure is used to identify areas of code that cover test case scenarios related to FBAR logic and the implementation for a unique LDC. These algorithmic properties are examined in the following sections:

3.1. FBAR logic, process and model

To fully implement an algorithm, one must understand how it works in terms of its testable structure and model representation. Therefore, the current set of test cases should be minimized, and thus tested in the small. Once implementation is resolved on this scale, test cases are maximized or extended to the large, in number, and in scale of I/O data integration. This would to some extent guarantee the correctness of the code on FBAR logic requirements.

For example, constructing an abstracted release of a reference table based on standard keyboard characters' input, including SPACE, would not exceed 96 entries: 95 printable ASCII characters (decimal # 32 to 127), and 1 control character. The use of the latter is to create a block or jump between every 96 bit chunk of memory dedicated to our table char entries. Each char entry consists of 8 bits or a byte, ready for a data compression. The data compression is performed and mapped per bit, allocated in memory for each identified char.

In Fig. 4, the encoded character corresponds to a unique compressed value as an enclosed form (*bit closure*) of AND and OR pairs, e.g., the columns having impure bits 01 and 10. A bit closure for 10, is 0, and for 01, is 1 in binary, which inclusively infer to *fuzzy transitive closure pairing of bits* [30] or *shortest path* for our binary set. The allocation of the raised single bits for the minimum LDC phase of FBAR is also shown in the same figure. The AND and ORed columns, each stand as a nibble, in total, giving 8 bits per character, which means, the character has been encoded on this level.

The process design and the development of the algorithm, however, are illustrated in Fig. 3.

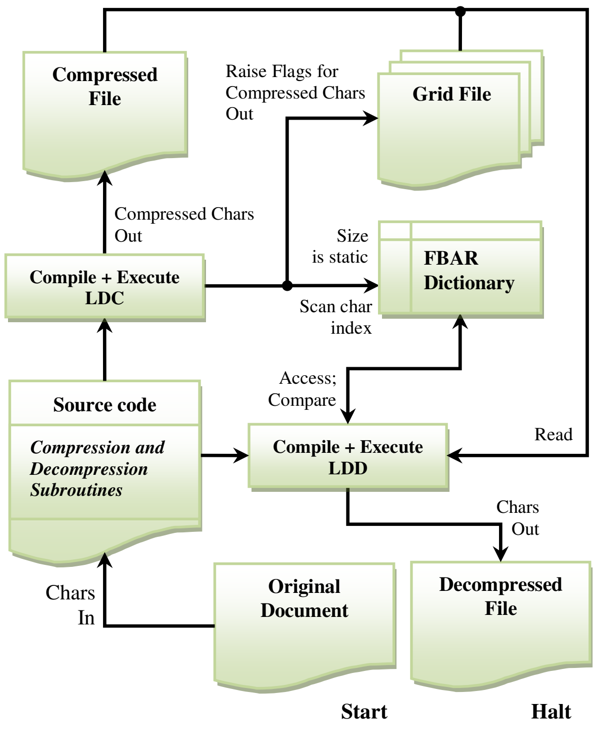


Fig. 3. Basic process design of FBAR binary I/Os

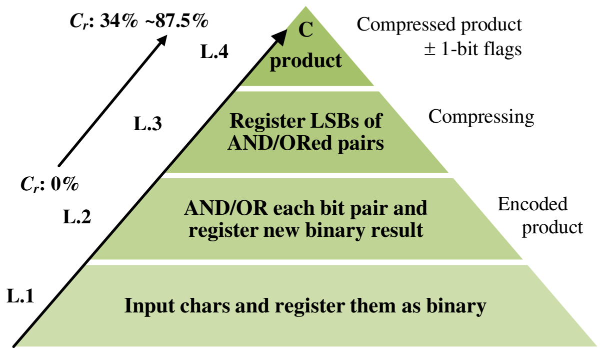

Cr: 34% ~87.5%
Cr: 0%
L.4
L.3
L.2
L.1
C product
Compressed product ± 1-bit flags
Register LSBs of AND/ORed pairs
Compressing
AND/OR each bit pair and register new binary result
Encoded product
Input chars and register them as binary

Fig. 4. Basic structure of FBAR binary projections

The encoding is unique and builds up our Fundamental Sequence (FS) encoding, to some extent contrasts to the 'entropy coder' and DCL model reported by CCSDS's green book [28]. Pairwise selection of bits according to bitwise projections of bits, after converting each character in the input sequence are picked and converted in parallel, one in ANDed, and the other in the ORed column. After this encoding process, high state and low state fuzzy binary conversions occur for compression. By now, every AND/ORed 'lesser significant bit' (LSB) pair is projected to the next levels of compression. The remaining bits are thus ignored. Each level of progressive projection from a lower layer to its upper, has its own 1-bit flag augment in aim of identifying *impure* 01, 10, and *pure* states of 11 and 00 for each converted data byte. In return, for a lossless decompression, the AND and OR columns could be paired according to Fig. 3 by tracing its sequential bitmap pattern to reconstruct data. This is done via a *grid file* as our *portable memory grid* on single bit flags to decompress data into its initial form.

3.1.1. Layers of lossless data compression

Highest layers of compression: Let grid file **G**, be a four-dimensional cube (*hypercube*) of pure and impure bit pairs with negation combinations, denoting a 'fuzzy transitive closure pairing of bit flags' [30], as follows:

ip: impure or pure pairwise bits' dimension:

iiii iiip iipi ipii piii iipp ippi ppii pipi ipip piip ippp pipp ppip pppi

zn: zero or negate pairwise bits' dimension:

zzzz zzzn zznz znzz nzzz zznn znnz nnzz nznz znzn nzzn znnn nnnz nznn nnzn nnnn

The key to either dynamic or static memory approach is in applying impure (**i**), pure (**p**) and *fuzzy transitive closures* to bit pairs (*pairwising FBAR logic*), where **p** as a custom bit-flag operator is either 11 or 00. The closure of this is simple to predict: it is 1 for 11 since AND/OR of 11 is 1, and 0 for 00 is similar. On the other hand, **i** is either 01 or 10. Based on the transitive closure, the latter bit pair-product of **i**, is the major problem, since it closes with either 1 for 01, or 0 for 10, which coincides with **p** conditions of 11 and 00 in bit products.

Solution: We first consider a pure sequence of bits, e.g., '11111111', and manipulate it with **ip**, then its result by **zn** combinations to reach the char-equivalent output in ASCII, e.g., @ ≡ 01 00 00 00. So, let **z** stand for *zero* or *ignore*, e.g., **z**(01) = 01, **z**(10) = 10, and **n** for negate e.g., **n**(01) = 10, **n**(11) = 00, etc. This is a *static solution*. For the dynamic solution we literary raise single bit **z**, **n**, **i**, **p** flags. We use **znip** to reconstruct data. But each occupies a single bit: **z** as 0, **n** as 1, **i** as 1, and **p** as 0. So, we raise them in a *static object* (in a grid/portable memory) to occupy 1 static byte per combination only.

Now we have successfully constructed four dimensions in a cube, embodying 4-bit flags/**zn** or **ip** combination. The motive for choosing this hypercube is anchored within the implementation of chars, being converted to binary, thereby generating self-contained flags within an input char of the **G** grid. This results in 50% pure compression, covering 2chars per entry, since each char is shared between 1**ip** and 1**zn** dimension, thus in total, 2chars ↔ 2**ip** + 2**zn** = 4 dimensions. More specifically, we put all of our emerging 1-bit **znip** flags in unique combinations for double-efficiency. We intersect them with other **znip**'s representing a second char input:

$$C(\text{2chars}) = 2\mathbf{znip} = (4\text{ bits} \mid 4\text{ bits}) \times (4\text{ bits} \mid 4\text{ bits}) \leq 8\text{ bits}$$
(*dynamic approach*)

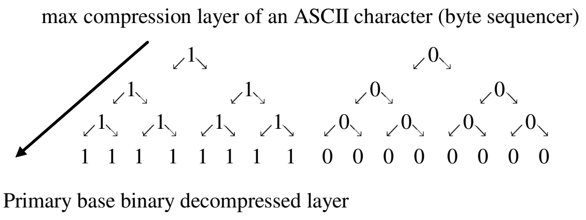
$$C(\text{2chars}) = 2\mathbf{znip} = (4\text{ bits } \times 4\text{ bits}) \times (4\text{ bits } \times 4\text{ bits}) = 8\text{ bits in}$$
`1x1x1x1` to `16x16x16x16` address (*static approach*)

The latter approach literary creates 4 dimensions in the given address range. The notation '`x`' or '∧', here, denotes just *intersection* of the values without bit manipulations (the occupying bit flags) between **zn** and **ip** dimensions, each independent of the other. This approach proves absolute double-efficiency. The former approach, however, ORs values in bitwise terms, hard but possible for an optimized version after the static version due to resulting in ≤ 8 bits for each 2char input.

Example: Consider the following levels of compression conducted by our program **P**, relevant to Fig. 4

$$\left.\begin{array}{l}
\mathsf{r} \xrightarrow{\text{in}} \boxed{\mathbf{P}} \to 01110010 \to \boxed{0100}\boxed{1101} \to \boxed{10}\boxed{11} \to \boxed{0}\boxed{1} \to 1 \\
\mathsf{e} \xrightarrow{\text{in}} \boxed{\mathbf{P}} \to 01100101 \to \boxed{0000}\boxed{1111} \to \boxed{00}\boxed{11} \to \boxed{0}\boxed{1} \to 1 \\
\mathsf{s} \xrightarrow{\text{in}} \boxed{\mathbf{P}} \to 01110011 \to \boxed{0101}\boxed{1101} \to \boxed{11}\boxed{11} \to \boxed{1}\boxed{1} \to 1 \\
\mathsf{o} \xrightarrow{\text{in}} \boxed{\mathbf{P}} \to 01101111 \to \boxed{0011}\boxed{1111} \to \boxed{01}\boxed{11} \to \boxed{1}\boxed{1} \to 1 \\
\mathsf{l} \xrightarrow{\text{in}} \boxed{\mathbf{P}} \to 01101100 \to \boxed{0010}\boxed{1110} \to \boxed{00}\boxed{10} \to \boxed{0}\boxed{0} \to 0 \\
\mathsf{v} \xrightarrow{\text{in}} \boxed{\mathbf{P}} \to 01110110 \to \boxed{0100}\boxed{1111} \to \boxed{10}\boxed{11} \to \boxed{0}\boxed{1} \to 1 \\
\mathsf{e} \xrightarrow{\text{in}} \boxed{\mathbf{P}} \to 01100101 \to \boxed{0000}\boxed{1111} \to \boxed{00}\boxed{11} \to \boxed{0}\boxed{1} \to 1 \\
\mathsf{d} \xrightarrow{\text{in}} \boxed{\mathbf{P}} \to 01100100 \to \boxed{0000}\boxed{1110} \to \boxed{00}\boxed{10} \to \boxed{0}\boxed{0} \to 0
\end{array}\right\rangle 11110110 \equiv \ddot{\text{o}}$$

Then we store the 'F6h' char, ö, from the ASCII table to the compressed file **C**. From there, when a binary value recalled by the decompression subroutine, the program then *interprets* the last layer of compression (rightmost column) for 'each enclosing bit', the following:



max compression layer of an ASCII character (byte sequencer)
Primary base binary decompressed layer

Therefore, the usage of these polarity combinations (impure/pure) and their counterpart states (negation **n** or ⌐), would be given in the following column matrices for a data reconstruction. The right-hand side matrix from the **G** file, its binary, represents the actual bit flags, and its right column is the interpretation of those bits when ORed between **zn** and **ip** dimensions. This is a *dynamic memory approach* using 'bit fields' in C programming.

In continue, consider the letter '`r`' from the top row, its corresponding bit from the last column before reaching the compressed char ö representing a compressed string called '`resolved`'.

$$\begin{bmatrix} 6 \\ 5 \\ 5 \\ 5 \\ 6 \\ 5 \\ 5 \\ 6 \end{bmatrix} \equiv \left\langle \begin{array}{c}
\underline{1011}\rfloor \lceil\mathbf{i} \lceil\mathbf{p} \lceil\neg\mathbf{p} \lceil\neg\mathbf{i} \\
\underline{1111}\rfloor \lceil\mathbf{i} \lceil\neg\mathbf{i} \lceil\mathbf{i} \lceil\mathbf{i} \\
\underline{1010}\rfloor \lceil\mathbf{i} \lceil\mathbf{p} \lceil\neg\mathbf{p} \lceil\mathbf{p} \\
\underline{1100}\rfloor \lceil\mathbf{i} \lceil\neg\mathbf{i} \lceil\mathbf{p} \lceil\mathbf{p} \\
\underline{1110}\rfloor \lceil\neg\mathbf{i} \lceil\mathbf{i} \lceil\neg\mathbf{p} \lceil\mathbf{p} \\
\underline{1011}\rfloor \lceil\mathbf{i} \lceil\mathbf{p} \lceil\mathbf{i} \lceil\neg\mathbf{i} \\
\underline{1111}\rfloor \lceil\mathbf{i} \lceil\neg\mathbf{i} \lceil\mathbf{i} \lceil\mathbf{i} \\
\underline{1110}\rfloor \lceil\neg\mathbf{i} \lceil\mathbf{i} \lceil\neg\mathbf{i} \lceil\mathbf{p}
\end{array} \right\rangle \leftarrow \boxed{\mathbf{G}}$$

We then decode via LDD subprogram comparator from the **G** file as follows:

this is the interpreted sequence

$$\boxed{\mathbf{C}} \to 1 \xrightarrow{\text{pure reconstruction}} 11111111 \xrightarrow{?} 01110010 \equiv \mathsf{r}\,,$$

while '`r`' being inputted to $\boxed{\mathbf{P}}$,

$$\therefore 11111111 \wedge \underline{1011}\rfloor \lceil\neg\mathbf{i} \lceil\mathbf{p} \lceil\neg\mathbf{i} \lceil\neg\mathbf{i} \xrightarrow{\text{out}} 10111010 \equiv \underline{\text{o}}\text{, is false,}$$

$$\text{iff}, \underline{1011}\rfloor \lceil\mathbf{i} \lceil\mathbf{p} \lceil\neg\mathbf{p} \lceil\neg\mathbf{i} \xleftarrow{\text{read}} \boxed{\mathbf{G}}, \text{ then} \left(\frac{\mathbf{ippi} \leftarrow 1001|0011 \rightarrow \mathbf{zznn}}{1011} \right)$$

Interpretation by LDD’s ‘if-else’ comparator: [do not negate the 1st impure pair; do not negate the 2nd pure pair; negate the 3rd pure pair; negate the 4th impure pair] of the sequence.

From the above two data points, we then deduce

$$\text{if } \left(\mathbf{1011} \wedge \mathbf{11} \wedge \left(01110101 \wedge 11111111\right)\right) \text{ then output } 01110010 \equiv \mathsf{r}$$

To code the *interpreter* within the *comparator*, one should assert in terms of the previous if-statement. Each two nibbles from the top-down of the **G** file (column matrices) represent one compressed character in the **C** file, in this case “ö”. Since this char is of Unicode type, and to avoid nonprintable or an extended byte, which is ≥ 2-byte allocation, we replace this char with a singleton {‘1’} derived from its most significant bit (MSB). The compressed character in **C** has now got one single bit or byte representative. Let this be a *byte sequencer*, if beginning with a 1, we put the ‘1’ char in the **C** file, otherwise, the ‘0’ char. Therefore, the interpreter interprets this as ‘11111111’, which means the bits of 0 Boolean value from the matrix are now altered into

$$\underline{1110}\rfloor \lceil\neg\mathbf{i} \lceil\mathbf{i} \lceil\neg\mathbf{p} \lceil\mathbf{p} \rightarrow \underline{1101}\rfloor \lceil\mathbf{i} \lceil\neg\mathbf{i} \lceil\mathbf{p} \lceil\neg\mathbf{p} \quad , \quad \underline{1111}\rfloor \lceil\mathbf{i} \lceil\neg\mathbf{i} \lceil\mathbf{i} \lceil\neg\mathbf{p} \leftarrow \underline{1110}\rfloor \lceil\neg\mathbf{i} \lceil\mathbf{i} \lceil\neg\mathbf{i} \lceil\mathbf{p}$$

Thus, the compressed characters in average, from the left matrix, build up 5.375 bits/char. To conduct the above statement, we thus code a `packed_struct` to pack flags as a structure definition to a *non self-embedded flag* approach (dynamic). For example, we code `f1:1` to `f4:1` for the **ip** dimensions of the **G** file. The right-hand side denotes the bit length of the flag variable on the left. For the **zn** dimensions, we code `f5:1` to `f8:1`, correspondingly. Now, to access a particular flag in **zn** or **ip**, we code, e.g., `pack.type = 6`, to access flag # 6. Here, the `packed_struct` contains eight members: four 1 bit flags `f1…f4` for probable **ip** combinations, the remaining flags, for a negation possibility upon the previous flags if, and only if, raised per combination. The **G** file could be considered as a low-level memory map assisting bit field compactions, even lesser than six 1-bit flags required for the ‘r’ char in the “resolved’’ sample. The further compacted version of the previous statement is

$$\text{if } \left(\overbrace{\mathbf{1011}}^{\text{Compacted } \mathbf{znip} \text{ flags}} \wedge \left(01110101 \wedge \overbrace{\{1\}}^{\text{Sequencer}}\right)\right) \text{ then output } 01110010 \equiv \mathsf{r}$$

The general version of this if-statement is embedded within the following pseudocode of the algorithm:

Pseudocode sample I: ***a lossless data compression***

```
WHILE there are still input characters DO
  CHARACTER = get input character
      CONVERT CHARACTER to BIN CHARACTERS
      PACK 1-bit FLAGS from any conversion level
  IF PACK + CHARACTER is in the Reference Table
  THEN
         PACK = PACK + CHARACTER in the G file
    ELSE
        OUTPUT the code for PACK as NEW STRING
        ADD NEW STRING + BIN CHARACTER to the C file
        NEW STRING = CHARACTER
    END of IF
END of WHILE
OUTPUT the code for PACK in G file
OUTPUT the code for NEW STRING in C file
```

The conversion sample on any input string, as shown above, are propagated via the intersection of the **znip** combinations established within the 4D **G** space. This is stored by *occupant chars* in the **G** file during the early stages of the FBAR compression process (see, ‘**method**’ below). This is a *static approach* and *double-efficient*, storing string values in the **G** file in terms of

$$C(m) \xrightarrow{\text{in}} \lambda(\mathbf{G}) = \lambda(\mathbf{G}) + \frac{m}{2}, \quad \mathbf{G} \geq 64\text{K}, \qquad (2)$$

where m is the number of string characters inputted to the program for a compression. Once compressed, the length λ of the grid file **G** is summed with the compressed m, equal to $m/2$. The default value of 64K comes from the three dimensions representing a char representative for each combination set of **ip** and **zn**. This default value is computed based on the possible number of grid outcomes, Eq. (2). This number is quite convenient for a 16-bit microprocessor to directly access and process the **G** file via a set of hardcoded ‘if-else statements’ on flags’ subroutine in our code.

As we shall later observe, to conduct an FBAR LDD, data access of the compressed file is in 65,536 rows, denoting a 64K limit. The expectancy of lower sub-layers of the 4th layer would decrease the number of possible combinations of 1×4-bit flags, making the cube denser than the current version.

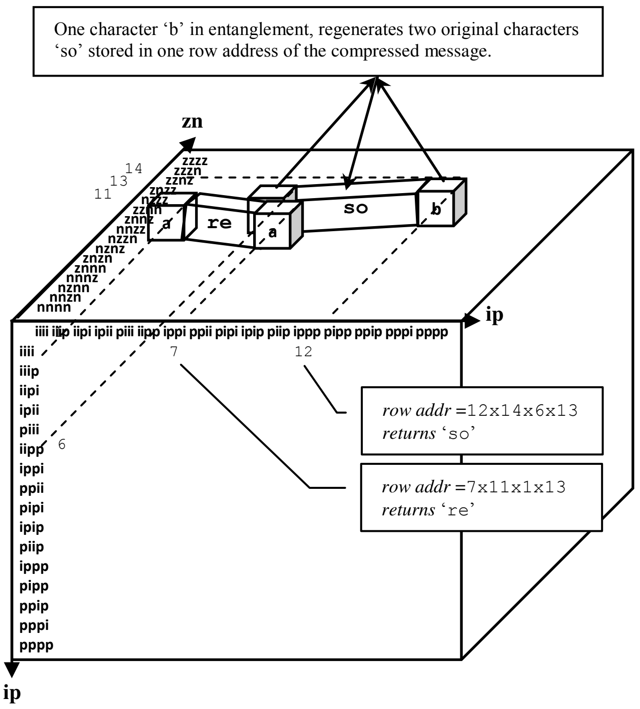


Fig. 5. The 4D logic constructor grid with input

This is due to having more bits available to decompress from those sub-layers of the algorithm. But in this case, the total number of possible combinations per dimension is fixed, or

$$\begin{aligned} \mathbf{G}_{\mathbf{xy}} &= \{ch_i ch_{i+1}\}\,\mathbf{ip} \times \{ch_i ch_{i+1}\}\,\mathbf{zn} \qquad (3) \\ &= 2^{4\times4} = 2^{16} = 65{,}536 \text{ possible grid outcomes} \end{aligned}$$

where the grid model is hereby shown in Fig. 5. Perceivably, in Eq. (3), out of the two *xy* bit flag field dimensions, we create a four-dimensional hypercube. This model could be considered as the **G** file's dictionary for a row number to its 4×1-bit flag translation. So, for every arbitrary input document, half of the size of that document is created between the four fixed dimensions of **ip** *x* vector for char ch_i, **ip** *y* vector for its neighboring char ch_{i+1}, and **zn** *xy* vector for both chars respectively. The **zn** and **ip** vector dimensions, each, are presented in separate rows in a list, mounting 16 indexed 4-bit flag sets correspondingly. The coverage of the grid is to concurrently cover all Unicode chars, even non-printable char scenarios for any data type. To verify the possible outcomes from Eq. (3), ASCII is, 256 × 256 = 65,536 for a 50% double-efficient compression.

A self-embedded flag set method: The cross-section, of which the compressed characters are recognized in the **G** file, is read by the 'decompression subprogram', thereby compared with the table contents for a successful data reconstruction. The entries are of the reference table, building up 95 standard ASCII chars. When the scanning of the **G** file entries reiterates for the next 96 char block, considering char # 96 as a block double byte (BDB), the program then counts from 97 up to 191 and so on, traversing all 65,536 rows, "flag sets", for an LDD.

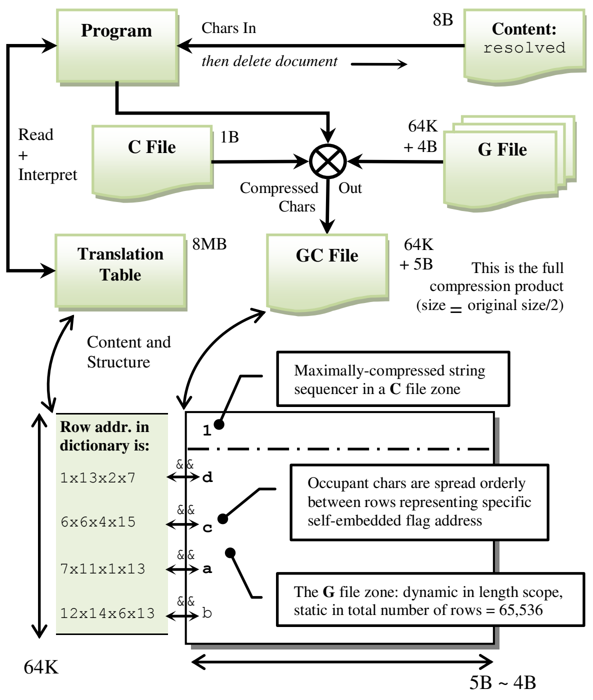

Program
Chars In
then delete document
8B
Content: resolved
Read + Interpret
C File
1B
64K + 4B
G File
Compressed Chars
Out
Translation Table
8MB
GC File
64K + 5B
This is the full compression product (size ≅ original size/2)
Content and Structure
Maximally-compressed string sequencer in a C file zone
Row addr. in dictionary is:
1x13x2x7
6x6x4x15
7x11x1x13
12x14x6x13
1
&&
d
c
a
b
Occupant chars are spread orderly between rows representing specific self-embedded flag address
The G file zone: dynamic in length scope, static in total number of rows = 65,536
64K
5B ~ 4B

Fig. 6. The GC file with an 8B to 5B~4B compression

We use the BDB as an indicator, e.g., a two-char '/a' representing the 1st full 96 byte allocation, '/b' for the 2nd, and … The BDBs are standard chars elicited from the ASCII table. The 'if and for loop' on the LDD, for 65,536 possibilities, is the key to this process. This is later explicated in pseudocode at the LDD phase. The rows are in matrix form, denoting at least two original chars held by a *position char* with a 1, otherwise, a 0 sequencer. The 'position char' as illustrated in Fig. 5, is an 'occupant char' stored in the **G** file, during compression (static), starting with an 'a' to the last ASCII 95 characters, representing in total, 95×2 = 190 char entries, or 95 compressed chars denoted by the ***C***(char) column in Table 1. For example, the elements in {a, b, c, d, …, /a}, are respectively interpreted by the program's interpreter as: the {1st 2chars, 2nd 2chars, 3rd 2chars, 4th 2chars, … end of the 95th 2chars [of the original file]}.

Table 1. I/O character process and occupation

Row address	*C*(char) #; C_r	Original chars; total	Occupant char	Size in bits
7x11x1x13	1; 2:1=50%	re 2	a	8
12x14x6x13	2; 2:1=50%	so 4	b	8
6x6x4x15	3; 2:1=50%	lv 6	c	8
1x13x2x7	4; 2:1=50%	ed 8	d	8
13x1x1x6	5; 2:1=50%	f 10	e	8
6x13x7x11	6; 2:1=50%	or 12	f	8
⋮	⋮ ⋮	⋮ ⋮	⋮	⋮
the same as last	48; 1:1=0%	∞ **96**	/a	16
8x12x8x12	49; 2:1=50%	55 98	a	8
8x12x11x2	50; 2:1=50%	5$ 100	b	8
⋮	⋮ ⋮	⋮	⋮	⋮

This alleviates char interpretation over binary when presented by char position through standard ASCII chars: a, b, …,□. To override memory overrun(s) during the vast access of files in read/writing data, we organize the '**G** with **C**' files into one single file, merging the targeted components of Fig. 3 into **GC**. A structural sample of **GC** is illustrated in Fig. 6. This approach makes the algorithm quite portable, thus no need to be concerned about memory allocation and management issues in this regard. The corresponding table to the grid, following bitmap pattern reconstruction for any character per impure and pure flag preference "*arose in bit field as necessary, is to hold a unique identity for that particular char,*" is given in Table 1. The previous output of the pseudocode sample I, thus complies with

```
OUTPUT code for CHARACTER + BIN CHARACTER in GC file
```

3.1.2 Layers of lossless data decompression

Now, by having all compression data established in the **GC** file, and having a *portable translation table* (always static in size≈8MB with Unicode), we could decompress data according to the following pseudocode:

Pseudocode sample II: ***a lossless data decompression***

```
READ the GC file row-by-row from end-of-file
OUTPUT temporary ROW_CHARACTERS
OUTPUT temporary (ROW_NUMBER == ROW_ADDRESS)
CHARACTER = ROW_CHARCTER
WHILE reading CHARACTER by CHARCTER DO
   READ ROW_NUMBER
   IF CHARCTER is not in the (ROW_ADDRESS AND
CHARACTER) of translation table with BIN CHARACTER
THEN
        STRING = get translation of OLD_CODE
        STRING = STRING + CHARACTER
 ELSE
        STRING = get translation of NEW_CODE
    END of IF
    OUTPUT STRING
    CHARACTER = 1st or 2nd or … or nth 2 characters in
               STRING
    REPLACE CHARACTER with 2 new characters from the
    translation table
    OLD_CODE = NEW_CODE
    DELETE temporary ROW_NUMBER and ROW_CHARACTERS
```

```
END of WHILE
```

The expectancy to reconstruct data, indicating a pure 50% LDC, requires a minimum of 96×96 = 9,216 'if-else' lines of code. The reason to that is, according to Table 1, we need to recover 2 original standard chars per 1 compressed char (see original chars and their occupant columns), just like the 'constructor process' illustrated in Fig. 6. So, one could say that the current grade of FBAR LDD is a simulation, and the LDC layout is just a tangible illustration of what is happening underneath the prototype. Ergo, the fully-implementable pseudocode covering char flag-set combinations is given below:

```
WHILE reading CHARACTER by CHARCTER AND compressed
BIN CHARACTER is '1' DO
READ CHARACTER as last block character
IF CHARACTER is a block character THEN
READ CHARACTER prepositioned to block character
READ ROW_NUMBER
GET ROW_ADDRESS from translation table
IF (CHARCTER ='d' AND ROW_ADDRESS = '1x13x2x7')
   OUTPUT STRING ='ed'
ELSEIF (CHARCTER = 'c' AND ROW_ADDRESS = '6x6x4x15')
   OUTPUT STRING ='lv'+'ed' = 'lved'
ELSEIF (CHARCTER = 'b' AND ROW_ADDRESS = '12x14x6x13')
   OUTPUT STRING ='so'+'lved' = 'solved'
ELSEIF (CHARCTER ='a' AND ROW_ADDRESS = '7x11x1x13')
   OUTPUT STRING ='re'+'solved' = 'resolved'
ELSE
PRINT no data or null compressed
END of IF
ELSE
PRINT no block character in range
End of IF
END of WHILE
```

The incremental concatenation of the '`OUTPUT STRING`' in the latter, corroborates with '`STRING = STRING + CHARACTER`' from the former pseudocode. The LDD interpreter, uses the `scanf()` tool to scan old data from the last block chars, i.e. from end-of-file (EOF) to its heading, subsequently, chars from end-of-line (EOL) after each BDB-read. These block chars as defined previously, could be lined-up after each 95 char accumulation, as e.g., {`/a`, `/b`, … `/` , `$a`, `$b`, … `$` , `@a`, `@b`, …, `@` ...}, alternatively, {€, □, ‚ , ƒ, … ÿ} as ASCII 8-bit or larger block chars. Once scanned, the interpreter reaffirms char occupants with their corresponding self-embedded flags (the related **G** row) from the translation table, and outputs data, reconstructibly. The "*translation table is the main component of the dictionary*", focusing on occupant chars and row address columns (see Table 1, columns colored in grey).

3.1.3 Layers of encryption and decryption

Lowest layers of compression: The flags raised in the 'reference table' *dynamically*, for encoding data, comply with the flag and polarity settings in Table 2. The main flags are # 0 to 6 polarity flags. The remaining flags are concatenated and thus raised in the grid file. Programmatically, one could select relevant bit pairs based on these tables to reconstruct data for lower levels of compression inclusive of maximum LDCs. Once bitwise combinations of the reference table are confronted within the LDD program code, bit access for reconstruction between the grid field and compressed file is enabled.

Table 2. Bit flag polarity combinations on bit pairs and nibbles during compression

Type no.	Polarity set	Implies to	1-bit flag
0	↓↑↑↓	most chars	f_0=1bit
1	↓↓↓↑	letters	f_1=1bit
2	↓↑↓↓	letters	f_2=1bit
3	↓↑↓↑	letters	f_3=1bit
4	↓↑↑↑	letters	f_4=1bit
5	↓↓↑↑	few letters	f_5=1bit
6	↓↓↓↑, ↓↑↑↓, …	dual chars	f_6=1bit
7	↘	all 2bit binary 10	f_7=1bit
8	↗	all 2bit binary 01	f_8=1bit

The above grid, however, is used and customized for any level of compression, either of lower layers of 4th up to its topmost possible LDC product.

A lower level encoding: For example, to reconstruct a character with decimal # 64, as "@", based on a raised flag, say, flag # 0 (*neutral* or *ignorable*), the equivalent of the character's binary would also be 01000000. The character's compressed version through FBAR using its flow (Fig. 4), or its model (Fig. 3), is "00↘0", denoting that the first two zeroes are pure and give 0000, whereas the second pair "↘0" is indeed impure. The latter's true face is "10 0", indicating flag # 7. Thus, the bit flag dereferences noise as 10 during decompression, and for the remaining 0 in "↘0", becomes 00. In total, we then have, 0000 1000 = 2 nibbles = 8 bits ≡ 1 character. Now, we establish the pattern based on flag # 0 i.e. its polarity set, since we code our algorithm that every nibble is of a previously-ANDed type, and next to it, from left-to-right of a binary sequence, the ORed type (consider them as odd and even nibbles in a full binary sequence with a length > 8 bits). Hence, the ANDed version sits above as the North Pole, and the ORed version sits below as the South Pole:

$$\begin{array}{l} 0000 \\ \downarrow\uparrow\uparrow\downarrow = 01\ 00\ 00\ 00 \equiv @ \ , \\ 1000 \end{array}$$

Programmatically, one could conceive this in terms of an equivalent pairwise selection from memory in a sequential manner.

Consider an accustomed byte to some char in terms of

$$\text{ANDed} \rightarrow 0000\ 1000 \leftarrow \text{ORed}\ ,$$

Equivalently, pairing the bits in terms of

$$x_0 \curvearrowleft x_4x_1 \curvearrowright x_5x_2 \curvearrowright x_6x_7 \curvearrowleft x_3 = 01000000 \equiv @$$

The pairwise mask function, shifting bits to the right otherwise to the left, could do this encoding, i.e. a "bit registry process' implemented in terms of the following portion of the pseudocode

Pseudocode sample III: ***a lossless data encoder***

```
CREATE a FILE POINTER for READ_WRITE operations
WHILE reading CHARACTER by CHARACTER DO
OUTPUT CHARACTER as temporary BIN CHARACTERS
   READ BIN CHARACTERS
   NEW STRING = BIN CHARACTERS
BITWISE AND(1st2 CHARACTERS of STRING from MSB to LSB)
BITWISE OR (2nd2 CHARACTERS of STRING from MSB to LSB)
BITWISE AND(3rd2 CHARACTERS of STRING from MSB to LSB)
BITWISE OR (4th2 CHARACTERS of STRING from MSB to LSB)
  IF 1st2 CHARACTERS in STRING is '01' THEN
     OUTPUT rightmost CHARACTER of this pair = '1'
  ELSEIF 1st2 CHARACTERS in STRING is '10' THEN
```

```
      OUTPUT rightmost CHARACTER of this pair = '0'
   ELSEIF 1st2 CHARACTERS in STRING is '00' THEN
      OUTPUT rightmost CHARACTER of this pair = '0'
   ELSE
      OUTPUT rightmost CHARACTER of this pair = '1'
   END of IF
CONTINUE SORTING 2nd2 CHARACTERS, 3rd2 CHARACTERS,
             4th2 CHARACTERS in STRING like before
OUTPUT RESULTS from BIN CHARACTERS to ASCII as 8 BIN
                    CHARACTERS = 1 ASCII CHARACTER
END of WHILE
```

As we can see, we simply compress data by selecting the least significant bit (LSB) of the pairs per nibble denoting closure points. This could be registered after applying bitwise *and-or*, and from there, after converting from compressed binary to compressed char, written to the **G** and **C** files in parallel. The simplified form of the 'if statement' with its 'continuing course on sorting binary chars' in the pseudocode, would be

```
...
SHIFT from MSB to 2nd rightmost CHARACTER in (
                  1st2 CHARACTERS, 2nd2 CHARACTERS,
                  3rd2 CHARACTERS, 4th2 CHARACTERS)
OUTPUT 2nd rightmost CHARACTER from (1st2 CHARACTERS,
  2nd2 CHARACTERS, 3rd2 CHARACTERS, 4th2 CHARACTERS)
...
```

This results in, for every 8 bits, a 4-bit output, and from there, 2 bits, and finally, a 1bit output char. We pack each 8×1-bit output into 1 ASCII equivalent char as our compressed version. The subsequent pseudocode represents what is necessary to code for an LDD, as a subroutine to the above code, recalling compressed values stored in char:

Pseudocode sample IV: ***a lossless data decoder***

```
WHILE maksing BIN CHARACTERS from BITWISE AND and
BITWISE OR results DO
STRING = 8 BIN CHARACTERS
ASSIGN '0' to a DOWN variable
ASSIGN '1' to an UP variable
FLAG_STRING = UP + DOWN CHARACTERS
    IF FLAG_STRING = (DOWN + UP + UP + DOWN)
    CHARACTERS THEN
        STRING = (MSB BIN CHARACTER + 5th BIT
        CHARACTER) + (6th BIT CHARACTER + 2nd BIT
        CHARACTER) + (7th BIT CHARACTER + 3rd BIT
        CHARACTER) + (LSB BIN CHARACTER +
                      4th BIT CHARACTER)
        OUTPUT STRING = OLD 8 BIN CHARACTERS
    ELSEIF CONTINUE CONCATINATE for other
    FLAG_STRING UP + DOWN combinations
...
END of IF
OUTPUT RESULTS from BIN CHARACTERS to ASCII as 8 BIN
              CHARACTERS = 1 ASCII CHARACTER
END of WHILE
```

So, considering the '@' char, we reconstruct 0001 0000 via bit concatenation from the '`8 BIN CHARACTERS`' or '`2 NIBBLE CHARACTERS`', denoted by a '+' in the code. Hence, during the decompression phase, having a set of up and down flags available, makes the algorithm to reconstruct data by tracing the arrows' directions in the polarities set. Interestingly, the "@" char is also a dual character (it behaves as such), and could be raised by flag # 6 due to giving the same result for its decompressed version with different polarity combinations. But for reasons needed to occupy fewer bits, even in form of 1-bit flags, we reconstruct data by reciprocating with the grid file, cross-referencing with distinct bit groups, and building up C_r values ≤ 2:1 compression. The main focus for reconstructing data, is considering negation flags # 1 to 4, pure and impure flags # 1 to 4, ORed in combination for each compressed character in the **C** file. A *comparator* as the FBAR program subroutine compares results between the static table as a point of reference with the dynamic component, **C** file, and the semi-dynamic component, the **G** file. The process relationships have been illustrated in Fig. 3. From there, a compression of 4-bits per compressed chars in the final layers as a 1-bit representative is stored. In total, 5 bits for each string entry identified for a decompression. To every unique combination of pairs made by the comparator, a specific 1-bit flag is allocated in the fixed size memory chunk with a specific address like from the portable compressed file **C**. This phase of LDC denotes a 5-6 bit compression, giving an average anticipation of 34 to 46% space savings for a 95 random string entries. The allocation of single bits raised in the memory, and from there, to the **G** file for each character per memory chunk is computed by the following equation:

$$C(m) = \lambda(\mathbf{G}) + \frac{m}{2} + \lambda(\mathbf{C}) = 64\,\mathrm{K} + \frac{5m}{8} \qquad (4)$$

where m is the number of string characters inputted to the program for a compression. Once compressed, the length λ of the grid file **G** is summed with the length of the compressed file **C**, for any quantity of chars measured by m in bits and bytes on each read for an LDD. Thus, the total is a value of (64 + m/2 bytes) + (m bits or m/8 bytes) compressed. The remainder bits added "m/8 bytes" come from the measure on the last layer of the LDC product. This makes any customized fixed table as a reference table to identify the initial character entries during the LDD phase of the algorithm.

Successful char identifications via 'for' and its nested 'if loops', makes a lossless compression absolute in all angles of bitwise operations. Identical chars, are created by these loop calls on different reference points between **GC** and translation table contents. The main rule for each row of entry is to always maintain a 2×4 *and-or* bit encoding, and a < 8-bit data compression.

3.2. Contribution

The main contribution in this paper is presenting a new model on self-embedded flags from Sect. 3.1.1. It allows an LDC algorithm to conduct superdense coding i.e., doubling the efficiency between two points of data transmission. This established a key difference in techniques, observed between the FBAR algorithm and other LDC algorithms. According to the "theory of data compression" [12, 13, 15], we conclude that almost every LDC uses Shannon entropy as its 'logic base' in conducting a lossless compression. In fact, repetition of characters in a certain frequency based on the theory of probability is embedded in such LDCs. In layman's terms, information entropy is the same as "randomness". A string of random letters and numbers along the lines of "5f78HJ2Z2Xp4V7Vb6" can be said to have high information entropy, or, large amounts of entropy, while the complete works of Shakespeare can be said to have low information entropy. Their LDC products are quite variant, which depend on content pattern probability or character rate of recurrence. FBAR's LDC, however, deals with the computation of binary logic regardless of content size and type, whereas other techniques are not bothered about. Binary logic in FBAR, deals with individual bits, their combination, repetition, cubic conservation, regardless of character repetition or content type. This means, based on a fixed size character reference table, Table 1, we derive a new more certain equation (least zero order H values), which is logarithmically the least probabilistic with discrete entropy (bits per character), compared to Shannon's entropy rate on English alphabet given by

$$H_{A} = \log_2 m \ = 4.75 \text{ bits/char}, \tag{5}$$

and for higher orders of H, for a given text source made up of English alphabet letters, becomes 4.07, 3.36, 2.77 and 2.3 bits/char, respectively. In FABR, however, fixed values of C for every order remain

$$H_{\wedge\vee(b)} = \log_b |\beta| = [0, 2.4[\text{ bits/byte}, \tag{6}$$

and for a binary sequence β, the binary probability of two states, b = 2, constructing 1 char, entropy H becomes 2, 1 and 0 bits/byte, regardless of source for a given fixed size LDC binary reference code. This makes the algorithm to compute information reliably based on fuzzy binary, rather than string characters.

The process in Eq. (6) evaluates every character by using *and-or*, *pure* and *impure* logic, and from there, further LDC's between bits of information. Eq. (5), however, deals with the random process to evaluate the whole sequence of characters using probability theory for an LDC result. Eq. (6), by comparison, improves less dependency on symbolic representations, and mainly, dependency on binary logic, thereby, fuzzy, and finally, quantum logic. The latter, however, remains quite intact with higher orders of probability equations promoting Shannon zero-order through third-order and general models, in simplistic sizes of LDC.

The reason is that, quantum logic by itself is based on probability behavior over bit states. To assist these relationships between logical events of the FBAR algorithm, we define them as LDC causality in form of supreme states of compression. For any data type at a quantum level, the current model (Fig. 5) holds good for superdense coding operators [27]. In our next report, we improve our model design, reconfiguring **znip** flags in an extended translation table, in aim of super-compressing an encoded message, thereby decode and decompress. The FBAR logic would then be called as FQAR or *qubinary* (quantum binary) *and-or* in its ultimate performance of LDC. Hence, a *negentropy* < 0 bits/byte of Eq. (6), denoting double efficiencies above 87.5% compression, for a universal predictability, is not farfetched in reality (see, e.g. [31]).

4. Related work

This section gives an overview of other works assisted in the FBAR algorithm for its prototypic design (Fig. 8) rather than implementation. Related work mainly constitutes *the separated versions* of the *combinatorial logic synthesis* of FBAR, i.e. {F, B, AND/OR} by well-known scholars, e.g., G. Boole, C. Shannon and L. Zadeh [33, 34, 3], chronologically. These works made it easier to distinguish our combinatorial logic by Eq. (1), from other LDC algorithms over various test cases as input strings and documents. Our logic is not of other LDCs that rely on randomness based on repeated symbols in content. In our implementation, satisfying a set of systematic hypotheses is indeed recognizable in Sect. 5.1.

The LZ77 and LZ78, are two LDC algorithms published in papers by Lempel and Ziv in 1977 and 1978 [4, 5]. The Lempel-Ziv algorithm is a *variable-to-fixed length code*. They are both dictionary coders, *unlike minimum redundancy coders*. However, they are only equivalent when the entire data is decompressed, as long as the entire dictionary is available. As of 2008, the LZ77-based compression method is by combining LZ77 with Huffman coding. Literals, lengths, and a symbol to indicate the end of the current block of data, are all placed together into one alphabet. This is often the specialty of LDCs used today for compressing common documents, which are reported as *considerably probabilistic-dependence compression techniques only*: e.g., XML model, DAG -compression [25], and lossless entropy coding [26].

To our knowledge, when FBAR is executed by an x86 compiler, it is evident how this LDC performs with fixed sized C_r's, regardless of what document or data type. The grid file, or compressed file, and the decoder in FBAR, do not act as same as the LDC packages mentioned above. However, apart from FBAR logic, design and implementation, the strategy in assimilating the components of the program into an LDC and LDD structure, like Fig. 8, *their evaluation* is of definite resemblance with these packages. To validate such issues in practice, we needed to test data from the input level to its corresponding output according to our componential model, Fig. 3. This led us to further verify our cyclic model from Fig. 2: which one of our hypotheses is valid, and which one disqualified on current machines. We reject their null hypotheses based on the implementation of each pseudocode (dynamic/ static, from Sect. 3.1), step-by-step, as a successful approach in evaluating the I/O products of the algorithm.

5. Experimental setup and application

The following addresses the preliminary conduction of the FBAR experiment in terms of, applicability and implementation of logic relative to performance, discussed as follows:

5.1. Test data generation for input data

Let us put hypotheses *H.1* to *H.4* into discussion before showing or discussing the prototype. These hypotheses were formulated in our thesis proposal as a systematic implementation satisfying our research questions in Sect. 1, whereby, each hypothesis constitutes the aims and objectives of our work. In virtue of our design process, the following hypotheses have been verified for data generation test cases, as 'true', otherwise 'false':

Table 3. Hypotheses and their 'true' or 'false' states

Hypotheses	x86	fqubit	Tested OS
H.1- Input of any data type to the FBAR's 1st layer, results in binary representing the same original content.	**True**	**True**	**Unix; Windows**
H.1₀ - The conversion of any data type to binary is impractical.	**False**	**False**	
H.2- A sequence of pairwise selection of bits to the FBAR's 2nd layer, when a parallel *and-or* applied, results in an encoded binary message in the 3rd layer.	**True**	**True**	**Unix; Windows**
H.2₀ -The pairwise selection and *and-or* operation on a binary sequence, is firstly *H.1* dependent, and secondly, irreversible for data reconstruction. (Or, backtracking to the original message is impractical.)	**False**	**False**	
H.3- A sequence of pure and impure pairwise selection of bits to the FBAR's 3rd layer, once detected and replaced with single bits, results in a compressed message in layer 4.	**True**	**True**	**Unix; Windows**
H.3₀- The pure and impure pairwise selection and compression to single bits on a binary, is firstly *H.1* and *H.2* dependent, and secondly, irreversible for data reconstruction.	**False**	**False**	
H.4- A sequence of single bit flags representing compressed data in FBAR's 4th layer, once reused adjacent to other purely compressed 1-bit data, results in a decompressed message from layer 4.	**True**	**True**	**Unix; Windows**
H.4₀- The sequential recall and reuse of bit flags from memory/grid, is firstly *H.1*, *H.2* and *H.3* dependent, and secondly, unachievable for an identical data reconstruction.	**False**	**False**	
H.5- A sequence of compressed data in form of *H.3*, when equipartitioned and paged into memory or confined signals in information space/grid, results in a maximum compression possible > 87.5% in layer 4.	**False**	**True**	**N/A**
H.5₀- The compression of any data length into one single bit is firstly *H.1*, *H.2* and *H.3* dependent, and secondly, unmanageable and irreversible for data reconstruction like *H.4*.	**True**	**False**	

The 'false' verdict indicates that the expected hypothesis, either null or else, is rejected, ergo, the 'true' verdict indicates otherwise. We set up our experiment under UNIX and Windows for each step of our systematic hypotheses. For comparisons' reasons, we study our LDC products coming from *H.1* to *H.4* under different platforms. We develop our prototype in C due to having efficient tools e.g., pointers, for our dynamic and static approach (see Sect. 3.1.1, and 'bit array' in Sect. 2.3). The implementation in our prototype led us to focus extensively on *H.4*, since it required more coding methods to supply the dictionary coder frame at the LDD phase of the algorithm. The *H.4* constructed a grey line between higher levels of compression to maximum levels of compression expected in an *auxiliary hypothesis H.5*. Frankly, the four hypotheses are within the territorial abilities of the prototype on x86 machines satisfying *C*'s ≤ 87.5%. Hypothesis *H.5*, however, is to be tested independently and promises double-efficient LDC ratios with negative entropy benefiting fuzzy quantum binary (*fqubit*). Ergo, the foreground of the *H.1* to *H.4* products is satisfactory to test input data according to our circular process presented in Fig. 2, Sect. 2.3. The column on fqubit in Table 3, under simulation conditions is applicable, yet untested for practical use. The current tests have been conducted *in the small* and *in the large* as follows:

5.2. Test data generation for input data

Small input data allows accurate comparisons between original chars during the input phase, compression and decompression. To see whether data is reconstructed successfully, the output is therefore compared with its original. From there, it is logical to make test-runs on large input data or file(s), since data integrity evaluations are conducted during small sample runs.

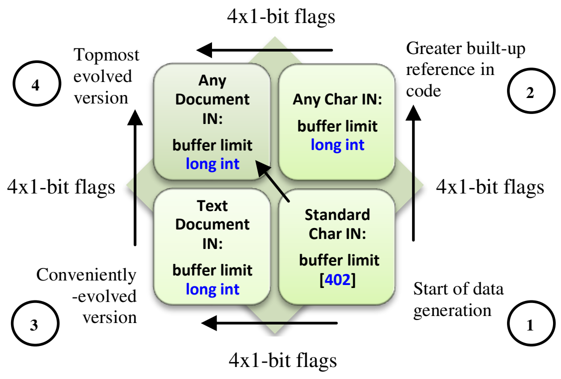


Fig. 7. Input data types used for a set of test-runs

Working with large samples on the first runs would be extremely complicated and almost impossible to manage per input document. Once char integrity evaluated on the smallest scales possible with certain buffer limit, assigning string size to the counter variable, building up the sample, would result in manageable flows, and easy validation on data results. The '`long int`' limit, is integrated within 'code loops' to store occupant chars in the **G** file, as 4-bit flag representatives. In case of an LDD with any size input, through proper access and comparisons of values from the translation table (Table 1), with the occupant chars within the grid, an evolution of different versions starting with textual type to any data type is achievable. This is shown in Fig. 7. The current FBAR subsists on the three, upper-right, lower-left and lower-right (starting point) of the matrix, evolving toward the last version of any document type beyond the level of chars.

5.3. The FBAR prototype

In this subsection, we introduce the FBAR prototype, as an LDC based on a *fuzzy binary and-or logic*. The current FBAR prototype is written in C with source code. Fig. 8 shows the basic structure and the main components of this prototype. In Fig. 8, the system starts by receiving an input string for preliminary conversions as specified in Sect. 3, starting with *and-or* logic. The starting point is by choosing the relevant 'menu option' executing one or more of the hypotheses *H.1-H.4*: **1-** the pairwise selection of bits after converting each character in sequence, as the encoding of AND/OR process, **2-** high state and low state fuzzy binary conversions, and **3-** the **G** file commitment over compressed bits for an LDC by raising 4×1-bit **znip** flags, are the main tasks of this prototype.

These tasks are outlined as 'conversion tools' in Fig. 8. Normal conversions like *char*-to-*bin* are classed as LDC routines. Reverse conversions for reconstructing data are classed as LDD. Having these prerequisites implemented for this logic, the program, at the LDD phase, loads a sequence of chars, and the prototype produces a set of mathematical compressed values of the same chars into the **GC** file (see, Fig. 6).

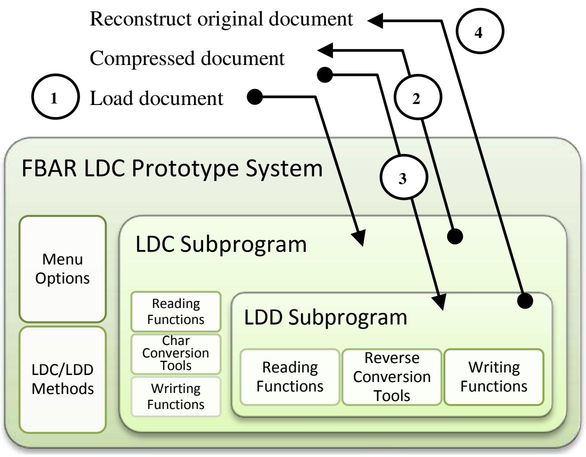


Fig. 8. Structural components of FBAR prototype

The prototype tests hypotheses *H.1-H.4*, as a representative of a 'dry run' for the algorithmic first time implementation. The prototype, by referring to the self-embedded flags in a translation table, dereferences char values with feedback based on 'if' and 'for loop' conditions. It uses a `deRef()` function conditioned in pseudocode sample II. The dereferencing function, once it finds a match between the **GC** file and dictionary, returns char values in a new file as reconstructed characters, just like before, as it is suppose to be in the original file. Once the decompression goal is achieved, the tool delivers a set of string characters identical to those characters that were initially inputted.

6. The LDC comparisons experiment

We have, by now, tested the FBAR's applicability. Now, we examine its code results. Therefore, a number of different LDC test packages are selected that vary in their code complexity and structure, as well as the complexity of input data they require. They range from classical code snippets, to more complex methods taken from the LDC Standard Dictionary Coder, and similarly, programmed compression switches. The compression test packages are listed in Table 4.

The outcome of this experiment is compared with the results obtained by probability distribution testing over a set of documents chosen by random. Three lossless compression packages are selected based on their ranks relative to FBAR prototype over input documents. The ranking is given on the basis of the three criteria, later given in Sect. 7. A sample document is tested several times, to obtain a good estimated result and, to make sure that the data is consistent and satisfactory to hypothesis *H.6*. The test run terminates when document is loaded and dealt with systematically, from one function call to another. Once the closure of these functions within the method construct is met with, then an exit or termination loop is called for the next document-load. These functions are pseudocode dependent and specified in Sects. 3.1.1 and 3.1.2.

6.1. Nonparametric comparisons test

This test gives LDC package comparisons' results based on ranks irrespective to the encoded, decoded and decompressed data. In fact, the focus is on the results of how long the computation lasts per sample, its spatial consumption i.e. the percentage of compression relative to sample's rank. The main motive for using nonparametric Freidman test is that, we cannot assume normality of the distribution we draw our samples from. Thus, one of the primary assumptions of parametric tests like *t*-test and ANOVA are not valid. Furthermore, since our number of samples is small ($n < 20$), we use Freidman test to analyze the data, and thus test its hypothesis, given below. Also, the FBAR case is confined to the distribution of repeated observations on LDC I/O samples like many non-parametric tests, based on the ranks of the data, rather than their raw values to calculate the statistic. In the following sections, we aim to follow this test to evaluate our algorithm compared to other LDCs. Therefore, we wanted to make sure that its results were statistically significant and not obtained by chance. Thus, we considered the following null hypothesis:

Let X contain our FBAR technique as well as a selection of state-of-the-art compression techniques. Furthermore, let Y contain a representative sample of documents of different type. Therefore,

H.6- A difference exists in the performances of the techniques in X as measured on Y by computation rate and space savings.

H.6$_0$- The difference in performances of the techniques in X as measured on Y by computation rate and space savings is zero.

7. Results

By referring to relevant sources giving details on LDC packages [19], one could outline the basis of the statistical test for results. Table 4 contains these packages with their respective ranks reflected in Table 5. We run our statistical test for comparing three or more related LDCs, as a result of their space savings, which makes no assumptions about the underlying distribution of the C_r data. The data is set out in a table comprising n rows by k columns. The data is then ranked across the rows and the mean rank for each column is compared. Bitrate ranking is statistically compared between the highest and lowest ranked algorithms, further constituting our null hypothesis. The comparisons data is given in Fig. 9.

The selection of an LDC algorithm depends on the following criteria applicable to all LDCs:

1. The ability to compress input data losslessly, regardless of type, content size and complexity.
2. Use memory for data access and management issues efficiently, e.g., data rate and spatial occupation of bits during compression when encoded/decoded, and referenced upon.
3. Must have a dictionary coder for validating data, referencing and dereferencing them during the reconstruction phase of data, i.e., decompression.

Based on the above characteristics, the ranking of the algorithm is given through percentages of C_r for each package. The C_r is not fixed for each algorithm and merely based on probability and character letter counts or, frequent reoccurrence to conduct a lossless compression. The only algorithm that differs from this behavior is FBAR, which exhibits predictable C_r ratios regardless of content size and complexity. Its fuzzy quantum version is FQAR, an expected outcome based on FBAR's current cipher properties. The selected packages in Table 4, were on the basis of best case probable scenarios in compressing data above 90% as a maximum LDC, 50% as a convenient LDC, and below 50% > 0% as a classic LDC, with reasonable bitrates.

Table 4. Test case LDCs based on *space saving* values

Document	#	WinZip	GZip	WinRK	FBAR	FQAR *
text	1	70.00%	85.70%	87.87%	50.00%	87.5%
book1	2	70.80%	69.00%	80.04%	49.47%	86.57%
book2	3	65.40%	63.80%	77.11%	48.95%	85.66%
paper1	4	65.60%	64.70%	73.58%	50.00%	87.5%
paper2	5	62.80%	61.60%	69.00%	50.00%	87.5%
paper3	6	60.00%	59.50%	68.25%	50.00%	87.5%
web1	7	72.20%	71.40%	75.37%	48.95%	85.66%
web2	8	53.80%	53.60%	54.57%	49.47%	86.57%
log	9	95.59%	95.37%	96.43%	48.95%	85.66%
cipher	10	73.30%	70.30%	77.82%	48.95%	85.66%
latex1	11	70.00%	69.00%	78.28%	50.00%	87.5%
latex2	12	66.52%	66.53%	75.70%	50.00%	87.5%

* FQAR is the fuzzy quantum version of FBAR, whereas the latter as fuzzy binary, is the predecessor to FQAR, displaying 87.5% C_r's.

Contradictorily, for the fixed C_r generated by FBAR, is conveniently more reliable in predicting C_r ratios compared to the probabilistic C_r's by PKZip, GZip, WinRK LDC packages. The ranking is further evaluated when package evidence of random sample inputs are measured non-parametrically. The rank of '1st' on FQAR, whilst as a column count is dismissed from experimental reality, is a dilemma between the ranks on definite techniques, unless implemented for an observation. The inclusion of FQAR is intricately significant due to the facts presented earlier on i.e. 'the 4D grid' in its expandable 4-bit flag combinatorial dimensions from Sect. 3.1.1. It gives 50% now, 87.5% later, on the same x86 machines. According to Eq. (6), for binary 2^0=1 bits/char cases, an 8 to 1-bit compression is evident. If 8 bits is 100% quantity, thus, 1-bit is 12.5%, giving a space saving of 100 – 12.5 = 87.5% relative to 'bitrate performance'.

The translation table results for maximum LDCs: By recalling Table 1, the focus on 87.5% LDC is that, the column with 96 occupant chars in the dictionary will not change in content translations. However, the grid's row address column in configuration '1x1x1x1', becomes '1x1x1x1 1x1x1x1', and the column with 2 original chars, becomes 8 chars to reconstruct. Thus, the representation of the '1st 1x1x1x1' with the '2nd 1x1x1x1' for its cube has a second non-commutative symmetry: '2nd 1x1x1x1' with the '1st 1x1x1x1', altogether, giving four distinct double char addresses simultaneously i.e., an 8:1 LDC. This satisfies $65{,}536^{4\ \mathrm{TTables}} = 1.84 \times 10^{19}$ unique combinations, or, 16 exabytes (EB) of grid rows. In case of columnar symmetry in two translation tables, $65{,}536^2$ = 4.1GB, handles the 16 EBs when column values are co-intersected by a *comparator matrix* in our code (residing in the LDD subprogram comparator). So, four 64K grid row combinations, handle the same EB values in four parallel tables. This requires complex matrix coding on an x86 machine. A 64-bit microprocessor, in principle, handles at most, 18 EBs of space [32], if based solely on 1 table. So, we program 4 tables to just have 32MB tables with our FBAR package. So beyond this limit, we run the FQAR model combined with the Bloch sphere [36] on a quantum computer, easing the complex matrix programming, to superdense the EBs down to the 64K limits of grid rows for incoming occupant chars.

Nowadays, compressors accumulate much more memory space, even more than 250 MBs for the highly ranked compressor (see Fig. 9). This is significant when *overhead information* and *memory caching* issues are studied from the usability aspect of the algorithm. The translation table, using memory cache, could be loaded into memory and accumulate much lesser space, which is significantly important for huge data transmissions, above TB limits on network and elsewhere, satisfying EB limits explained above.

The statistical test: The test involved the ranking of the data in the rows based on the selection criteria (former section), then comparing the mean rank in each column. Thus, the values of LDC would be ranked across each row as shown below. We derived these rankings collaboratively based on Fig. 9, Tables 4, 6-7 results.

Table 5. Current test case LDCs with ranks

Document	#	WinZip	GZip	WinRK	FBAR	FQAR
text	1	4; **3**	3; **2**	1; **1**	5; **4**	2
book1	2	3; **2**	4; **3**	2; **1**	5; **4**	1
book2	3	3; **2**	4; **3**	2; **1**	5; **4**	1
paper1	4	3; **2**	4; **3**	2; **1**	5; **4**	1
paper2	5	3; **2**	4; **3**	2; **1**	5; **4**	1
paper3	6	3; **2**	4; **3**	2; **1**	5; **4**	1
web1	7	3; **2**	4; **3**	2; **1**	5; **4**	1
web2	8	3; **2**	4; **3**	2; **1**	5; **4**	1
log	9	2; **2**	3; **3**	1; **1**	5; **4**	4
cipher	10	3; **2**	4; **3**	2; **1**	5; **4**	1
latex1	11	3; **2**	4; **3**	2; **1**	5; **4**	1
latex2	12	4; **3**	3; **2**	2; **1**	5; **4**	1

In Table 5, we consider the rankings to be valid relative to the fuzzy quantum version (the FQAR column), while if dismissed, we consider the ranks to be distributed between 1-to-4 instead of 1-to-5. This is applied to observe the four first columns from the left relative to FBAR, in **bold** values. Now we start testing

Decision rule: Reject $H.6_0$ if $F_r \geq$ critical value at $\alpha =$ 0.05 or 0.01, corresponding to 5% or 1% probability P. Otherwise, stay consistent with null hypothesis $H.6_0$.

Calculation method: The differences between the sum of the ranks is evaluated by calculating the Friedman test statistic from the formula

$$F_{\mathrm{r}} = \left[\frac{12}{nk(k+1)} \sum_{i=1}^{k} R_i^2 \right] - 3n(k+1), \qquad (7)$$

where k is the number of columns ('performance of algorithms'), n is the number of rows, and R_i is the sum of the ranks from columns. In compliance with our decision rule, the results on F_{r} which rejects $H.6_0$, are given in Table 7, since p-value < α. The critical *p-value* of F_{r} for {4 observed columns + 1 hypothetical column} and 12 rows at α = 0.05 or 0.01, is 0.0001. The distribution of the $F_{\mathrm{r}(4)}$ statistic is chi-square with k–1 degrees of freedom (df) or, df = 4. The test statistic F_{r} for all versions was 39.22. Without the FQAR, the result on F_{r} was 34. The p-value for the Freidman test is $P(F_{\mathrm{r(df)}} \geq F_{\mathrm{r\ observed}})$, the probability of observing a value at least as extreme as the test statistic for a chi-square distribution with df = 4.

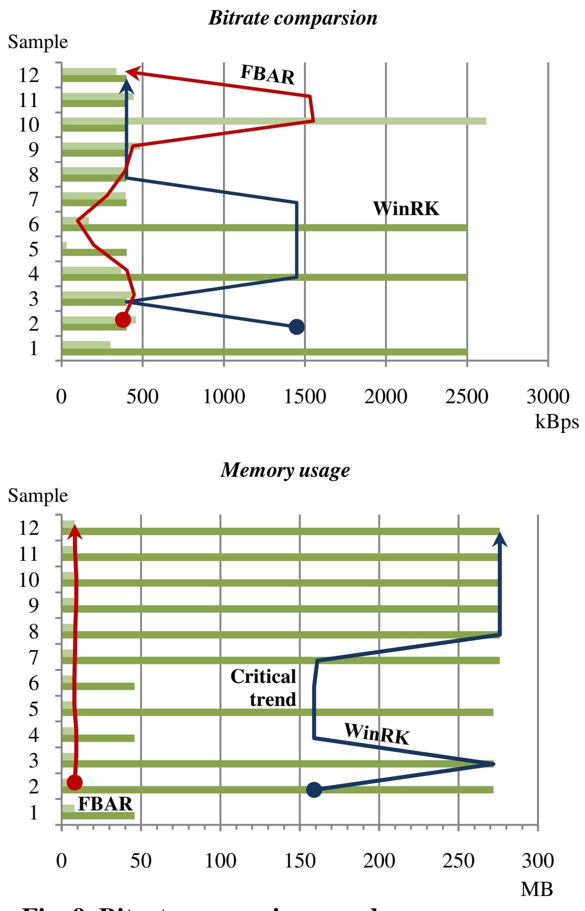


Fig. 9. Bitrate comparisons and memory usage

We thus conclude that, the bitrate and space saving performances have had a significant result on the LDCs for the randomly loaded documents compared to FBAR. By conventional criteria, the P-value = 0.0001 < 0.01 rejects $H.6_0$, since this difference is considered to be extremely statistically significant. Although, dismissing the column on FQAR results-in rank change on algorithms, we still observe P = 0.0001 < 0.01 rejecting $H.6_0$. Fig. 9 shows the bitrate and memory performance on 12 test documents, with their critical and optimal trends. The results are elicited from Table 6. The bitrate relative to memory usage was observed between the high and low ranked algorithms on 'space savings' (Table 4): WinRK vs. FBAR. As we can see, for higher bitrate performances, WinRK has a critical usage of memory per input sample. In some cases, even having 10 kBps for encoding and decoding data, required 800 MB memory on a 2GHz Athlon CPU. This ranks WinRK's memory performance lower than expected, as 4th, compared to FBAR. When we associate values of the upper chart with the lower chart, it is evident that the empirical data relative to memory usage on FBAR is optimal, and uniformly correlated except, the jump of bitrate on sample # 10. This is due to the excessive repetition of characters within the sample grid. The original input chars were ignored due to their pattern simplicity, forming simplistic forms of storable data. Therefore, the algorithm is not submissive for taking in too much information and thus its computation. The average base of bitrate was estimated 475 kBps for FBAR, and 925 kBps for WinRK on the 12 samples.

From the bar charts, it is possible to see that in some cases, there is already, right at the beginning, a major difference between the two results. There is also a difference observable at the end, where the mean coverage achieved by FBAR over memory usage is least critical than the mean coverage of the other compressor. This shows that there are significant differences between algorithms on their performances.

8. Discussion

The FBAR was tested in an experiment in which the outcome was compared with the results of other LDC algorithms. The string values were treated as binary during the encoding and lossless compression procedures. The strings were compressed into equivalent characters from the ASCII table into file **C**, thereby to a 4D grid file **G**. The grid file dimensions, each comprises of 16 fixed length code combinations, making 65,536 possible outcomes. From there, the translation table of the 95 printable and 1 nonprintable character block was used to make comparisons when the resultant document was converted back to binary for decompression. Table 6, shows the difference between all layers being implemented from the lowest layer(s) of lossless compression (LLLC) to the highest (HLLC).

The LDC time parameter is the result of (LLLC + HLLC) time t_L, measured in seconds. On the other hand, having the highest layer with only '1 byte sequencer' equal to '1', according to the example given on pseudocode sample II, gives optimum performance. In other words, in total, **C** = '1' in content, makes the interpreter to interpret '11111111' for the whole document, otherwise, '00000000' on the first char input encounter. From there, applying self-embedded flags, altogether performs good bitrates by comparison.

Table 6. Estimates on compression with rate performance on FBAR's LDC and LDD

		t_L = CPU time/s				Compressed size (bytes)	bits/char
File	Size (bytes)	LLLC	HLLC	LDC	LDD	(**C**+**G** files) – 64 K LLLC vs. HLLC	LLLC vs. HLLC
text	61608	0.18	0.02	0.2	0.24	31124.8+**1**: 7701.00	[2.1,2.4] : [0,2]
book1	678244	1.31	0.14	1.45	1.40	342654.5 + **1**: 84780.50	[2.1,2.4] : [0,2]
book2	1772074	3.2	0.75	3.95	3.43	895266.5+ **1**: 221509.25	[2.1,2.4] : [0,2]
paper1	52516	0.13	0.01	0.14	0.20	26531.5+ **1**: 6564.50	[2.1,2.4] : [0,2]
paper2	117493	0.26	0.115	0.375	0.32	59358.4+ **1**: 14686.63	[2.1,2.4] : [0,2]
paper3	10262	0.05	0.01	0.06	0.10	5184.4+ **1**: 1282.75	[2.1,2.4] : [0,2]
web1	747766	1.63	0.22	1.85	1.71	377777.6+ **1**: 93470.75	[2.1,2.4] : [0,2]
web2	598125	1.29	0.2	1.49	1.36	302177.7+ **1**: 74765.63	[2.1,2.4] : [0,2]
log	1840924	3.43	0.27	3.7	3.58	930050.1+ **1**: 230115.50	[2.1,2.4] : [0,2]
cipher	777654	0.25	0.04	0.29	0.32	392877.28+ **1**: 97206.75	[2.1,2.4] : [0,2]
latex1	209212	0.43	0.03	0.46	0.49	105695.6+ **1**: 26151.50	[2.1,2.4] : [0,2]
latex2	155641	0.42	0.03	0.45	0.92	78631.1+ **1**: 19455.13	[2.1,2.4] : [0,2]
translator	≈ 8 MB	N/A	N/A	N/A	N/A	N/A	2 bits/char read
Total	7021519	12.58	1.835	14.415	14.07	3547342: 877689.88	Avg. 2.25:1

Table 7. Rank sum and mean ranks via Freidman's test on the observed data

Document	**WinZip**	**GZip**	**WinRK**	**FBAR**	**FQAR**
sum of ranks	35	43	22	60	14
(sum of ranks)^2	1225	1849	484	3600	196

This is given by the additional byte (in **bold**) added to the HLLC column of the table.

We determine the limits of the application to be mostly on hardware constraints in design, rather than FBAR logic per se. To tackle this, we eliminated issues related to single bit usage of flags, considering their unique combination in **G** file is indeed avoiding 'bit array' models in programming. In fact, hard-coding 65,536 grid units via 'if loop statements', reading line-by-line with 95 printable char replacements, is more useful than the currently-available tools utilized for an x86 compiler. This enabled us to have all flags embedded in our marked-grid units by a standard char.

After verifying the theoretical estimates of 37.5%, 50% and 87.5% fixed size compressions, we began to compute the bitrate factor of our FBAR LDC. The result on randomly chosen documents for performing an LDD is listed in Table 6. The bitrate for both LDC and LDD relative to CPU time/s are computed and listed in the same table. We then included specific test results in form of Freidman's mean ranks and rank sum in recognition of hypothesis *H.6* of this paper (Table 7).

According to the sequencer approach mentioned above, it takes 5 to 6 levels of conversions with a CPU time t_L = {long + short + shorter + shortest} session to conduct all four FBAR LDC layers. Therefore, the HLLC version would practically engross one layer involvement during data computation. Hence, the logical results would give t_L on HLLC $\ll$ LLLC. This occurs relative to accessing the 'translation table' on 4×1-bit flags identity on each **G** row for an LDD.

9. Conclusion

In this study, we introduced and implemented FBAR logic, thereby evaluated its lossless compression ability compared to other known compressors.

We observed that almost every LDC uses probabilistic Shannon entropy as its 'logic base' in conducting lossless compression. However, we have also observed that our LDC performs fixed compression ratios, contrasting probabilistic standards of a typical LDC algorithm. Our LDC does not use Shannon codeword, and performs compression based on the new logic, FBAR. We thus conclude that, our algorithm contains predictable values due to a self-embedded flag structure for every double-character input.

The LDD FBAR simulation was tested on an x86 machine, under both UNIX and Windows platforms. Test samples as char-based documents, e.g., HTML, LaTeX and plain text, were examined for our prototype and compared with other compressors, varying from low, average to high ranks. FBAR achieves higher space saving percentages, above 50% as estimated, simulated and discussed in theory from its quantum state protocol. In the context of quantum information theory, the 50% compression is significant, proving double-efficiency on 16 bits transmitted via 8 quantum bits by our model. Similar percentages from other compressors never prove this efficiency regardless of their rank number.

Future generation computers, by using this model, e.g., combining the 4D grid model with the famous Bloch sphere in quantum information, could sustain a great deal of space and bitrate savings. We conclude that, this model could be considered as a solution to complex negentropy problems in information theory. This specifically concludes double-efficient values estimated greater than 87.5% e.g., 93.75% compression.

In terms of usage, a user would be able to know how much physical space is available within a reasonable time, before and after compression. This confidence in predictability makes FBAR a reliable version compared to the probabilistic LDCs available on the market. We finally conclude that this algorithm could be used in most aspects such as encryption, binary, fuzzy and quantum information technologies.

References

[1] K. Gödel, *Über die Vollständigkeit des Logikkalküls*. Doctoral dissertation. University Of Vienna. The first proof of the completeness theorem, 1929.

[2] D. Joiner (Ed.), 'Coding Theory and Cryptography', Springer, pp. 151-228, 2000.

[3] "Fuzzy Logic". *Stanford Encyclopedia of Philosophy*. Terms: **i-** Fuzzy logic, **ii-** Logical consequence. Retrieved in 2008.

[4] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, Vol. 23, pp. 337–342, 1977.

[5] J. Ziv and A. Lempel, "Compression of Individual Sequences Via Variable-Rate Coding," *IEEE Transactions on Information Theory*, Vol. 24, pp. 530–536, 1978.

[6] T. A. Welch, "A Technique for High-Performance Data Compression," *Computer*, pp. 8–18, 1984.

[7] L. A. Zadeh *et al.*, Fuzzy Sets, Fuzzy Logic, Fuzzy Systems, *World Scientific Press*, 1996.

[8] L. A. Zadeh, "Fuzzy algorithms". *Information and Control* 12 (2): 94–102, 1968.

[9] L. A. Zadeh, "Fuzzy sets". *Information and Control* 8 (3): 338-353, 1965.

[10] M. R. Titchener, "Compression Performance, Absolutely!", *Proceeding of the Data Compression Conference*, pp. 474, IEEE Comp. Soci., Washington, DC, USA, 2002.

[11] V. Engelson, D. Fritzson, and P. Fritzson. Lossless Compression of High-volume Numerical Data from Simulations. In *Proceedings of the Conference on Data Compression*. DCC. IEEE Comp. Soci., USA, 574, 2000.

[12] C. E. Shannon, The Mathematical Theory of Comm-unication, *Univ. of Illinois Press*, Champaign, 1998.

[13] C. E. Shannon, "A Mathematical Theory of Commun-ication". *Bell System Technical Journal*, Vol. 27, pp. 379-423, pp. 623-656, 1948.

[14] J. A. Patel, B. Cho, I. Gupta, Confluence: A System for Lossless Multi-Source Single-Sink Data Collection, Distributed Systems & Computer Networks, *Dept. of Comp. Sci., University of Illinois*, USA, pp. 1-3, 2009.

[15] C. E. Shannon, Redirected from EFF: Electronic Foundation Frontier group, Home database at: http://www.data-compression.com/index.shtml Accessed June, 2009.

[16] P. Viana, A. Gordon-Ross, E. Barros, and F. Vahid, A table-based method for single-pass cache optimization. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI, Cat. Cryptography and Architecture*, ACM, New York, NY, pp. 71-76, 2008.

[17] M. Czachor, "Notes on nonlinear quantum algorithms", Report No. *quant-arXiv.org*: ph/9802051v2, 1998.

[18] D. S. Abrams and S. Lloyd, "Nonlinear quantum mechanics implies polynomial-time solution for NP-complete and #P problems", *arXiv.org*: Report No. quantph/9801041, 1998.

[19] English text, *1995 CIA World Fact Book*, Lossless data compression software benchmarks/comparisons, Maximum Compression, at: http://www.maximumcompression.com/data/text.php, Accessed September, 2009.

[20] S. Zhang, Z. Li, Y. Liu, R. Geldenhuys, H. Ju, M.T. Hill, D. Lenstra, G.D. Khoe and H.J.S. Dorren, "Optical shift register based on an optical flip-flop with a single active element", *Proceedings Symposium IEEE/LEOS Benelux Chapter*, Ghent, 2004.

[21] C. D. Meyer, "Matrix Analysis and Applied Linear Algebra", *Soc. for Industrial and Applied Math.*, 2000.

[22] D. Schrader, I. Dotsenko, M. Khudaverdyan, Y. Miroshnychenko, A. Rauschenbeutel & D. Meshede, "Neutral Atom Quantum Register". *Phys. Rev. Lett.*, 93, 150501, 2004.

[23] P. A. M. Dirac. The principles of quantum mechanics (Fourth Edition ed.). *Oxford UK: Oxford University Press*. p. 18 ff, 1982.

[24] F. Pistono, "*Open Source implementations of encoding algorithms for video distribution in the HTML5 era*" (thesis project, 2009), Dept. of Comp. Sci., University of Verona, Italy, p. 2, 2010.

[25] I. Veldman, A. de Keijzer and M. van Keulen, Compression of Probabilistic XML Documents, Vol. 5785/2009, *Cat. Comp. Sci., Springer Berlin/ Heidelberg*, pp. 255-267, 2009.

[26] X. Xie and Q. Qin, Fast Lossless Compression of Seismic Floating-Point Data, *IEEE Comp. Soci.*, pp. 235-238, 2009.

[27] C. Bennett and S. J. Wiesner. Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states. *Phys. Rev. Lett.*, 69:2881, 1992.

[28] *CCSDS Green Book. Informational Report Concerning Space Data System Standards*, Lossless Data Compression, CCSDS 120.0-G-2, Sect. 3.1, 42 pp, 2006.

[29] R. Guastella, *Lossless Data Compression for Embedded Systems*, at: http://www.embedded.com/opensource /217800397 , Accessed April, 2010.

[30] J. Jacas and L. Valverde, *On fuzzy relations, metrics and cluster analysis*, at: http://dmi.uib.es/~valverde/gran1/GRAN1.html Accessed May, 2010.

[31] A. Hyvärinen and E. Oja, *Independent Component Analysis: A Tutorial, node14: Negentropy*, Helsinki University of Technology, Laboratory of Computer and Information Science, 1999.

[32] IBM, *A brief history of virtual storage and 64-bit addressability*, at: http://publib.boulder.ibm.com/infocenter/zos/basics/topic/com.ibm.zos.zconcepts/zconcepts_102.htm. Retrieved in May, 2010.

[33] G. Boole, *Cambridge and Dublin Mathematical Journal*, Vol. III, pp. 183-98, 1848.

[34] C. E. Shannon, *A symbolic analysis of relay and switching circuits,* Massachusetts Institute of Technology, Dept. of Electrical Engineering, 1940.

[35] S. W. Smith, The Scientist & Engineer's Guide to Digital Signal Processing, *California Technical Pub., 1st Ed.*, Chap. 27, 1997.

[36] D. Chruściński, "Geometric Aspect of Quantum Mechanics and Quantum Entanglement", *Journal of Physics Conference Series*, Vol. 39, pp. 9-16, 2006.

Appendix A

Appendix A gives a detailed description of the main components of FBAR, which were briefly introduced in the paper. These components are separated in Algorithm's structure (Section A.1), Data Compression (Section A.2), Data Decompression (Section A.3), Test Cases (Section A.4).

A.1 Algorithm structure

The Algorithm's structure comprises of program code, logic and process model. The algorithm itself could be viewed as **Algorithm = logic + control** by Kowalski in 1979 [35]. The logic component expresses the axioms that may be used in the computation, and the control component determines the way in which deduction is applied to the axioms. The axioms used for an FBAR compression are firstly classed as AND/OR Boolean logic, secondly, fuzzy logic, and finally, quantum logic. This combinatorial logic constructs the compression algorithm with control.

The *combinatorial logic synthesis* to retain FBAR was found by the current author, Alipour [2] in 2009. When {F, B, AND/OR} are conceived separately, each as a different field of calculus would have its own founder, i.e., chronologically, G. Boole, C. Shannon and L. Zadeh [44, 45, 5]. In 1938, Claude Shannon showed how electric circuits with relays were a model for Boolean logic. Hence, a sequence of 0's and 1's, constitute binary [45]. Therefore, in contrast, the current author questions that: *why not this binary is to be united with the highly probable states of quantum via fuzzy logic?* In fact, *is there a way to assimilate the discrete version F, B, A/R, into one unified version of all, FBAR*?

The combinatorial version, *uniting binary via fuzzy with quantum logic*, in this thesis report, is to prove the *relatedness* of these logical representations satisfying the posed question given as follows:

To maintain the 'control' aspect (first paragraph), the structure must include a process model, a model in which the logic is conducted cohesively and correctly through a program code. A slight change in the logical axioms in implementation would change the algorithm, making it irresponsive to its true logic conditions. This is the very characteristic required for a permissible entity to exist in the universe of discourse, either of being true or false in its *logical consequence*, outlining its structure. The FABR structure is cohesive in its states of logic and must be deductive (infer to a logical axiom) thus partitioned into finite forms of binary bits, no matter where we define them on a scale of time. In fact, we take every state to be reciprocal to its predecessor and its counterpart, firmly related in value and its quantifier: usually denoted by $\exists, \forall$, in this case, quantified with 'and' $\wedge$, and, 'or' $\vee$ operators.

The FBAR deductive system indicates that its logical consequences are validly true on FBAR's very structure when quantifiers are used in its language of the first-order AND/OR formulae (compare with Gödel's completeness theorem). We need to know this because of the relatedness of logic states, their *combinatorial logic synthesis*, when such logic is applied, rather than propositional calculus which returns such combinatorial logic as two unrelated propositions. The combinatorial logic with a true relatedness for at least two states is founded in the following reciprocal relation

$$\textit{binary states} \leftrightarrow \textit{fuzzy states} \leftrightarrow \textit{quantum states} \equiv \{0,1\} \leftrightarrow \{1 \searrow 0, 0 \nearrow 1\} \leftrightarrow \{00,01,10,11\} \quad (1)$$

A finite state representation for units of binary (a single bit) could be of use to present, not only its process model, in the major, its *data transmission architecture* when binary values are projected from one layer to another. Obviously, these states of binary are conditioned as 0 or 1, indicating a circuitry power level of some logic gate. The layers receiving binary states with respect to time to process them could be envisaged as memory layers where data abstraction and bitwise projections are handled. These issues are categorized as memory management over data relative to CPU usage (process time).

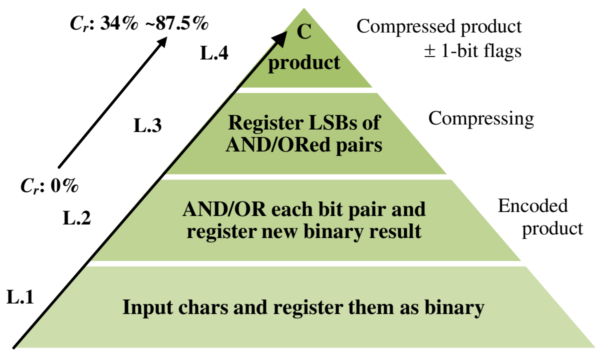

C_r: 34% ~87.5%
L.4
C product
Compressed product ± 1-bit flags
L.3
Register LSBs of AND/ORed pairs
Compressing
C_r: 0%
L.2
AND/OR each bit pair and register new binary result
Encoded product
L.1
Input chars and register them as binary

Fig. A.1: Basic structure of FBAR binary projections

The logic sates' relationships in the above model (Fig. A.1), are demonstrated in form of an experiment, which obeys the bit mapping procedure in a bitwise AND/OR processing system (Fig. A.2). In coding theory and cryptography [3], *superdense coding* is used to attempt an $2n$-bit binary transmission via $2n/2$ quantum bits. In the FBAR model, however, we demonstrate just that with absolute predictable states at its basic levels of LDC (or 50% compression). The confrontation of

Boolean logic with Fuzzy logic and Quantum logic in Rel. (1) is progressive between four compression layers (L.1-L4), i.e., a lower data layer of a certain encoded type is compressed to its upper through binary projections. In fact, 'fuzzy logic' reciprocally connects 'binary' with 'quantum logic' due to being a midpoint of the quantum version representing 8 probable entangled states {00, 01, 10, 11}, against the 2 probable binary states of {0, 1}. All of which, represent data as true or false depending where (in what space?) and when (what bitrate or frequency?) data is being processed. In this context, 'fuzzy logic' represents 4 states of logic, denoting the extremes and their middle points. The link between logical projections is established by simultaneously, 'in parallel', applying AND and OR to every paired set of bits from a binary sequence. We further elaborate on bitwise projections in the following subsections, relative to application design and resultants returned by the algorithm:

A.1.1 the FBAR architecture and technical expressions

FRBAR technical details rely on the following constituting terminology in terms of specific notations with definitions and related examples:

Notation	**Short definition**	**Example**
C_r	Data compression ratio.	2:1 compression
C	Compressed data; compression.	$C_{-1} > C_n$, $n \in \mathbb{N}$
C'	Decompressed data; decompression.	$C \xrightarrow{out \times ref} C'$
H	Entropy rate in e.g., Shannon systems.	$H_{\mathcal{A}} > H_{\wedge\vee(b)}$
m	String size e.g., English alphabet size.	$m = 26 + 1_{\text{space}}$
ch_i	A character, where $i \in \mathbb{N}$.	$\{ch_1ch_2ch_3\ldots\}$
x	Array dimension for the residing bits in memory.	if $x_i = 0$ then $x = [000\ldots0]$
y	Array dimension for projected bits from upper memory to lower layers.	if $y_i = \neg x_i$, $x_i = 0$ then $y = [111\ldots1]$
$x \to y$	Bits from horizontal plane projected vertically onto a compressed binary.	if $y_i = 1$ and $x_i = 0$ then $y = [0101...]$
λ	Variable lengths function on e.g., string, time or binary length.	$C(x) = \frac{\lambda(x)}{\lambda(t)}$
∞	Infinity; undefined, subject to removal via e.g., new characters.	$2ch - \infty = 2ch$
$\searrow$	Fuzzy state leaned to low level logic.	$1\searrow0 = \{0 , \approx0\}\equiv0$
$\nearrow$	Fuzzy state leaned to high level logic.	$0\nearrow1 = \{1 , \approx1\}\equiv1$
$\curvearrowleft$	Right bit-to-left bit selector.	$x_0 \curvearrowleft x_4$
$\curvearrowright$	Left bit-to-right bit selector.	$x_0 \curvearrowright x_4$
β	Binary vale or sequence, where $\forall\beta \in f(x) = x \to y = b$	$x_0 \curvearrowright x_4x_5 \curvearrowright x_1 = \beta$, if $\forall x_i = 0$, $x_5 = 1$, $\therefore \beta = 0001$
Δ	Change of… ; difference	$\Delta m = m_2 - m_1$
$\wedge$, &	Logical AND otherwise, bitwise AND	$0\wedge1=0$, $1\wedge1 = 1$
$\vee$, $\vert$	Logical OR otherwise, bitwise OR	$0\vee1=1$, $0\vee0 = 0$
$\leftrightarrow$	Bidirectional between states or logic	$x\leftrightarrow y \equiv x \to y \to x$
$\equiv$	Equivalence; identical to …	2 chars $\equiv$ 16 bits
$\therefore$	Logical deduction; therefore …	$\{ch_1ch_2\} = \{\$\%\}$ $\therefore$ $ch_1 = \$$, $ch_2 = \%$

With regard to these notations and their definitions, we must also familiarize ourselves with the conceptual aspects of FBAR in detail prior to implementation, since both aspects are in conjugation with delivering an FBAR compression product to its user.

A.1.1.1 Aims and objectives

The aim of this study is to find out how FBAR compression is applied to current and future generation computers. We shall chronologically establish our new technique as follows:

In our design and development:

- Develop the compressor in form of a prototype with the FBAR technique into four layers. We reflect these layers as FBAR logic set, projecting one logic state to another.
- Develop the prototype by a programming language like C, due to its better memory management addressing efficiency, such as the *bit field* usage over *bit arrays*, customized in other languages.
- Develop the prototype complying with the descriptions presented in Table A.4.
- Develop the prototype following the Table A.1 flow in sequence during implementation.
- Make relevant ASCII and binary conversions from one another as we encode data and compress them through FBAR logic, generating predictable data compression ratios.

- Implement single bit flags on memory attributes, representing the compressed bit address and its polarity for data compression layers. (This is our main focus.)
- Retain the retrieved data from memory to reconstruct ASCII characters (decompression).
- Generate lossless C_r values based on FBAR logic only, contrasting frequent Shannon entropy or probabilistic methods used today. (An example on Shannon is given by Ref. [18]).

In our analysis:

- Compare the algorithm's data compression ratio with others known in the market such as: GZip, WinZip, WinRK , etc. These lossless compressors [22] present *an ideal maximum compression* value, which can *vary* from data-to-data and from file-to-file.
- Elaborate upon the obtained C_r results and compare them with the results coming from other compressors no matter how variant. (For our metric comparisons, we use the direction stated on our hypothesis *H.6* in the upcoming section.)
- Compare results, and apply their deduction to the evaluation phase of the algorithm.

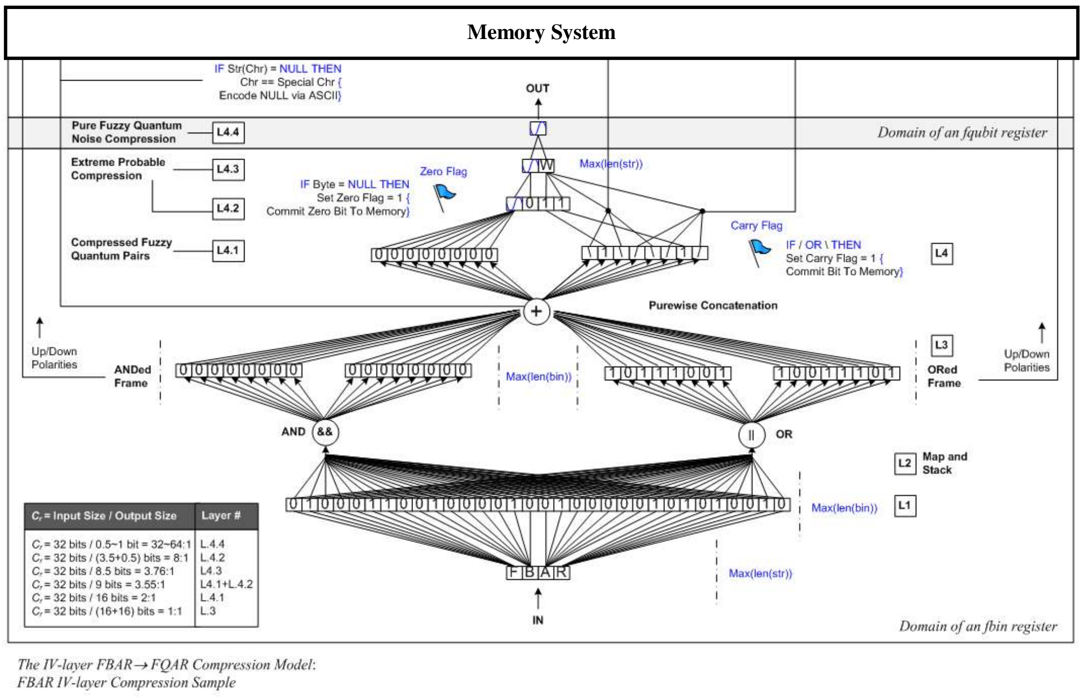
For product evaluation:

- Validate whether FBAR is of a reliable type of algorithm (lossless?), where the input data after compression must be as same as the output data when decompressed.
- Run a set of testing protocols on every step of the design i.e. the "entropic" analysis of the FBAR closed system. This is vital for our quantitative measurements on the layers, *assuring no data loss or deterioration during algorithmic data transformations*.
- Count the exact bytes of information, where each byte consists of 8 bits ≡ 1 character.
- Validate byte count results by collecting samples of different types of data (in which other compressors vary in producing compression values).
- Build the empirically-executable tests on the conceptual aspect of FBAR, addressing the compressor's data I/O products from background.
- Identify the binary, fuzzy and quantum forms of data (as qubits or bit constituents) for either compression or decompression phase of the algorithm. The decompression phase should perform the exact amount of data with quality via these forms of logic.
- Check whether the resulted data, no matter its level of compression from input A to output B over time, relative to complements A' and B' decompression outputs, is lossless.

A.1.1.2 Research Methodology

Our research mainly focuses on experiment based on current affordable techniques, both hardware and software, i.e., how the data is obtained and compressed losslessly, using relevant tools of experimentation. However, for evaluating the algorithm's data compression and performance, we compare it with other data compression tools based on comparison statistical methods. In this regard, we have sectionalized our research approach as follows:

- Experimental setup by having relevant programming packages installed on a computer, thereby coding, compiling and testing our software on an operating system. We follow our design platform, functions and algorithmic flow from Tables A.1-4. For a progressive experiment, we then run a metric software analysis for data compression comparisons.
- Implementing the designed algorithmic technique and test it in practice.
  Description: This is done through prototyping, coding and simulation, satisfying hypotheses *H.1*-to-*H.6* from the upcoming section: The *coding* is used for building our *prototype*, testing FBAR logic for current computers. Its successful presentation would be a steppingstone to establish *superdense coding* [34], achieving FQAR standards.
- Analysis of different methods and introducing the utilized solution for the algorithm through spreadsheet, mathematical and 'software evaluation' packages. We further, through statistical analysis of compression ratios, collect values that come from our experiment. These values are expected to come from different data samples i.e. arbitrary documents in which other compressors vary in producing compression values.
- Validating the fixed data compression ratio for our samples relative to data integrity. We conduct experiments with different data samples on such parameters.
- Evaluating and comparing the results obtained from the experiment i.e. FBAR-output products over data compression-decompression relative to data rate (further analysis).


Memory System
IF Str(Chr) = NULL THEN Chr == Special Chr { Encode NULL via ASCII}
OUT
Pure Fuzzy Quantum Noise Compression
L4.4
Domain of an fqubit register
Extreme Probable Compression
L4.3
L4.2
Max(len(str))
Zero Flag
IF Byte = NULL THEN Set Zero Flag = 1 { Commit Zero Bit To Memory}
Compressed Fuzzy Quantum Pairs
L4.1
Carry Flag
IF / OR \ THEN Set Carry Flag = 1 { Commit Bit To Memory}
L4
Purewise Concatenation
Up/Down Polarities
ANDed Frame
Max(len(bin))
L3
ORed Frame
AND
&&
||
OR
L2
Map and Stack
L1
Max(len(bin))
Max(len(str))
IN
Domain of an fbin register
The IV-layer FBAR→ FQAR Compression Model:
FBAR IV-layer Compression Sample

C_r = Input Size / Output Size	Layer #
C_r = 32 bits / 0.5~1 bit = 32~64:1	L.4.4
C_r = 32 bits / (3.5+0.5) bits = 8:1	L.4.2
C_r = 32 bits / 8.5 bits = 3.76:1	L4.3
C_r = 32 bits / 9 bits = 3.55:1	L4.1+L.4.2
C_r = 32 bits / 16 bits = 2:1	L.4.1
C_r = 32 bits / (16+16) bits = 1:1	L.3

Fig. A.2: The FBAR architecture and pairwise AND/OR relationships (an expansion to Fig. A.1)

Foremost, we must conceive how 'bits and bytes' are in correlation with each other in terms of their logical consequence, coefficients between their rise and fall of states, previously explicated for Eq. (1). Their consequences of "logic states" are mainly understood in their relationships within the FBAR architecture. The FBAR architecture consists of a memory system which could be portable in form of 'reference tables' or a 'dictionary' per input information. The "input information" is the valuable data that we input to the FBAR program before a lossless data compression (LDC). From there, the valuable data by its substitutes, later known as 'character occupants', reconstructing the original information from the availably-compressed data becomes plausible in practice. This latter phase of FBAR operations is called a lossless data decompression (LDD). It is evaluated in terms of information entropy, assuring that whether data during the levels of LDC and LDD, has been lost otherwise successful per compression session. In recognition of Fig. A.1, its expanded version, Fig. A.2, gives an overview of the four-layer algorithm from bottom-to-top with an input string. When attaining levels of output at the top, the original string is compressed into 1 signal length, or, a value of 1-bit. (Extracted and adapted from Alipour [2]). The main operators during paired bitwise (*pairwise*) conversions are '+' for concatenating chars or their string(s), '&&' for logical AND, '||' for logical OR conditions. However, one must not confuse the latter two with bitwise AND/OR.

We convey with the logical AND/OR on the basis of input data "*which could be of integer type conditions converted from two or more, number of chars in a variable*," returning integer values which are zero (false) and nonzero (true). However, focusing on *single bit chars* compared to *integers* from one layer of bitwise-paired projection to another, requires the use of bitwise operators '&' for AND, '|' for OR, in bit fields as low level operators for 1-bit flags, retiring 1 or 0 Boolean values. The '&&' and '||' usage for both logical and bitwise operations are applicable, since in this case, we are pairwising the bit chars, or, '&, &'ing '|, |'ing them, depending on, in which programming language we implement the algorithm, thus evaluating their converted input sequences of data. To implement these bitwise projections per logical input condition i.e., "pairs of bits are ORed and ANDed, thereby evaluated for a concatenation procedure according to Fig. A.2," we follow the steps from our contextual algorithmic flow. This flow targets on an efficient and fixed size compression product for x86 machines. The flow constitutes FBAR methods and its algorithmic components as follows:

A.1.1.3 Methods and components

The definition of FBAR methods can be distinguished into the Fuzzy Binary AND/OR constituents' methods (or logic constructor methods), and memory flag methods for components addressing data reconstruction:

***Convert** string character to binary → **select** bit pairs from head to tail of the binary sequence → **apply** AND-OR logic to each pair, thus an encoded message is generated → **select** bit pairs from head to tail of the encoded message → **raise** 1-bit flags for those that are impure and those with unique polarities (from table A.5) → **compress** impure and pure pairs in form of single bits, thus a minimum compression based on FBAR achieved ↔ **further compression** requires complex memory mappings and sub-bit projections ↔ **ultimate compression** requires sub-bit projections onto refreshable signals in a fuzzy qubit register → **decompress** data by dereferencing flags plus 1-bit insertion per compressed bit → **reverse** sequence from the stack for proper binary sequence equivalent to ASCII characters → **decode** message by recalling bit position and polarities between the previously ANDed and ORed data (based on Table A.5 and memory contents) or, a static recall of bits based on the bit flag hypercube model (like Table A.6) → **decompress** data by converting binary to ASCII obtaining the original message.*
Legend: ↔ stands for "revert back to previous state" otherwise, precede the next step; → stands for "continue with the next step"

Table A.1: A finite flow table on the FBAR algorithmic implementation.

For our step-by-step experiment addressing the above flow, we devised a set of alternative hypotheses with their null hypotheses testing our algorithm as specified in the following table. For conducting each of them, a question was also formulated:

H.1- Input of any data type to the FBAR's 1st layer, results in binary representing the same original content.
H.1₀- The conversion of any data type to binary is impractical.
Directionality over H.1- Input of ASCII data to the FBAR algorithm, results in binary representing the same original content.
Q.1- Is the conversion of any FBAR input data to binary possible?

H.2- A sequence of pairwise selection of binary to the FBAR's 2nd layer, when a parallel *and-or* applied, results in an encoded binary message in the 3rd layer.
H.2₀- The pairwise selection and *and-or* operation on a binary sequence, is firstly *H.1* dependent, and secondly, irreversible for data reconstruction even in case of implementation. (Or, backtracking to the original message is impractical.)
Directionality over H.2- A pairwise selection of bits from a binary sequence and applying *and-or* logic on every pair, results in a definitive encoded message.
Q.2- Is it possible to conduct the pairwise selection and *and-or* application on a converted data? If conducted, is the process reversible?

H.3- A sequence of pure and impure pairwise selection of binary to the FBAR's 3rd layer, once detected and replaced with single bits, results in a compressed message in layer 4.
H.3₀- The pure and impure pairwise selection and compression to single bits on a binary, is firstly *H.1* and *H.2* dependent, and secondly, irreversible for data reconstruction even in case of implementation.
Directionality over H.3- A pure and impure pairwise selection of bits from a binary sequence and compressing every pair to either 0 or 1 logic, considering impure pairs with single bit flags, results in a compressed message.
Q.3- Is it possible to conduct a pure and impure pairwise selection, thereby its compression into single bits on an encoded data from the previous layer? If conducted, is the process reversible?

H.4- A sequence of single bit flags representing compressed data in FBAR's 4th layer, once reused adjacent to other purely compressed 1-bit data, results in a decompressed message from layer 4.
H.4₀- The sequential recall and reuse of bit flags from memory/grid, is firstly *H.1*, *H.2* and *H.3* dependent, and secondly, unachievable for an identical data reconstruction even in case of implementation.
Directionality over H.4- Proper setup and storage of 1-bit flags *before reaching* layer 4 compression in memory.
Q.4- Is it possible to conduct a pure and impure bit sequence reconstruction, thereby a decompression of multiple bits from the final compression layer? If conducted, is the output data identical to its original?

H.5- A sequence of compressed data in form of *H.3*, when equipartitioned and paged into memory or confined signals in information space/grid, results in a maximum compression possible > 87.5% in layer 4.
H.5₀- The compression of any data length into one single bit is firstly *H.1*, *H.2* and *H.3* dependent, and secondly, unmanageable and irreversible for data reconstruction like *H.4*, even in case of implementation.
Directionality over H.5- A simulation or minimum conduction of this model by introducing the upper limits of FBAR as FQAR via signal processing and quantum memory architecture.
Q.5- Is it possible to integrate in scale the algorithm to an ultimate single-bit compression? If implemented/ simulated, is the process manageable and reversible for data reconstruction?

Table A.2: The FBAR systematic hypotheses

We tested hypotheses *H.1-H.4* with a 'dry run' for the algorithmic implementation. So this experiment would examine the algorithm's code with mathematical compression values. Hypothesis *H.5*, however, holds good in upgrading the technique for future quantum computers. Apart from *H.5*, all the above hypotheses are testable through feasible experimentation, since they contain a testable logic in binary, fuzzy and quantum according to Rel. (1) and Table A.1 flow.

Let *X* contain our FBAR technique as well as a selection of state-of-the-art compression techniques. Furthermore, let *Y* contain a representative sample of documents of different type. Therefore,

H.6- A difference exists in the performances of the techniques in *X* as measured on *Y* by computation rate and space savings.
Directionality over H.6- Perform the test using (non-)parametric methods, and compare the difference.
H.6₀- The difference in performances of the techniques in *X* as measured on *Y* by computation rate and space savings is zero.

We have further addressed the following hypothesis using Freidman's test to evaluate our algorithm compared to other LDC algorithms used today, by extending Table A.2 to a final hypothesis for LDC comparisons (*H.6*). Hypothesis *H.6* has been covered in Appdx. B, with a relevant preamble given to it in §A.4. To implement the flow from Table A.1, we had to implement the hypotheses relative to their nulls, putting each null into perspective of our implementation. As we can see, each hypothesis supports its subsequent, and thus consistent with the implementation to reject its null. If a null hypothesis is not rejected or tackled with, the algorithm is of an unsuccessful application.

Of course, this was a risk that we needed to take into consideration for our implementation. Since we have had noticed that FBAR logic is unequivocally solid in its representation(s), we thereby simulated its objectives for an LDD implementation. The reason is it required an extensive number of lines of code to program, converting its 4D grid model 'if-else' conditions, to a *fully-correct readable memory grid* (a file), which is static in size, and portable from one computer to another.

Obviously, planning this risk in its infancy is totally eliminated from the list of risks, since FBAR is provable, not only on its conceptual, also, on its implemental level, whereas the latter requires more time and manpower to fully implement the algorithm covering LDCs of 87.5 % compression on x86 machines. The current version, however, guarantees 50% pure compression with a default sequencer of '1', to manipulate its values through a grid file (*the portable memory grid*), containing self-embedded 1-bit flags. This sequencer is later known as e.g., 1 = '1111…1' pure binary for the whole number of available characters of the original file, before its compression (discussed in § A.1.2.3). We first begin with the improved version of Table A.1's flow, presented in form of a short flow pseudocode as follows:

Pseudocode main sample: ***an FBAR lossless data compression and decompression***

```
WHILE reading INPUT CHARACTERS from STRING DO
STRING = 8 BIN CHARACTERS
BITWISE AND 1st 2 BIN CHARACTERS, 3rd 2 BIN CHARACTERS
BITWISE OR 2nd 2 BIN CHARACTERS, 4th 2 BIN CHARACTERS
OUTPUT STRING = 4 BIN CHARACTERS for AND + 4 BIN CHARACTERS for OR
IF STRING OUTPUT = impure '01' OR '10' THEN
STORE 1-bit flag for '01'
STORE 1-bit flag for '10'
ELSE
STORE 1-bit flag for '00'
STORE 1-bit flag for '11'
END of IF
CLOSE 1st pair of BIN CHARACTERS = rightmost CHARACTER of the pair
GOTO IF for this new condition
OUTPUT STRING = rightmost 1st BIN CHARACTER
CLOSE 2nd pair of BIN CHARACTERS = rightmost CHARACTER of the pair
GOTO IF for this new condition
OUTPUT STRING = rightmost 2nd BIN CHARACTER
CLOSE 3rd pair of BIN CHARACTERS = rightmost CHARACTER of the pair
GOTO IF for this new condition
OUTPUT STRING = rightmost 3rd BIN CHARACTER
CLOSE 4th pair of BIN CHARACTERS = rightmost CHARACTER of the pair
GOTO IF for this new condition
OUTPUT STRING = rightmost 4th BIN CHARACTER
CONCATINATE STRINGS = 1st + 2nd + 3rd + 4th single BIN CHARACTERS
OUTPUT STRING = 2 BIN CHARACETRS for AND + 2 BIN CHARACTERS for OR
CONTINUE CLOSE on 4 BIN CHARACTERS
...
OUTPUT STRING = 1 BIN CHARACTER
STORE STRING in FILE as COMPRESSED RESULT
STORE 8x1-bit FLAGS as single ASCII CHARACTERS representing STRING COMPRESSED RESULT
in FILE
END of WHILE
WHILE reading COMPRESSED FILE for DECOMPRESSION DO
COMPARE 1-bit FLAGS from FILE with 1-bit FLAGS in translation table or DICTIONARY
RETURN 2 4x1 bit FLAGS from DICTIONARY as 2 or more NEW CHARACTERS
NEW CHARACTER = OLD CHARACTER of ORIGINAL FILE
OUTPUT STRING = STRING + NEW CHARACTER
OUTPUT STRING = ORIGINAL CHARACTER
END of WHILE
```

As we can see, the main pseudocode begins with compressing data by ANDing and ORing while raising 1-bit flags in a bit-field, which aims to avoid extraneous memory space allocation(s). As we shall see later, this falls into *bit fields* vs. *bit arrays* category, whereby the author, beyond the both coding concepts, discovered a *self-embedded 4×1-bit flags approach in a new grid model* (later discussed in § A.1.2). The grid model is part of the compressed file, a product to be compared with the 'dictionary coder/decoder' for lossless decompression purposes. The program compares the flags within a set of rows as self-embedded flag addresses with the ones in the translation table as the main component of the dictionary. Once a flag comparison is done, then data reconstruction in the new file begins by writing character-by-character for each newly-constructed row into it, identical to the original file which is now unavailable. These relevant functions are defined within the '`DECOMPRESSION`' subroutine as an interpreter of the program for the growing string i.e. '`OUTPUT STRING = STRING + NEW CHARACTER`' from line # 37 of the pseudocode. The implementation of the latter is pointed out in the range of line # 33 to 39. Gradually, the bitwise conversions over unsigned characters as '`BIN CHARACTERS`' (in case of programming in C), are compressed when packing the characters in terms of their closures i.e. the ending state of fuzzy or logic within each pair of impure 10, or 01, and pure 11, or 00. Meaning that, the `CLOSE` of a pair 01 results in high state logic or '1', and '10' results in low state '0', for 11, a '1', and for '00' a '0'. In C, this could be done by using the `mask()` function, shifting characters as our '`BIN CHARACTERS`' from right to left '<<' and from left to right '>>' for a specific character. Once we attain the right character as the rightmost character for each pair, we output '`STRING = rightmost BIN CHARACTER`'. In continue, once we reiterate the if condition by visitng and revisitng its conditions after each `CLOSE` made on '`BIN CHARACTERS`' for each LDC layer (line # 13 to 28), we then could say, a compression prior to the encoding levels (line # 2 to 12) has occurred. In the following subsections, we show the collapsed versions of the current pseudocode:

- The basic collapsed version: Methods as the overall structure of the code (current section).
- The expanded collapsed version: Functions and arguments as the modular structure of the code (§ A.1.2.1).
- The specialized collapsed version: Specific conditions as the nodal structure of the code (§§ A.2 and A.3).

The "basic collapsed version" highlights the methods like the above pseudocode; the "expanded collapsed version", highlights functions and arguments; finally, the "specific conditions of the collapsed version" or "specialized collapsed version", highlights if-else and counting conditions on loops and nested loops in the program, coded in terms of e.g. If, For, Switch Cases, functions calls in the code for an LDC and LDD subroutines. Showing this part later as other pseudocode versions is due to establishing the facts of the returned lossless compression results by the algorithm in §§ A.2, A.3 and Appdx B. For achieving these results, we implemented the relevant mathematical operations via loops, conditioning memory data transactions and management elements over user's I/O data.

In continue, by recalling the revertive states coming from Table A.1's flow (denoted by a ↔), is subjective to simulate the quantum hardware for future applications that supports maximum compressions of FBAR, via signal processing and quantum information techniques (Appdx. C). However, this is not essential when we focus on the basic four layers of FBAR, producing a compression ratio of fixed values below and greater than 2:1. The current experiment focuses on the functionality of the technique itself i.e., the FBAR's logic model.

Main function(s)	**Helper operator:** ***Comment***	**Operates on… :** ***Comment***	**Specific task**
`Len()`	For Loop: Demarcate a series limit for function's iteration by an integer	txt: this is a text as string; bin: this is a binary sequence	Returns the precise length of a string or binary
`Mid()`, `Left()`, `Right()`	For Loop: Demarcate a series limit for function's iteration by an integer	txt: this is a text as string; bin: this is a binary sequence	Locates a specific n^{th} character of the string or binary from head or tail sequence for *encoding* purposes.
`LongToBinary()`	Asc(strChar): this is a helper function with a conversion operation over a string character	strChar: this is a string character equivalent is Chr$(#), where # is a decimal number.	Converts long value into a binary string.
`BinaryPair()`, `Cat()`	& or +: Concatenation	bin: this is a binary sequence	Displays binary pairs or duals when necessary
`BinaryAND()`, `BinaryOR()`	Bitwise AND: And logic, Bitwise OR: And logic	bin: this is a binary sequence	AND/OR two binary values or bits
`Replace()`	& or +: Concatenation; /: high level polarity or 1 closure, or, impure 01 logic; \: low level polarity or 0 closure, or, impure 10 logic	strChar: this is a string character bin: this is a binary sequence	Replaces a non-binary or binary character with an ASCII otherwise binary character
`Rev()`	/: high level polarity or 1 closure, or, impure 01 logic becomes \: low level polarity or 0 closure, or, impure 10 logic becomes the former	txt: this is a text as string; bin: this is a binary sequence	Reverses a sequence of string or binary for special projections (*mostly binary in form of pairs and nibbles*)
`varPtr()`	Addr: Memory address in form of e.g. base address 0x0 flag paging	regA, regB: memory register A contains bit position and binary (*key identifier*); memory register B is for bit polarity and bit address	A *variable pointer* gets a pointer to memory variable that allocated position for a char or a single bit if any
`DeRef()`	Case *flag #*: Apply case for a function's specific polarity flag (#: 0-to-8 possible polarities or a total of 9 cases)	varPtr(regA) and varPtr(regB)	Dereferences data to allow memory content read for decompression

Table A.3: The main functions in .Net or VB for an FBAR algorithmic simulation.

We have elicited from the findings on FBAR [1], the above and thereby the following programmable functions in coding theory, adaptive to the data structure of FBAR for basic to maximal compressions.

The transformation of FBAR to its highest levels of compression within its four-layer encodings is done via qubit registers. In this regard, a seclusive proposal in § 3.5.1 [2], is given for their new hardware design principles. The design in theory, with its practical aspects of an *n*-fqubit register, is briefly outlined in Appx. C. In the classical version on current computers (x86 machines), however, the 'main challenge' is to implement the FBAR's 4th layer projections commencing with an 8-bit to 5-bit iterative compression, which yields a 37.5% compression. It is evident with a fixed sequencer of either 1 or 0, representing 11111111, and 0000000 respectively. Therefore, it would generate an 8-bit to 4-bit compression on x86 machines, denoting a 50% pure 'space savings'. This fact is already elucidated, when properly programmed according to FBAR pseudocodes. Further challenges meet those compression values generated from 2*n*:1 ratios for *n* > 3 hypothetical values.

Coded function(s)	**Helper operator:** ***Comment***	**Operates on… :** ***Comment***	**Specific task**
`length()`	For Loop: In the loop, limit for function's iteration by a length variable as Integer	txt: this is a text as string; bin: this is a binary chars length	Returns the precise length of a string or binary chars
`mask()`	For Loop: In the loop, limit input string iteration by a left shift '<<' or right shift '>>' mask from one bit char to another. Gives also: Ascii to BinChar: this is a helper function with a conversion operation over a string character	txt: this is a text as string; bin: this is a binary char	Locates a specific n^{th} character of the string or binary from head or tail sequence for *encoding* purposes or Char-to-BinChar *conversions*
`gridWrite()`	Addr: Memory grid address in form of e.g. a base address 1x1x1x1 flag paging in a portable file called "grid"	txt: this is a text string data reconstructed as such in a new file	A *file pointer* writes the matched characters for each grid read (grid file), now written to a new fie.
`gridRead()`, `deRef()`	Case *flag #*: Apply case for a function's specific 1x1x1x1 flag location in a dictionary file called "dic"	Pointer **a* and **b* : dic file contains bit position and binary (*key identifier*); pointer is used for similarities if spotted in grid row # char pattern checks, between grid file and dic contents.	Dereferences double or more original chars from the hardcoded 4x1-bit flag data representing occupant chars (position) to allow the right char reconstruction.

Table A.4: Main functions coded in C for an x86 implementation or simulation.

The FBAR methods, arguments and function calls, as a whole, must obey the following pseudocode, as presented earlier, which is the *basic collapsed version*. Here goes the basic version enabling the execution of a `fuzzyFlag` constructor

```
WHILE maksing BIN CHARACTERS from BITWISE AND and BITWISE OR results DO
STRING = 8 BIN CHARACTERS
ASSIGN '0' to a DOWN variable
ASSIGN '1' to an UP variable
FLAG_STRING = UP + DOWN CHARACTERS
IF FLAG_STRING = (DOWN + UP + UP + DOWN)
CHARACTERS THEN
STRING = (MSB BIN CHARACTER + 5th BIT
CHARACTER) + (6th BIT CHARACTER + 2nd BIT
CHARACTER) + (7th BIT CHARACTER + 3rd BIT
CHARACTER) + (LSB BIN CHARACTER +
4th BIT CHARACTER)
OUTPUT STRING = OLD 8 BIN CHARACTERS
ELSEIF CONTINUE CONCATINATE for other
FLAG_STRING UP + DOWN combinations
...
END of IF
OUTPUT RESULTS from BIN CHARACTERS to ASCII as 8 BIN
CHARACTERS = 1 ASCII CHARACTER
END of WHILE
```

Fig. A.3, shows the basic structure and the main components of the FBAR prototype. In this figure, the system starts by receiving an input string for preliminary conversions as illustrated in Fig. A.2, starting with *and-or* logic.

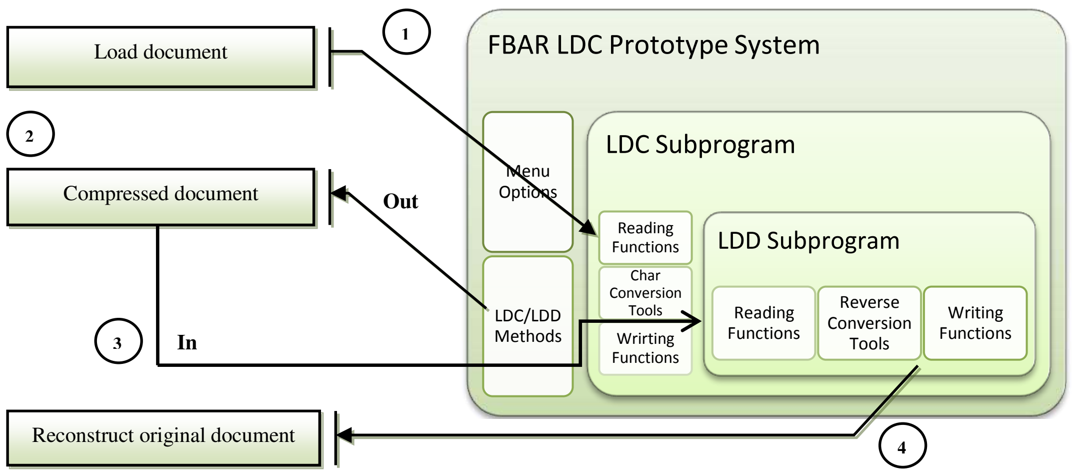


Fig. A.3: Structural components of the FBAR prototype with simplistic process states

The starting point is by choosing the relevant 'menu option' executing one or more of the hypotheses *H.1-H.4*:

1- the pairwise selection of bits after converting each character in sequence, the encoding of AND/OR process,
2- high state and low state fuzzy binary conversions, and
3- the **G** file (grid file) commitment over compressed bits for an LDC by raising 4×1-bit **znip** (*zero*, *negate*, *impure* and *pure*) flags,

are the main tasks of this prototype. These tasks are outlined as 'conversion tools' in Fig. A.3, which all is clearly explained in the following sections.

A.1.2 Lossless Data Processing

From the previous section we have studied the algorithmic components, logic and its architecture in aim of proving the possibility of its implementation grounds on LDC and LDD inclusively. In this section, we study FBAR's LDC process. The following figure (Fig. A.4) shows a 'circular process' of an FBAR LDC, a combination of the algorithmic design and program's process model, whereas the latter comprises of functions, methods , etc. as propounded previously.

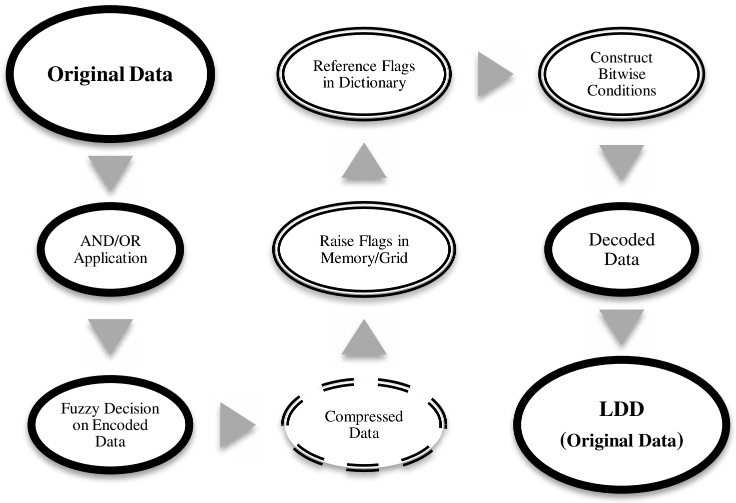

Original Data
AND/OR Application
Fuzzy Decision on Encoded Data
Compressed Data
Raise Flags in Memory/Grid
Reference Flags in Dictionary
Construct Bitwise Conditions
Decoded Data
LDD (Original Data)

Fig A.4: The circular process of an FBAR LDC comprised of program design and memory

Fig. A.4 represents a 'circular process' of an FBAR LDC with *dictionary*, a combination of the algorithmic design and program's process model. The process comprises of program design and memory transactions with the usage of relevant functions and methods coded in C.

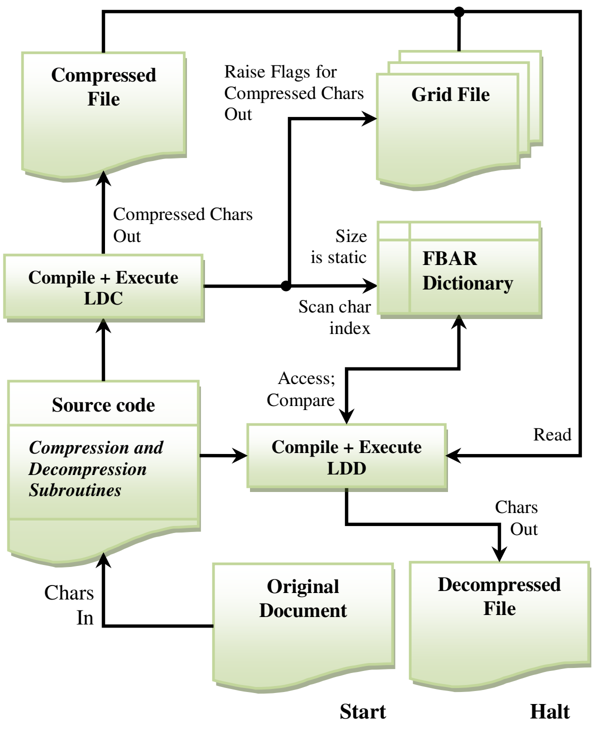
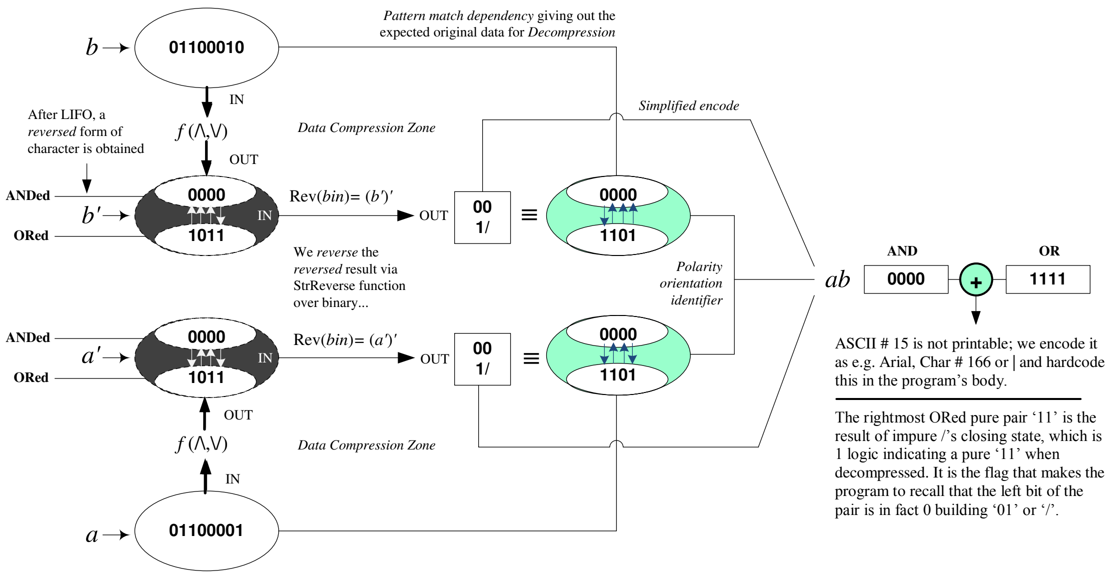


Fig. A.5. Basic process design of FBAR binary I/Os

To conduct a successful data decompression, we renounce bit values based on a predictive pattern of bits in a memory structure. This occurs subsequent to the double-dashed circle process component in Fig. A.4. We constructed a 'char and binary' LDC reference table to satisfy these conditions during the compression phase of the algorithm. The conditional output per character input subsists on relevant bit-flags and extended bits that are allocated in the memory.



b
01100010
Pattern match dependency giving out the expected original data for Decompression
IN
Simplified encode
After LIFO, a reversed form of character is obtained
f (Λ,V)
Data Compression Zone
OUT
ANDed
b′
ORed
0000
1011
IN
Rev(bin)= (b′)′
OUT
00
1/
≡
0000
1101
We reverse the reversed result via StrReverse function over binary...
Polarity orientation identifier
ab
AND
0000
+
OR
1111
ANDed
a′
ORed
0000
1011
IN
Rev(bin)= (a′)′
OUT
00
1/
≡
0000
1101
ASCII # 15 is not printable; we encode it as e.g. Arial, Char # 166 or | and hardcode this in the program's body.
The rightmost ORed pure pair '11' is the result of impure /'s closing state, which is 1 logic indicating a pure '11' when decompressed. It is the flag that makes the program to recall that the left bit of the pair is in fact 0 building '01' or '/'.
OUT
f (Λ,V)
Data Compression Zone
IN
a
01100001

Fig A.6: The FBAR data compression and decompression model for two characters '*a*' and '*b*'.

The allocation, read/write and reference process is shown in Fig. A.5, representing three major procedures to reconstruct data during an LDD. The process design and the development of the algorithm are illustrated in Fig. A.5. As illustrated in this figure, an FBAR dictionary consists of a *translation table* (later inducted as Table A.7), and a *reference table* (later inducted as Table A.8), both building a static size of flag information, later used by the program's interpreter for char comparisons. The allocation of the raised single bits for the minimum compression phase of FBAR is shown in Fig. A.4, which corresponds to Fig. A.6 for an encoding phase of the algorithm with raised flags to the input binary sequences on chars ‘*a*’ and ‘*b*’. This process basically implies to the lowest layers of FBAR compression as exemplified and discussed as follows:

Lowest layers of compression: As we see, certain 1-bitflags raised in a ‘reference table’ (§ A.1.2), for encoding data, Fig. 7’s I/O products, will comply with the following flag and polarity settings table:

Type no.	Polarity set	Implies to	1-bit flag
0	↓↑↑↓	`most chars`	f_0=1bit
1	↓↓↓↑	`letters`	f_1=1bit
2	↓↑↓↓	`letters`	f_2=1bit
3	↓↑↓↑	`letters`	f_3=1bit
4	↓↑↑↑	`letters`	f_4=1bit
5	↓↓↑↑	`few letters`	f_5=1bit
6	↓↓↓↑, ↓↑↑↓, …	`dual chars`	f_6=1bit
7	↘	`all 2bit binary 10`	f_7=1bit
8	↗	`all 2bit binary 01`	f_8=1bit

Table A.5. Bit flag polarity combinations on bit pairs and nibbles during compression

The main flags are # 0 to 6 polarity flags. The remaining flags are concatenated and thus raised in the grid file. Programmatically, one could select relevant bit pairs based on these tables to reconstruct data for lower levels of compression inclusive of maximum LDCs. Once bitwise combinations of the reference table (A.7) are confronted within LDD program code, bit access for reconstruction between the grid field and compressed file is enabled. The above grid, however, is used and customized for any level of compression, either of lower layers of 4th up to its topmost possible LDC product.

A lower level encoding: For example, to reconstruct a character with decimal # 64, as “@”, based on a raised flag, say, flag # 0 (neutral or ignorable), the equivalent of the character’s binary would also be 01000000. The character’s compressed version through FBAR using its flow Table A.1, or its model (Fig. A.6), is “00↘0”, denoting that the first two zeroes are pure and give 0000, whereas the second pair “↘0” is indeed impure. The latter’s true face is “10 0”, indicating flag # 7. Thus, the flag bit dereferences noise as 10 during decompression, and for the remaining 0 in “↘0”, becomes 00. In total, we then have, 0000 1000 = 2 nibbles = 8 bits ≡ 1 character. Now we establish the pattern based on flag # 0 i.e. its polarity set, since we code our algorithm that every nibble is of a previously-ANDed type, and next to it, from left to right of a binary sequence, the ORed type (consider them as odd and even nibbles in a full binary sequence with a length > 8 bits). Hence, the ANDed version sits above as the North Pole, and the ORed version sits below as the South Pole:

$$\begin{array}{l} 0000 \\ \downarrow\uparrow\uparrow\downarrow = 01\ 00\ 00\ 00\ \equiv @\ , \\ 1000 \end{array}$$

Programmatically, one could conceive in terms of an equivalent pairwise selection from memory in a sequential manner. Consider an accustomed byte to some char in terms of

$$\text{ANDed} \rightarrow 0000\ 1000 \leftarrow \text{ORed}\ ,$$

Equivalently, pairing the bits in terms of

$$x_0 \curvearrowleft x_4 x_1 \curvearrowright x_5 x_2 \curvearrowright x_6 x_7 \curvearrowleft x_3 = 01000000 \equiv @$$

How to select and pair bits like above, is further elaborated in the following subsections

A.1.2.1 Function calls and arguments

So far, we have seen how FBAR method is defined in terms of logic constructs and their relevant components. Now we want to find the argument list assigned to each method. Arguments are presented in the same way for logic constructors and memory flag methods. The leading node element for arguments is the `fuzzyFlag()` function, and thus those arguments that register ‘bin chars’ inclusive of reading the contents of the constructive grid file with its ‘1-bit flag set comparator’ for their reconstruction at the LDD phase of the algorithm.

```
int fuzzyflag(void) {
packed_struct1.flag = 1;
packed_struct1.status = 3;
if( packed_struct1.flag == 1 ){
printf(" 1-bit; ");
}
return 0;
}
```

As we shall see by the end of this section, flags could be packed into efficient forms in terms of bit fields (explanations given after Eq. (4)). More interestingly, such flags are also self-embedded when we discover the right combinations of bits to manipulate pure data out of 1's as '1111…1' otherwise, 0's as '000…00', in terms of 'original data', when we devise some 'single bit combinatorial flag tables' akin to the periodic table with unique identity per a set of char entries.

So far, we know how to assign conditions and extract relevant data for the algorithm's logic. The combinations of bits to reconstruct the original characters, not only on the encryption level, on a compression level perceptively unique in representations i.e. a 4D cube or hypercube model of flags, makes all bitwise conversions from one layer of LDC to another reliably precise prior to any probabilistic pattern behavior. To further engage with our low-level conversions leaning toward high-level conversions, we implement certain functions defining our problem specific issues on LDC representations in our code. For instance, the pairwise mask function, shifting bits to the right ">>" otherwise to the left "<<", could do this encoding, i.e. a "bit registry process' implemented in terms of the following portion of the pseudocode

```
1.  #include <stdio.h>
2.  #include <string.h>
3.  #include <stdlib.h>
4.  #include <limits.h>
5.  //...
6.  void showBits(unsigned char ch, int width)
7.  {
8.  unsigned char mask;;
9.  for (mask = 1 << (width-1); mask; mask >>=1){
10. putchar(ch & mask ? '1' : '0');
11. }
12. }
13. void foo(const char *str)
14. {
15. FILE *fp, *gp;  /* a file pointer when I/O read_write operations are used */
16. int i, j;
17. fp=fopen("C.txt","w");
18. gp=fopen("G.txt","w");
19. for ( i = 0; str[i]; ++i )
20. {
21. unsigned char bit_and = 0;
22. unsigned char bit_or  = 0;
23. printf("str[%2d] = '%c' %02X ", i, str[i], (unsigned)str[i]);
24. showBits(str[i], CHAR_BIT);
25. printf(" ; ");
26. for (j = 0; j < CHAR_BIT / 2; ++j){
27. int m = CHAR_BIT-2*j;  /* {8, 6, 4, 2} */
28. int n = CHAR_BIT/2-j-1; /* {3, 2, 1, 0} */
29. unsigned char x =((unsigned  char)str[i]>>(m-1)) & 1;
30. unsigned char y =((unsigned char)str[i]>>(m-2)) & 1;
31. bit_or  |= (x | y) << n; /* apply bitwise AND */
32. bit_and |= (x & y) << n; /* apply bitwise OR */
33. }
34. }
```

Then we simply compress data by selecting the least significant bit (LSB) of the pairs per nibble, denoting closure points. This could be registered by the `fprintf()` and `putchar()` for the simulation grade, otherwise, `str[i]`, which is an array of chars instantiated for the implementation, during the bitwise AND/OR operation on variables `bit_or` and `bit_and` of the code (see 'for loop', line # 19, 31 and 32). From there, after converting the compressed binary to the compressed char, thereby written to the grid, we thus compress both **G** and **C** files in parallel. To view the results of these conversions, resulting in a compression, we type the following lines of code:

```
showBits(bit_and, 2); /* show bits in a total length of two from the ANDed column: an LSB selection */
showBits(bit_and, 1); /* show bits in a total length of two from the ANDed column: an LSB selection */
putchar('\n');
fprintf(fp,"\n");
```

One could, however, comment out or omit the lines of code that merely show bits indeed via `showBits()` function. The reason in using the function `putchar` and the argument `stdout`, are for simulation purposes only, displaying results on the screen to the user/programmer. These are coded for testing small samples only, such as countable strings with a custom buffer limit = 402 in the program. Therefore, `stdout` in showing the process for large tests is inappropriate and overly time consuming. Hence, without it, permits the program for its implementation to conduct relevant computations and processing with acceptable CPU time scenarios. The subsequent pseudocode represents the equivalent version of the actual code programmed above as an encoding solution in our algorithm:

Pseudocode sample I: a lossless data encoder

```
CREATE a FILE POINTER for READ_WRITE operations
WHILE reading CHARACTER by CHARACTER DO
OUTPUT CHARACTER as temporary BIN CHARACTERS
READ BIN CHARACTERS
NEW STRING = BIN CHARACTERS
BITWISE AND(1st2 CHARACTERS of STRING from MSB to LSB)
BITWISE OR (2nd2 CHARACTERS of STRING from MSB to LSB)
BITWISE AND(3rd2 CHARACTERS of STRING from MSB to LSB)
BITWISE OR (4th2 CHARACTERS of STRING from MSB to LSB)
IF 1st2 CHARACTERS in STRING is ‘01’ THEN
OUTPUT rightmost CHARACTER of this pair = ‘1’
ELSEIF 1st2 CHARACTERS in STRING is ‘10’ THEN
OUTPUT rightmost CHARACTER of this pair = ‘0’
ELSEIF 1st2 CHARACTERS in STRING is ‘00’ THEN
OUTPUT rightmost CHARACTER of this pair = ‘0’
ELSE
OUTPUT rightmost CHARACTER of this pair = ‘1’
END of IF
CONTINUE SORTING 2nd2 CHARACTERS, 3rd2 CHARACTERS,
4th2 CHARACTERS in STRING like before
OUTPUT RESULTS from BIN CHARACTERS to ASCII as 8 BIN
CHARACTERS = 1 ASCII CHARACTER
END of WHILE
```

As we can see, we simply compress data by selecting the least significant bit (LSB) of the pairs per nibble denoting closure points. This could be registered after applying bitwise *and-or*, and from there, after converting from compressed binary to compressed char, written to the **G** and **C** files in parallel. The simplified form of the ‘if statement’ with its ‘continuing course on sorting binary chars’ in the pseudocode, would be

```
...
SHIFT from MSB to 2nd rightmost CHARACTER in (
1st2 CHARACTERS, 2nd2 CHARACTERS,
3rd2 CHARACTERS, 4th2 CHARACTERS)
OUTPUT 2nd rightmost CHARACTER from (1st2 CHARACTERS, 2nd2 CHARACTERS, 3rd2
CHARACTERS, 4th2 CHARACTERS)
...
```

This results in, for every 8 bits, a 4bit output, and from there, 2bits, and finally, a 1bit output char. We pack each 8×1bit output into 1 ASCII char equivalent as our compressed version. The subsequent pseudocode represents what is necessary to code for an LDD, as a subroutine to the above code, recalling compressed values stored in char:

Pseudocode sample II: ***a lossless data decoder***

```
WHILE maksing BIN CHARACTERS from BITWISE AND and BITWISE OR results DO
STRING = 8 BIN CHARACTERS
ASSIGN ‘0’ to a DOWN variable
ASSIGN ‘1’ to an UP variable
FLAG_STRING = UP + DOWN CHARACTERS
IF FLAG_STRING = (DOWN + UP + UP + DOWN)
CHARACTERS THEN
STRING = (MSB BIN CHARACTER + 5th BIT
CHARACTER) + (6th BIT CHARACTER + 2nd BIT
CHARACTER) + (7th BIT CHARACTER + 3rd BIT
CHARACTER) + (LSB BIN CHARACTER +
4th BIT CHARACTER)
OUTPUT STRING = OLD 8 BIN CHARACTERS
ELSEIF CONTINUE CONCATINATE for other
FLAG_STRING UP + DOWN combinations
...
END of IF
OUTPUT RESULTS from BIN CHARACTERS to ASCII as 8 BIN
CHARACTERS = 1 ASCII CHARACTER
END of WHILE
```

So, for ‘@’ we reconstruct 0001 0000. Hence, during the decompression phase, having this flag available makes the algorithm to reconstruct data by tracing the arrows’ directions in the polarities set. Interestingly, the “@” char is also a dual character (it behaves as such), and could be raised by flag # 6 due to giving the same result for its decompressed version with different polarity combinations. But for reasons needed to occupy fewer bits, even in form of 1-bit flags, we reconstruct data by reciprocating with the grid file, cross-referencing with distinct bit groups, building up C_r values ≤ 2:1 compression.

A.1.2 The grid model, static versus dynamic allocations

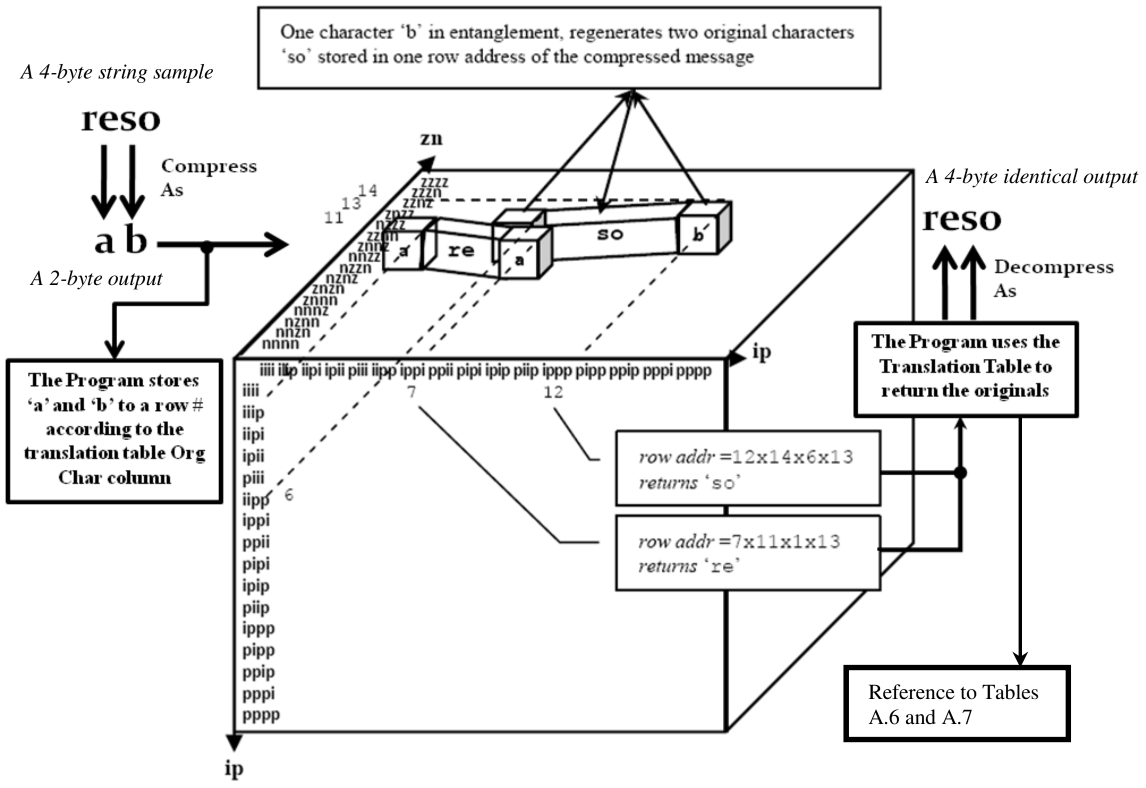

One character 'b' in entanglement, regenerates two original characters 'so' stored in one row address of the compressed message
A 4-byte string sample
reso
Compress As
a b
A 2-byte output
The Program stores 'a' and 'b' to a row # according to the translation table Org Char column
zn
ip
re
so
a
b
A 4-byte identical output
reso
Decompress As
The Program uses the Translation Table to return the originals
row addr =12x14x6x13 returns 'so'
row addr =7x11x1x13 returns 're'
Reference to Tables A.6 and A.7
iiii iiip iipi iipii piii iipp ippi ppii pipi ipip piip ippp pipp ppip pppi pppp
nnnn nnzn nznn nnnz znnn znzn nznz nzzn nnzz znnz zznn nzzz znzz zznz zzzn zzzz
11 13 14 7 12 6

Fig. A.7: The 4D logic constructor grid with input proving a successful superdense technique.

A.1.2.1 A robust static solution to LDDs

The main focus for reconstructing data, is considering negation flags # 1 to 4, pure and impure flags 1 to 4, ORed in combination for each compressed character in the **C** file. A comparator as the FBAR program subroutine compares results between the static table as a point of reference with the dynamic component, **C** file, and the semi-dynamic component, the **G** file. The process relationships have been illustrated in Figs. A.4, A.5. From there, a compression of 4-bits per compressed chars in the final layers as a 1-bit representative is performed. In total, 5 bits for each string entry identified for a decompression. To every unique combination of pairs made by the comparator, a specific 1 bit flag is allocated in the fixed size memory chunk with a specific address like from the portable compressed file, **C**. This phase of LDC denotes a 5-6 bit compression, giving an average anticipation of 34 to 36% space savings for a 95 random string entries. The allocation of single bits raised in the memory, and from there, to the **G** file for each character per memory chunk is computed by the following equation:

$$C(m) \xrightarrow{\text{in}} \lambda(\mathbf{G}) = \lambda(\mathbf{G}) + \frac{m}{2}, \quad \mathbf{G} \geq 64\text{K}, \tag{2}$$

where m is the number of string characters inputted to the program for a compression. Once compressed, the length λ of the grid file **G** is summed with the compressed m, equal to $m/2$. The default value of 64K comes from the three dimensions representing a char representative for each combination set of **ip** and **zn** as specified above. This default value is computed based on the possible number of grid outcomes, Eq. (2), quite convenient for a 16-bit microprocessor to directly access and process the **G** file via a set of hardcoded 'if else statements' on flags subroutine in our code. As we shall later observe, to conduct an FBAR LDD, data access of the compressed file is in 65,536 rows, which is compatible with Excel 97-2003's maximum number of spreadsheet rows. The expectancy of lower sub-layers of the 4th layer would decrease the number of possible combinations of 1×4-bit flags, making the cube denser than the current version.

The expectancy of lower sub-layers of the 4th layer would decrease the number of possible combinations of 1×4-bit flags, making the cube denser than the current version. This is due to having more bits available to decompress from those sub-layers of the algorithm. But in this case, the total number of possible combinations per dimension is fixed, or

$$\mathbf{G_{xy}} = \{ch_i ch_{i+1}\}\,\mathbf{ip} \times \{ch_i ch_{i+1}\}\,\mathbf{zn} = 2^{4\times4} = 2^{16} = 65{,}536 \text{ possible grid outcomes} \tag{3}$$

where the grid model is hereby shown in Fig. A.7. Perceivably, in Eq. (3), out of the two *xy* bit flag field dimensions, we create a four dimensional hypercube. So, for every arbitrary input document, half of the size of that document is created between the four fixed dimensions of **ip** *x* vector for char ch_i, **ip** *y* vector for its neighboring char ch_{i+1}, and **zn** *xy* vector for both chars respectively. The **zn** and **ip** vector dimensions, each, are presented in separate rows in a list, mounting 16 indexed 4-bit flag sets correspondingly. The coverage of the grid is to concurrently cover all Unicode chars, even non-printable char scenarios for any data type.

The main rule for each row of entry is to always maintain a 2×4 *and-or* bit encoding, and a < 8 bit data compression. In addition, memory transactions are abstracted in Figure 7, as double line circles of the process. The middle phase of the algorithm is shown as a double-dashed circle process component. This component should eventually substantiate that, Compressed Data $<, \ll, \lll \ldots <$ Original Data. The main contribution of this process is in two parts: the first part is where AND/OR logic is applied to the input data, thereby fuzzy decisions i.e. closure points of logic on the pair products 01, 11, 10, 00, leading to 1, 1, 0 and 0 state logic, respectively. The second part, however, is where flags are raised and memory transactions occur after dictionary index establishment of data, reference point and bitwise data comparisons. The comparisons are executed through 'if and else' statement conditions for every combination of single bits, identified in terms of: bits of original data, flag bits denoting bit position and array index. Subsequently, the radical phase of processing data coincides with parallel reconstruction of data from the dictionary coder per data sequence. This is the decoding phase of the process, leading to a lossless data decompression (LDD). The LDD holds values identical to the original data, i.e. a fully-reconstructed data, which denotes an LDC cyclic behavior. It is now obvious to substantiate the cycle in terms of

$$\text{At } t_1,\ f(\text{Original Data}) = f'(C) = \text{LDD} = \text{Original Data},$$

$$\text{At } t_2 \text{ static/dynamic},\ f(\text{LDD}) = f'(C) = \text{Original Data},\ \therefore \text{ applicable to } t_i \text{ cyclically, where } i > 1. \quad (4)$$

The function f result in Eq. (4), is a compressed data C as an FBAR compression conducted at an initial time session t_1, such that $t_1 >$ a subsequent time session t_2, performing an FBAR LDC. The primed function f', engages data decompression to the extent of time sessions' difference $\Delta t = t_2 - t_1$, for all static and dynamic memory accesses. The conversions of functions from one data form to another, preserves this cycle by engaging the use of 1-bit flags. The use of a flag, or sentinel, in FBAR cases, is a customized type of flag, set to either true or false i.e., Boolean data type for any variable. Its sole purpose is to indicate when a key point in the LDC processing has been reached. This includes things like breaking out of a loop, satisfying a pairwise data compression, being able to access a resource or the **G** file, its if-else decision tree by the interpreter for a char reconstruction, sharing between threads, and string entry binary combination according to FBAR's LDC Tables A.7 and A.8.

A.1.2.2 A dynamic allocation to LDDs

In C/C++, Boolean variables consume 1 byte of memory. But all that is really needed is 1 bit: 0 means false, 1 means true. Many times, especially when dealing with graphics, rather than consuming a whole byte of memory for each Boolean, several Booleans are combined into a single byte of memory, where each Boolean uses a different bit in the byte. These are then referred to as bit flags, or bit fields.

A customized version of a bit flag is –1, and when set to true, indicates that this bit flag negates all possible combinations in terms of 'down down down down' polarity set, compared to flag # 0 to 6, which each have at least two opposing directions between AND and OR poles of the double nibble binary. In Table A.8, one could substitute flags with extended bits for –1 to have more compressed bits referenced in a 1x96 memory block.

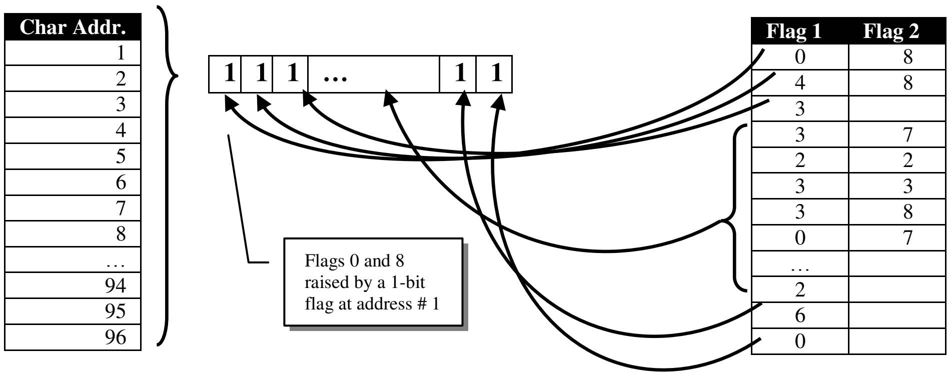


Fig. A.8: The corresponding 96-bit-block memory in use for the fixed-size reference table.

The polarity set has its own two unique flags as a customized type: a post-flag 9 and –1, whereas the latter is user customized since the negation of other types physical space (memory) is the extension of their polarities. Hence, ~{1,2,3,4,5,6,7,8} = {–1, 9}, where for a given combination, the remaining polarity combinations become a subset of type 6. In this case, this analogy of preset and extended bit combinations, makes $\{–1, 9\} \subset 6$ a tautology. Thus, ~ {9} = {1, 2, 3, 4, 5, 6, 7, 8} – {↓↓↑↓} and ~{–1} = {1, 2, 3, 4, 5, 6, 7, 8} – {↓↓↓↓}, from the memory chunk. Contemplatively, as if ~ {1} and ~ {9} are symmetric, and are the complements to the whole byte enumeration value closing with the 8th bit. Now, the 9th bit extension, or its symmetry, the –1st preset upon {1, 2, 3, 4, 5, 6, 7, 8}, is the complement to a possible byte closing with the 8th bit, and starting with the 1st bit, without enumeration substitutes. This is how we envisage the location of flags –1 and 9 after a full byte allocation for a total set of flag combinations.

The 0 bit case not included in the set is the 0 flag type, and one could conceive this as an easy bit or a preset flag occupying no space. The reason is that the table's attribute (the Flag 1 and 2 columns), gives us a range of possible combinations' representative for polarity indicators i.e. 1 to 6 inclusive of probable impure flag conditions to be raised when deemed necessary. These flags are 7 and 8, and are unique flags adjacent to –1 and 9, respectively. If we leave any space of the full attribute empty, in this case, flag # 0, then without worrying about how much space is allocated in the 96 memory addresses, we ignore it as a space occupier since this is the only row that when not present compared to the remaining, denotes the excluded flag type amongst other flags. So, the pre-setup of this flag is obvious in bit count during loops defining possible combinations of the 1-bit flags and char address attributes.

The following is an example, using an 8-bit unsigned integer to store 8 flags which relay to the 'bit filed' of 'bit flag' concept in C language. Our approach, if used in terms of 'bit arrays', we would just encode rather than compress data since bit arrays consume at least 1 full byte of memory for a single Boolean variable. We shall later state that a bit field approach, however, is necessary to preserve total bits allocated for char entries in Table A.8. A bit field is distinguished from a bit array, in that, the latter is used to store a large set of bits indexed by integers and is often wider than any integral type supported by the language. Bit fields, on the other hand, typically fit within a machine word, and the denotation of bits is independent of their numerical index. Now, let's try the bit field approach and thereby, bit array for the sake of its usefulness to a set of encodings in aim of simulating the correctness of FBAR table LDC I/Os:

```
unsigned char options;
```

The possible options, that can be turned on or off independently are declared in an *enum* like this e.g., just using some arbitrary identifiers on the left, but exact identifiers for FBAR polarities set on the right:

```
      enum Options {
        f1      = 0x01,
        f2      = 0x02,
        f3      = 0x04,
        f4      = 0x08,
        f5      = 0x10,
        f6      = 0x20
        // ...
      };
```

Note how each option is given a specific value. These values are carefully picked to match each bit in the 8-bit variable:

```
      // 0x01 ==   1 == "00000001"
      // 0x02 ==   2 == "00000010"
      // 0x04 ==   4 == "00000100"
      // 0x08 ==   8 == "00001000"
      // 0x10 ==  16 == "00010000"
      // 0x20 ==  32 == "00100000"
      // 0x40 ==  64 == "01000000"
      // 0x80 == 128 == "10000000"
```

Now, each flag can be set independently, by using the bitwise OR operator:

```
      options = f1 | f4 | f7;
      // options == 0x01 | 0x08 | 0x40 == "01001001"
```

And can be tested using the bitwise AND operator:

```
      if (options & f1) {} // true
      if (options & f4) {} // false
```

However, it is imperative to have further flag representative based on the fixed reference table, which also comprises of extended if statements covering not only flags, representing compressed resultants in a unique manner. Hence, we could hide the possible flag options and have just one raised bit value within the *n*x96 memory blocks representing Table A.8 for a text input, column Input, via columns Flag 1 and Flag 2, as follows:

```
      // 0x96x01 == 1 == "0000000000000000000000000000000000000000000000000000
      //                  00000000000000000000000000000000000000000011"
      // 0x96x02 == 2 == "0000000000000000000000000000000000000000000000000000
      //                  00000000000000000000000000000000000000000101"
      // 0x96x03 == 3 == "0000000000000000000000000000000000000000000000000000
      //                  00000000000000000000000000000000000000001001"
      // ... ==   ... == ...
```

wherein continue, supposing that all standard ASCII characters raised in the first count, we then get

```
      // 0x96x01 == 1 == "1111111111111111111111111111111111111111111111111111
      //                  11111111111111111111111111111111111111111111"
      // 0x96x02 == 2 == "0000000000000000000000000000000000000000000000000000
      //                  00000000000000000000000000000000000000000000"
      // 0x96x03 == 3 == "0000000000000000000000000000000000000000000000000000
      //                  00000000000000000000000000000000000000000000"
      // ... ==   ... == ...
```

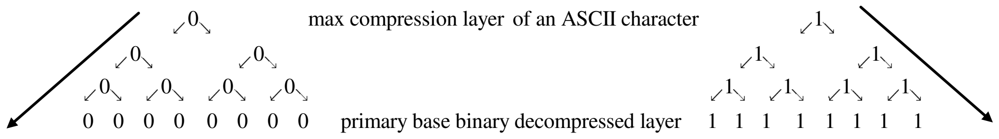
whereas its subsequent counts for other char combinations is inefficient for compressing data. This approach solely relies on bit array lengths of minimally 8-bit lengths and maximally *n*×8-bit blocks.

A.1.2.3 Maximum FBAR LDC ratios and their respective LDDs

In connection with the last presented maximal length ASCII memory block occupation, a definite question pops into our minds is,

Q. Why not we create a pure bit-byte sequencer representing a whole block instead of occupying it like the above code for the interpreter, before any char conversions?

A. When characters formulate words in, e.g., text, the distance gap between array indices increases and thus filled up with 0's. This would be merely useful when characters are lined up in a certain repetitive order, as laid out in the ASCII table in an ordered fashion of decimal. So, the mapping of our flags into a plausible data compression using the 'bit field' approach lays out in the memory fixed size blocks of 96 bits partitioned into 8 bit words, with a cross pattern intersection bits looks as follows:


max compression layer of an ASCII character
primary base binary decompressed layer
0
0 0
0 0 0 0
0 0 0 0 0 0 0 0
1
1 1
1 1 1 1
1 1 1 1 1 1 1 1

In this tree, each block exhibits at least a set of flag combinations in terms of 1-bit flag representatives without considering a full 8-bit word, in form of binary packets in memory. The packets preclude fuzzy logic conditions for an LDD, retaining bits in a logic constructor grid for each primary binary result. The primary result is in form of a pure base binary '00000000, otherwise 11111111'. The combinations of the grid obey impure and pure pairwise bit combinations, intersecting with negated bit pairs relevant to each decompressed pure sequence, '00000000', otherwise, '11111111'. This contrasts with the version that investigates bits in form of a bit array, allocating a full 8-bit length representation. The 'bit array' approach is merely useful to check FBAR table for an LDD based on equivalent encoded characters, 'printable', when attaining the final levels of decompression. The possible combinations of negation, impure and pure bit grid from Fig. A.7, are as follows:

ip: impure or pure pairwise bits' dimension:

iiii iiip iipi ipii piii iipp ippi ppii pipi
ipip piip ippp
pipp ppip pppi

zn: zero or negate pairwise bits' dimension:

zzzz zzzn zznz znzz nzzz zznn znnz
nnzz nznz znzn
nzzn znnn nnnz nznn nnzn nnnn

where all combinations are presented after we logically AND them in our comparator when an LDD phase is initiated. This gives us the idea of representing all occupying information in terms of a sequencer of '1' representing 111…1 blocks by default to manipulate based on pure and impure flag combinations. For instance, if we have a pure, pure, impure and pure set with 1 leftmost bit to negate, a sequencer for a byte 11111111 generates 11110111 for the impure pure dimensions and after negation, 00110111 which is equivalent to char '7' decoded in ASCII. This approach solely relying on the grid file with a default sequencer of '1' for the whole data, gives a pure 50% LDC.

A self-embedded flag set method: The cross-section, of which the compressed characters are recognized in the **G** file, is read by the 'decompression subprogram', thereby compared with the **C** file content and table for a successful data reconstruction. The entries are of the reference table, building up to 95 standard ASCII chars. When the scanning of the **G** file entries reiterates for the next 96 char block, considering char # 96 as a block double byte (BDB), the program then counts from 97 up to 191 and so on, traversing all 65,536 rows, "flag sets", for an LDD. We use the BDB as an indicator, e.g., a two-char '`/a`' representing the 1st full 96 byte allocation, '`/b`' for the 2nd and … The BDBs are standard chars elicited from the ASCII table. The 'if and for loop' on the LDD, for 65,536 possibilities, is the key to this process. This is later explicated in pseudocode at the LDD phase. The rows are in matrix form, denoting at least two original chars held by a 'position char' with a 1, otherwise, a 0 sequencer. The position char as illustrated in Fig. 5, is an 'occupant char' in the **G** file, starting with an '`a`' to the last ASCII 95 characters, representing in total, 95×2 = 190 char entries, or 95 compressed chars denoted by the ***C***(char) column in Table A.6.

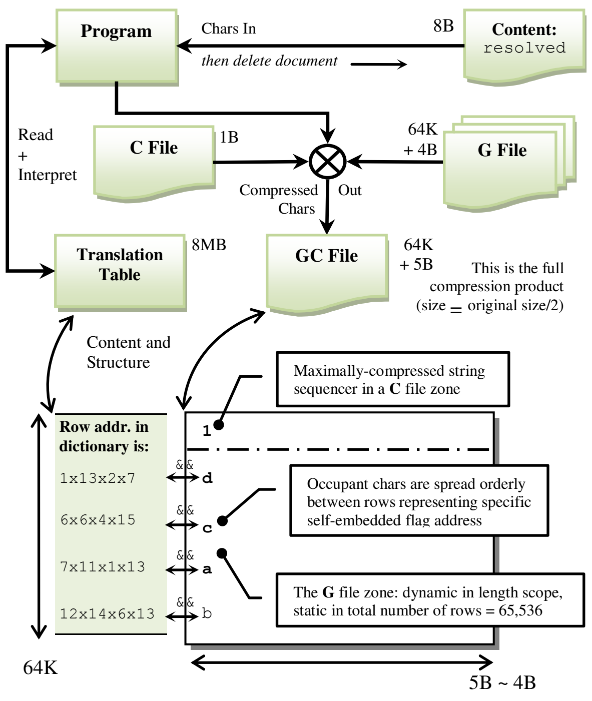

Program
Chars In
8B
Content: resolved
then delete document
Read + Interpret
C File
1B
64K + 4B
G File
Compressed Chars
Out
Translation Table
8MB
GC File
64K + 5B
This is the full compression product (size ≅ original size/2)
Content and Structure
Maximally-compressed string sequencer in a C file zone
Row addr. in dictionary is:
1
1x13x2x7
d
6x6x4x15
c
7x11x1x13
a
12x14x6x13
b
&&
Occupant chars are spread orderly between rows representing specific self-embedded flag address
The G file zone: dynamic in length scope, static in total number of rows = 65,536
64K
5B ~ 4B

Fig. A.9: The GC file with an 8B to 5B~4B compression

For example, the elements in {`a`, `b`, `c`, `d`, …, `/a`}, are respectively interpreted by the program's interpreter as: the {1st 2chars, 2nd 2chars, 3rd 2chars, 4th 2chars, … end of the 95th 2chars [of the original file]}.

Row address	C(char) #; C_r	Original chars; total	Occupant char	Size in bits
7x11x1x13	1; 2:1=50%	re 2	a	8
12x14x6x13	2; 2:1=50%	so 4	b	8
6x6x4x15	3; 2:1=50%	lv 6	c	8
1x13x2x7	4; 2:1=50%	ed 8	d	8
13x1x1x6	5; 2:1=50%	f 10	e	8
6x13x7x11	6; 2:1=50%	or 12	f	8
⋮	⋮ ⋮	⋮ ⋮	⋮	⋮
the same as last	48; 1:1=0%	∞ **96**	/a	16
8x12x8x12	49; 2:1=50%	55 98	a	8
8x12x11x2	50; 2:1=50%	5$ 100	b	8
⋮	⋮ ⋮	⋮	⋮	⋮

Table A.6: I/O character process and occupation

This alleviates char interpretation over binary when presented by char position through standard ASCII chars: `a`, `b`, …, □. To override memory overrun(s) during the vast access of files in read/writing data, we organize the '**G** with **C**' files into one single file, merging the targeted components of Fig. 3 into **GC**. A structural sample of **GC** is illustrated in Fig. 6. This approach makes the algorithm quite portable, thus no need to be concerned about memory allocation and management issues in this regard. The corresponding table to the grid, following bitmap pattern reconstruction for any character per impure and pure flag preference "*arose in bit field as necessary, is to hold a unique identity for that particular char,*" and is given in Table A.6.

The *chiefly-key exponent* to all of this, is the following expanded table to the latter translation table exemplar on the 4x1-bit flags, which indicates the very notion of any reconstructible character (Org. as original character from the 4th column). To reconstruct multiple characters from the 4th column, the comparator reads data located on the 3rd column representing the original character position as specified for Table A.7. The 1st column is just an index to the 4×1-bit flag combination address. This address is no needed to be traced, and thus, just by the comparator's 'if-else' functions of the subprogram, it affirms 'occupant chars' (3rd column) of the compressed file, with the address representing 'impure' and 'pure' bitwise operations on the sequencer. The sequencer is either a sequence of 11111111 via char '1' or '00000000' via char '0', as specified.

The column of 96 characters in Table A.7, is of standard printable ASCII characters, revealing the position of the 'first to 96th 2-chars (double) of the original file'. The column on the address part '1x1x1x1' is the actual row being occupied by a character (the address) in the compressed file. Once the program reads this dictionary parallel to the compressed file, returns the original character according to the corresponded row (last column containing 2 characters). This version indicates a 50% LDC.

Row #	Bit flag add.	95 ASCII Chars as Occupant Chars representing the "Org." column via the "4x1-bit flag Addr." column	Org. char
1	1x1x1x1	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890`~!@#$%^&*()-=_+[]{}\\|;:'"/?.>,<	ª ª
2	1x2x1x1	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890`~!@#$%^&*()-=_+[]{}\\|;:'"/?.>,<	¥ ª
3	1x3x1x1	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890`~!@#$%^&*()-=_+[]{}\\|;:'"/?.>,<	• ª
4	1x4x1x1	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890`~!@#$%^&*()-=_+[]{}\\|;:'"/?.>,<	© ª
…	…	…	…
65534	16x16x16x14	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890`~!@#$%^&*()-=_+[]{}\\|;:'"/?.>,<	ÿó
65535	16x16x16x15	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890`~!@#$%^&*()-=_+[]{}\\|;:'"/?.>,<	ÿü
65536	16x16x16x16	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890`~!@#$%^&*()-=_+[]{}\\|;:'"/?.>,<	ÿÿ

Table A.7: The actual translation table contents for an LDD access and management

For an 87.5%, obviously, the column with 96 characters will not change, however, the '1x1x1x1' column in its configuration would become '1x1x1x1 1x1x1x1' , and the last column with 2 characters, becomes 8 characters, since the cubic representation of the '1st 1x1x1x1' with the '2nd 1x1x1x1' has a second non-commutative symmetric format: '2nd 1x1x1x1' with the '1st 1x1x1x1', giving four distinct addresses simultaneously. So, for the former, this means, 2-original characters results in 1-character in compression (2:1 or 50%), and for the latter, 8 original characters results in 1 compressed character (100% – 12.5% = 87.5% or 8:1) as an 'occupant character' (see Table A.6), occupying a row in the compressed file **GC**. This is how the process of the new lossless data compression occurs. The following is the magnified version of the contents of the dictionary (translation table above) for a 50% compression. The symmetry, altogether, gives four distinct double char addresses simultaneously i.e., an 8:1 LDC. This satisfies $65{,}536^{4} = 1.84 \times 10^{19}$ unique combinations, or, 16 exabytes (EB) of grid rows. In case of columnar symmetry in two translation tables, $65{,}536^{2} = 4.1$GB, handles the ≈ 16 EBs when column values are intersected by a comparator matrix in our code. So, four 64K grid row combinations, handles the same EB values in four parallel tables. This requires complex matrix coding on an x86 machine. A 64-bit microprocessor, in principle, handles at most, 18 EBs of space. So, beyond this limit, we run the FQAR model combined with the Bloch sphere on a quantum computer, easing the complex matrix programming, to superdense the EBs down to the 64K limits of grid rows.

1	1x1x1x1	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1 234567890`~!@#$%^&*()-=_+[]{}\\|;:'"/?.>,<	ª ª
2	1x2x1x1	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1 234567890`~!@#$%^&*()-=_+[]{}\\|;:'"/?.>,<	¥ ª
3	1x3x1x1	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1 234567890`~!@#$%^&*()-=_+[]{}\\|;:'"/?.>,<	• ª
4	1x4x1x1	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1 234567890`~!@#$%^&*()-=_+[]{}\\|;:'"/?.>,<	© ª
…	…	…	…
65534	16x16x16x14	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890`~!@#$%^&*()-=_+[]{}\\|;:'"/?.>,<	ÿó
65535	16x16x16x15	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890`~!@#$%^&*()-=_+[]{}\\|;:'"/?.>,<	ÿü
65536	16x16x16x16	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890`~!@#$%^&*()-=_+[]{}\\|;:'"/?.>,<	ÿÿ

For highest doubled-efficiencies, we extend the number of **znip** *columnar combinations* from the previous translation table in terms of

$$\begin{tikzcd}
\text{Table 1} \arrow[r, leftrightarrow] & \text{Table 2} \arrow[r, leftrightarrow] & \text{Table 3} \arrow[r, leftrightarrow] & \text{Table 4} \\
1x1x1x1 & 1x1x1x1 & 1x1x1x1 & 1x1x1x1 \\
\ldots & \ldots & \ldots & \ldots \\
16x16x16x16 & 16x16x16x16 & 16x16x16x16 & 16x16x16x16
\end{tikzcd}$$

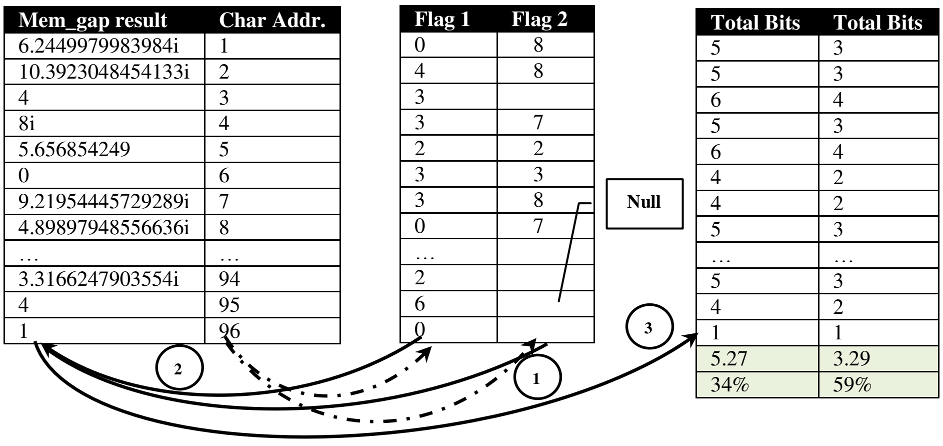
This is called FQAR, or, *a strongly quantum oriented algorithm*: It delivers double doubled-efficiencies, and thereby quadrupled efficiencies as well. We described this in terms of fulfilling 4.1 GB and 15.61 EB combinations in the above paragraphs, respectively. In other words, we simply say, for all occasions, the program's *interpreter/comparator matrix* must be able to handle 1, 2 and 4 translation tables for all intersections between them, needing just 8, 16 and 32 MB static size on the x86 version instead of the EB barrier denoting no columnar interactions whatsoever. For example, an intersection of {1x1x1x1} with {1x2x1x1} with {16x16x16x15} with {1x4x1x1}, from translation tables 1-to-4, will thus return, {ªª¥ªÿü©ª} original chars, hence the considerable length of 64 bits, is thus self-contained by the program's comparator efficiently (using just 8 bits, out of the 32MB of the traversed tabular space, denoting an 87.5% LDC).

A.1.3 Dynamic and static results returned by the algorithm

We continue to associate this relevantly-customized 4D-grid with the sequencer '1' or '0' to eventually decode and reconstruct data when attaining levels of LDD. The grid as a 'logic constructor' is pretty useful for maximum state of FBAR LDCs. With the polarity flags for lower levels of compression, useful to the 1st through 3rd levels of FBAR's 4th layer, the grid could be thus mapped to the memory efficiently, especially when quantum technology is involved. However, the main purpose is first to use fuzzy logic on an x86 machine, thereby applying it to quantum forms in the future, pertinent to the merits of logic states presented in Eq. (1). Once data validated on its integrity, a step further for an actual LDC is prioritized. Hence, the concept of 'bit fields' using low level operators in C, is well-structured as an advanced tool in programming, no matter the complexity of the bits of the compressed information. One could interpret a bit field as integral type, compacting multiple bits in partitioned fields of array. Bit fields allow integer members to be stored into memory spaces smaller than the compiler would ordinarily allow. Hence, the usage of two-dimensional stack allocation of bits becomes the most privileged in the FBAR LDC process model design. Fig. A.10 is the representation of such stacked allocations, assuring lossless compression as a dictionary to be reused during decompression. This is a crucial subject to normalize data, securing no loss of data integrity through *bit insertion* rather than character or integer insertions, thereby linking data objects via database keys. Further than insertion is, *updating data* and *deletion anomalies*. The optimized form of memory under its 'area integral' denoting data accumulation and thus allocation, using the Pythagoras theorem, generates the stacked line chart (Fig. A.11) based on Table A.8 values. The following equation, is computed to construct this stack for a set of raised flags to the corresponding bit allocation per char entry

$$\Gamma_{\text{memory}}(x) = \sqrt{-1\times(\text{total bits allocated})^2 - (1\text{bit flag}_\wedge + 1\text{bit flag}_\vee)^2} \tag{5}$$

where Γ_{memory} is the memory gap function, and x is the changing total bits in the memory from the original input string, whereby its values fall into both complex and real numbers categories. The imaginary part is visually contained within the stack, and here, not shown in Fig. A.11. The complex numbers on their imaginary part consisting with imaginary unit $i = \sqrt{-1}$, denote the memory space (matrix) allocated for MSB values, inclusive of the range above the 4th bit occupied by a single bit flag for each char entry. This memory equation comes from the way an observation is made on memory gap(s). We elaborate on how it would be possible to make data compaction more efficient than the static approach during compression by calculating the imaginary number for each gap distance. The equation, however, geometrically benefits from the Pythagorean relation, which mainly focuses on the spatial limit per process time unit (spatial and temporal for addressing efficiency, inclusively). Of course, the static approach undoubtedly demonstrates double-efficiency as a robust solution in all circumstances with the aid of the 4D grid and translation table. It is, however, on the other hand, well-defined to make it further efficient per double-efficient occurrences for each double, quadruple and ...-chars handled by 1 compressed char, as a dynamic way to solve the problem further. The equation shows a gap is always existent in the memory in terms of unnecessary extra bits occupying that space. The gap is where an expert programmer does not want to see extra 0's and 1's prior to the raised flags or byte information. Imagine, for a 1bit flag, we really need a 1bit segment, and in return, based on 'bit array' standards in C or any other programming language supporting this data structure, allocates a full byte for that single bit. So, in time, we experience an exponential growth of this occupation of bits in memory, forming a gap i.e., computable by Eq. (5). Therefore, this equation is essential to compute highly efficient compression in any layer of the algorithm (encoding layers upwards). It is necessary, before reaching this equation, to recall the previous subsections on the bit array solution in terms of "bit field" usage (see the 1st paragraph after our pseudocode main sample, Appdx. A; or see §A.1.2.2 which all leads to the discovery of the 4x1 bit flag model), in aim of understanding the memory gap problem via this equation. In layman's terms, just consider the gap area being filled up with a bunch of e.g. unnecessary 0's relative to those bits that we want to physically allocate. The following table contains the values computed for the allocated bits for this memory gap



Mem_gap result	Char Addr.
6.2449979983984i	1
10.3923048454133i	2
4	3
8i	4
5.656854249	5
0	6
9.21954445729289i	7
4.89897948556636i	8
…	…
3.3166247903554i	94
4	95
1	96

Flag 1	Flag 2
0	8
4	8
3	
3	7
2	2
3	3
3	8
0	7
…	
2	
6	
0	

Total Bits	Total Bits
5	3
5	3
6	4
5	3
6	4
4	2
4	2
5	3
…	…
5	3
4	2
1	1
5.27	3.29
34%	59%

Fig. A.10: The address of characters and their raised 1-bit flags with respect to their *i* values

The values are elicited from Table A.8. The columns 'Flag 1' raised for the ANDed pair of bin characters or $\text{1bit flag}_{\wedge}$, and 'Flag 2' raised for the ORed pair of 'bin characters' or $\text{1bit flag}_{\vee}$, and the column on either 'Total Bits' is for 'total bits allocated' from the equation. The 'Char Addr.' column, indicates the character position or its actual address from the original file, that is currently being compressed e.g., index # 1 for the 1st character, index # 2 for the 2nd character, index # 8 for the 8th character and so on. By other means, a selected or input char is now in session for an LDC. As we follow the progressive finite steps of the conversions algorithm from layers 1 to 4, during the perpetual conversions of the 'char' into 'binary chars' based on the main pseudocode procedures, the binary char results are thus returned for a memory gap computation, whilst the compressed are returned relative total bits allocation columns. The listed values on the left table denoting imaginary and real numbers, correspond to the left column on the Total Bits table, giving a 34% compression. A different scenario for efficient compression, gives a 59% compression while contemplating that there are/were memory gap issues to tackle with.

The above information is very useful when revolutions of space for every periodic frequency are occupied in quantum forms rather than Boolean forms of logic. This is subject to future generation computers relying upon quantum information technology. The projection of gaps in form of complex numbers could be listed by the following matrix:

$$x_i y_j \rightarrow \begin{pmatrix} 0 & -1 \\ 1 & 0 \\ \vdots & \vdots \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} 0 & 1 \\ -1 & 0 \\ \vdots & \vdots \\ -1 & 1 \end{pmatrix} = \left\{ \begin{array}{l} \{\{0,-1\},\{\{\{-1\},\ \{\{\{-1\},\ \{\{\{-1\},\ \{-1\}\ldots\}\}\}\}\}, \\ \{\{\{\{0\},\ \{0\}\}\},\ \{\{\{0\},\ \{0\}\ldots\}\}\}\} \end{array} \right\}$$

This is the remaining matrix solution for a proper flag configuration, and it obviously denotes the spatial occupation for the range below the 4th bit to the LSB position. Both MSB and LSB flag conditions, build up a supplementary module (auxiliary bits) to dereference bits via pointers once their address is recalled at the decompression phase. All entry points as chars are defined with certain flag type combinations for 'pattern match opportunities' on every pair of bits constructing a character. The range, as previously specified, is literally {–1, 1, 2, 3, 4, 5, 6, 7, 8, 9} distributed across the 96×12 memory scope per 1 to 96 char entries. Hence, contemplating this table with its extending dimensions to the memory's upper and lower bounds (data accumulation or stack integral), marks each token or symbol entry as an FBAR dictionary coder /decoder, pertinent to any LDC standards in a temporal and spatial feedback course.

The first procedure during decompression, is observing the feedback loop to all encoded data instances between 'if and else' statements per char entry, makes pattern match of each char entry to its binary form possible. The pattern match from one bit of the binary sequence representing a text unit (a char entry) to another is the main method of the FBAR coder algorithm. Referencing to the bit's position (rather than the string or entry position), is spotted in the 2D representation of the dictionary. A perceptive partitioning of the memory for the raised flags is of importance in terms of 96/12 = 8 bit size partition (tree view above). This type of partitioning resembles with the 'greatest common devisor' (GCD) concept on numerical analysis for abstracting data structures. Ergo, the present "data structure" is reused to eventually decode with respect to the bit's reference point. Therefore, a map of all reference points of compressed data is generated. The next procedure is to rematch mapping points or bits with other bits relative to their memory location, position and polarity type. Once re-matched in the decoded pattern match technique, once again, recalls polarities between ANDed and ORed versions of the product, assuring an LDD.

Input	Binary	AND Out	OR Out	Out1	Out1	Out2	Out2	Out3	Flag1	Flag2	Addr.	Total Bits	
A	01100001	0000	1101	00	11	0	1	1	0	8	1	5	3
B	01100010	0000	1101	00	11	0	1	1	4	8	2	6	4
C	01100011	0001	1101	01	11	1	1	1	3		3	5	3
D	01100100	0000	1110	00	10	0	0	0	3	7	4	6	4
E	01100101	0000	1111	00	11	0	1	1	2	2	5	6	4
F	01100110	0000	1111	00	11	0	1	1	3	3	6	6	4
G	01100111	0001	1111	01	11	1	1	1	3	8	7	6	4
h	01101000	0000	1110	00	10	0	0	0	0	7	8	5	3
i	01101001	0000	1111	00	11	0	1	1	0		9	4	2
j	01101010	0000	1111	00	11	0	1	1	4		10	5	3
k	01101011	0001	1111	01	11	1	1	1	4	8	11	6	4
l	01101100	0010	1110	00	10	0	0	0	4		12	5	3
m	01101101	0010	1111	00	11	0	1	1	0	7	13	5	3
n	01101110	0010	1111	00	11	0	1	1	3	7	14	6	4
o	01101111	0011	1111	01	11	1	1	1	2		15	5	3
p	01110000	0100	1100	10	10	0	0	0	6	8	16	6	4
q	01110001	0100	1101	10	11	0	1	1	2	8	17	6	4
r	01110010	0100	1101	10	11	0	1	1	1	8	18	6	4
s	01110011	0101	1101	11	11	1	1	1	6		19	5	3
t	01110100	0100	1110	10	10	0	0	0	1		20	5	3
u	01110101	0100	1111	10	11	0	1	1	2		21	5	3
v	01110110	0100	1111	10	11	0	1	1	3	8	22	6	4
w	01110111	0101	1111	11	11	1	1	1	3		23	5	3
x	01111000	0100	1110	10	10	0	0	1	5		24	5	3
y	01111001	0100	1111	10	11	0	1	1	0	8	25	5	3
z	01111010	0100	1111	10	11	0	1	1	5		26	5	3
A	01000001	0000	1001	00	01	0	1	1	0		27	5	3
B	01000010	0000	1001	00	01	0	1	1	5		28	5	3
C	01000011	0001	1001	01	01	1	1	1	0		29	5	3
D	01000100	0000	1010	00	00	0	0	0	3		30	5	3
E	01000101	0000	1011	00	01	0	1	1	2		31	5	3
F	01000110	0000	1011	00	01	0	1	1	3		32	5	3
G	01000111	0001	1011	01	01	1	1	1	3		33	5	3
H	01001000	0000	1010	00	00	0	0	0	0		34	4	2
I	01001001	0000	1011	00	01	0	1	1	0	7	35	5	3
J	01001010	0000	1011	00	01	0	1	1	4		36	5	3
K	01001011	0001	1011	01	01	1	1	1	4		37	5	3
L	01001100	0010	1010	00	00	0	0	0	6		38	5	3
M	01001101	0010	1011	00	01	0	1	1	2	7	39	6	4
N	01001110	0010	1011	00	01	0	1	1	1		40	5	3
O	01001111	0011	1011	01	01	1	1	1	6		41	5	3
P	01010000	0000	1100	00	10	0	0	0	1		42	5	3
Q	01010001	0000	1101	00	11	0	1	1	-1	8	43	**7**	5
R	01010010	0000	1101	00	11	0	1	1	5	8	44	6	4
S	01010011	0001	1101	01	11	1	1	1	5	8	45	6	4
T	01010100	0000	1110	00	10	0	0	0	1	7	46	6	4
U	01010101	0000	1111	00	11	0	1	1	-1		47	**6**	4
V	01010110	0000	1111	00	11	0	1	1	1	1	48	6	4
W	01010111	0001	1111	01	11	1	1	1	1		49	5	3
X	01011000	0000	1110	00	10	0	0	0	5	7	50	6	4
Y	01011001	0000	1111	00	11	0	1	1	9		51	**6**	4
Z	01011010	0000	1111	00	11	0	1	1	5		52	5	3

1	00110001	0100	0101	10	11	0	1	**1**	9		53	**6**	4
2	00110010	0100	0101	10	11	0	1	**1**	4	4	54	6	4
3	00110011	0101	0101	11	11	1	1	**1**	0	8	55	5	3
4	00110100	0100	0110	10	10	0	0	**0**	3		56	5	3
5	00110101	0100	0111	10	11	0	1	**1**	-1		57	**6**	4
6	00110110	0100	0111	10	11	0	1	**1**	3		58	5	3
7	00110111	0101	0111	11	11	1	1	**1**	1	8	59	6	4
8	00111000	0100	0110	10	10	0	0	**0**	0		60	4	2
9	00111001	0100	0111	10	11	0	1	**1**	0		61	4	2
0	00110000	0100	0100	10	10	0	0	**0**	2		62	5	3
`	01100000	0000	1100	00	10	0	0	**0**	2	2	63	6	4
~	01111110	0110	1111	10	11	0	1	**1**	3	7	64	6	4
!	00100001	0000	0101	00	11	0	1	**1**	2	8	65	6	4
@	01000000	0000	1000	00	00	0	0	**0**	6		66	5	3
#	00100011	0001	0101	01	11	1	1	**1**	0		67	4	2
$	00100100	0000	0110	00	10	0	0	**0**	3		68	5	3
%	00100101	0000	0111	00	11	0	1	**1**	2	2	69	6	4
^	01011110	0010	1111	00	11	0	1	**1**	1		70	5	3
&	00100110	0000	0111	00	11	0	1	**1**	3	8	71	6	4
*	00101010	0000	0111	00	11	0	1	**1**	4	4	72	6	4
(	00101000	0000	0110	00	10	0	0	**0**	0	8	73	5	3
)	00101001	0000	0111	00	11	0	1	**1**	0	0	74	5	3
-	00101101	0010	0111	00	11	0	1	**1**	2		75	5	3
=	00111101	0110	0111	00	11	0	1	**1**	0	7	76	5	3
_	01011111	0011	1111	01	11	1	1	**1**	5	5	77	6	4
+	00101011	0001	0111	01	11	1	1	**1**	4		78	5	3
[	01011011	0001	1111	01	11	1	1	**1**	5		79	5	3
]	01011101	0010	1111	00	11	0	1	**1**	9	7	80	**7**	5
{	01111011	0101	1111	11	11	1	1	**1**	1		81	5	3
}	01111101	0110	1111	10	11	0	1	**1**	0	7	82	5	3
\	01011100	0010	1110	00	10	0	0	**0**	5		83	5	3
|	01111100	0110	1110	10	10	0	0	**0**	0	7	84	5	3
;	00111011	0101	0111	11	11	1	1	**1**	0		85	4	2
:	00111010	0100	0111	10	11	0	1	**1**	4		86	5	3
‘	00100111	0001	0111	01	11	1	1	**1**	3	3	87	6	4
“	00100010	0000	0101	00	11	0	1	**1**	3		88	5	3
/	00101111	0011	0111	01	11	1	1	**1**	4	4	89	6	4
?	00111111	0111	0111	11	11	1	1	**1**	5		90	5	3
.	00101110	0010	0111	00	11	0	1	**1**	3	4	91	6	4
>	00111110	0110	0111	10	11	0	1	**1**	1		92	5	3
,	00101100	0010	0110	00	10	0	0	**0**	2		93	5	3
<	00111100	0110	0110	10	10	0	0	**0**	6		94	5	3
SPACE	00100000	0000	0100	00	10	0	0	**0**	0		95	4	2
	00001010	0000	0011	00	01	0	1	1			96	1	1

Table A.8: The LDC reference table: String to binary conversions, compression and logic.

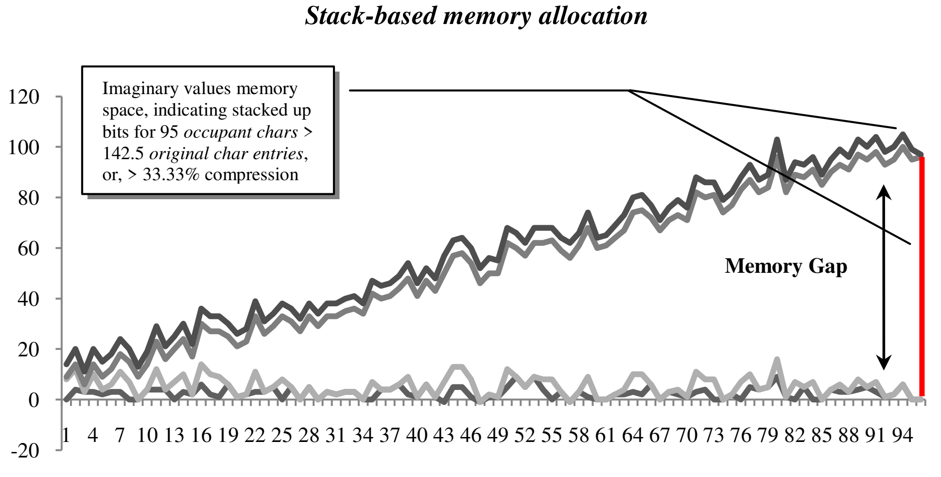


Fig. A.11: The memory gap for every erected stack of 96 chars, filled up with 0 bit values

The C99 standard requires the allowable data types for a bit field to include qualified and unqualified **bool**, **signed int**, and **unsigned int** [37]. In addition, this implementation supports the following types

- $\boxed{\texttt{int}}$
- `short, signed short, unsigned short`
- `char, signed char,` $\boxed{\texttt{unsigned char}}$
- `long,` $\boxed{\texttt{signed long}}$`,` $\boxed{\texttt{unsigned long}}$
- `long long, signed long long, unsigned long long`

In all implementations, the default integer type for a bit field is **unsigned**. Considering these data types are very important to handle data allocation and information size issues for validating LDC and LDD of the algorithm. For properly conducting this, we mostly used '`int`' and '`unsigned char`' data types (as boxed-in), to properly shift the targeted bits in compacted forms of integrity. The operators for conducting a compaction technique on the 1-bit flags, incorporate bitwise AND and OR operators as previously above, in our code written in C. This emphasizes on the powerful ability in using AND/OR combinatorial logic, as the main motive supporting middleware logic of fuzzy and in special, quantum logic handling 8-bit states simultaneously.

The C compiler automatically packs the above bit fields from Table A.8 (Flag 1 and 2 columns), as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case, then some compilers may allow memory overlap for the fields whilst other would store the next field in the next word. This further concerns the 'portability' issue of bits depending on memory architecture. In C, the polarity structure could be specified per string char entry, and thus the numbers of bits representing each entry occupy values less than 8 bits binary, can be specified. This permits the code to access from a particular memory address by assigning a pointer of the above flag structure to later access the memory. In other words, each field is accessed and manipulated, as if it were an ordinary member of a structure. The keywords signed and unsigned mean what you would expect, except that it is interesting to note that a 1-bit signed field on a two's complement machine can only take the values 0 or –1.

Flag type exception handling: Let flag type 9 be a post-setting possible for a customized 1-bit flag due to possible memory access, then we customize possible attribute combinations prior to {1,2,3,4,5,6,7,8}, with either of them in the set. Similarly, this infers to –1 as a presetting for a customized bit combination without repetitions of values within the rows or bits tuples set. For compression purposes, we eliminate the ones which have identical bit pairs (nibbles) output, to avoid allocating an extended 1-bit flag on bit pair impurity, unless needed otherwise. There are at least three ways to manage our flags if not treated self-embedded in our memory system:

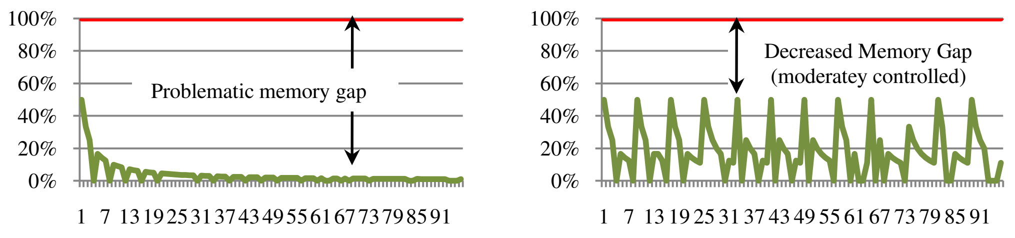


Fig. A.12: The memory gap for every erected stack of 96 chars, filled up with 0-bit values

a) **Static access:** Large memory gap between 0 and 1 results of maximum compression (the figure on the left).
b) **Partitioned access:** Optimized gap allocation on flags for 0 and 1 results of maximum compression (the figure on the right).
c) **Static, partitioned and dynamic access of a GC file:** Reliably confident access due to predicted combinatorial access of self-embedded flags in the **GC** flags for all impure, pure and their complement on every pair of bits. Self-embedded flag access (or even non self-embedded), by this method, gives highest efficiencies possible for the hardcoded flags returning original chars in dictionary.

A highly-efficient method for accessing GC flags: The structure for the c) solution, is a hypercube (grid's model) with fuzzy quantum complex number relationships. For example, the non self-embedded flags dynamic access is already shown in Fig. A.10. Fig A.9 however, is a good example on self-embedded flags. The latter is more resilient for memory management. Furthermore, it is quite efficient and more advantageous in memory access methods, since all 1-bit flags are hardcoded in a translation table (dictionary) for the **GC** file contents, in terms of a unique identity for input chars. This makes address access quite precise and the least complicated for the interpreter during row address to address comparisons. In addition, the uniqueness of the translation table is that one could standardize the data compression output by establishing combinations of ‘bits and flags’ in ‘if loops’ within a ‘for loop’ integration for each 96 single bit spaces of memory.

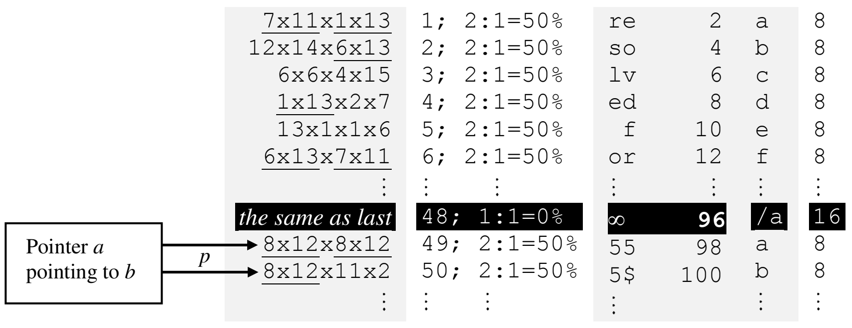
To avoid repetition of flag combinations, we refer to the table’s set theory elements. Possible combinations are given in these sets with a maximum cardinality of 10, and a minimum cardinality of 1 in practice. So, once a flag number, gives an identical combination of the previous rows, we, as the role of LDC flag combinator or fixed point analyzer e.g., use the anonymous recursion (fixed point combinator) concept to choose another flag number from its set, based on non-combinatory preference of bits. Here is an example:

Once again, let’s have a look at Table A.6. By paying more attention to the input string sample ‘`resolved`’, we realize that the grid row address, is partly repeated for those chars that are recurring in occupation (Fig. A.13). Of course, we can leave the program (its interpreter) to do the extensive top-to-bottom comparisons for all 95 chars iteratively per end-of-each char-block for the row range # 1 to 65,536. But it is undoubtedly very efficient if a pointer spots and tags all of the recurring static addresses, in this case, the ones spotted are underlined based on the restricted commutative rule for all founded 4×1-bit flags. Therefore, the pointer p for this order returns a set of repeated flags by reference:

{(7x11, 1) p (7x11, 6)}, {(1x13,1)p(1x13,4)}, {(6x13, 2)p(6x13, 6)}, {(8x12, 49)p(8x12, 50)}, {(8x12, 49)p(…)}

representing chars: {(r, 1) p (r, 6)}, {(e, 1)p(e, 4)}, {(o, 2)p(0, 6)} , {(5,49)p(5, 50)}, {5, 49}p{…}, correspondingly.

Once tagged, the ‘pointer flags’ in a total of n_p = 5, are packed up for five respectively-distinct addresses in the memory, while the remaining identities are being checked during original char reconstruction process. Once data reconstructed, the flags that are packed and residing temporarily in memory, are appended to the reconstructed data lines according to record. This however, requires one further issue to consider, and that is, keeping the record on data reconstruction intact until the packed data is recalled from the memory. From there, temporary records are deleted and thus, the decompression phase is said to be completed.



7x11x1x13 1; 2:1=50% re 2 a 8
12x14x6x13 2; 2:1=50% so 4 b 8
6x6x4x15 3; 2:1=50% lv 6 c 8
1x13x2x7 4; 2:1=50% ed 8 d 8
13x1x1x6 5; 2:1=50% f 10 e 8
6x13x7x11 6; 2:1=50% or 12 f 8
the same as last 48; 1:1=0% ∞ 96 /a 16
8x12x8x12 49; 2:1=50% 55 98 a 8
8x12x11x2 50; 2:1=50% 5$ 100 b 8
Pointer a pointing to b
p

Fig. A.13: A dynamic pointer for an efficient self-embedded 4-bit flag static address allocation

Further compression, of course, as the columns attain the topmost highest layers of the algorithm, indicates that quantum information and computation is in place. The convolution of quantum design and technology is already illustrated and briefly explained in Appdx. C. FBAR’s LDC, in context, deals with the computation of binary logic regardless of content size and type, whereas other techniques are not bothered about. Binary logic in FBAR, deals with individual bits, their combination, repetition and their conservation of information, regardless of character repetition or content type. This means, based on a fixed size character reference table, Table A.7, we derive a new, and a more certain equation (least zero order H values), which is logarithmically the least probabilistic with discrete entropy (bits per character), compared to Shannon’s entropy rate on English alphabet as follows:

$$H_{\mathcal{A}} = \log_2 m \; = 4.75 \text{ bits/char}, \qquad (6)$$

and for higher orders of H for a given text source made up of English alphabet letters, becomes 4.07, 3.36, 2.77 and 2.3 bits/char, respectively. In contrast, the entropy rate for possible versions of FBAR based on the C_r columns in Fig. A.13 (or Table A.7), is computable by the following equation

$$H_{\wedge\vee(b)} = \log_b |\beta| = [0, 2.4[\text{ bits/byte} \tag{7}$$

where H is the entropy devoted to binary b probability over two 0 and 1 states within the context of and-or fuzzy binary logic. The entropy is computed on a binary sequence β, e.g. β =0001 with a cardinality of a proximate of a positive integer number. In this example, $|\beta|$ = 4 or, a nibble equivalent. Thus, for a binary sequence β, the binary probability of two states, b = 2, constructing 1 char, higher orders of H as we see within the resulted interval, becomes 2, 1 and 0 bits/byte, regardless of source for a given fixed size LDC binary reference code.

The upcoming section shows how to retain necessary data based on the bit flag reference table (Table A.6 for high level compression, Table A.8, for the lower levels), hence making a lossless data decompression possible during the experiment.

The prototype after code compilation creates a grid object (a portable file on driver) which contains information about the flag-set class under test. In addition, this object could be stored as a grid file **G**, a compressed version of all possible combinations on 4-bit flags per input char. The **G** file uses the compressed file **C** to decode, and thereby decompress data, losslessly.

$$\left.\begin{array}{l}
\text{r} \xrightarrow{\text{in}} \boxed{\mathbf{P}} \rightarrow 01110010 \rightarrow \boxed{0100}\boxed{1101} \rightarrow \boxed{10}\boxed{11} \rightarrow \boxed{0}\boxed{1} \rightarrow 1 \\
\text{e} \xrightarrow{\text{in}} \boxed{\mathbf{P}} \rightarrow 01100101 \rightarrow \boxed{0000}\boxed{1111} \rightarrow \boxed{00}\boxed{11} \rightarrow \boxed{0}\boxed{1} \rightarrow 1 \\
\text{s} \xrightarrow{\text{in}} \boxed{\mathbf{P}} \rightarrow 01110011 \rightarrow \boxed{0101}\boxed{1101} \rightarrow \boxed{11}\boxed{11} \rightarrow \boxed{1}\boxed{1} \rightarrow 1 \\
\text{o} \xrightarrow{\text{in}} \boxed{\mathbf{P}} \rightarrow 01101111 \rightarrow \boxed{0011}\boxed{1111} \rightarrow \boxed{01}\boxed{11} \rightarrow \boxed{1}\boxed{1} \rightarrow 1 \\
\text{l} \xrightarrow{\text{in}} \boxed{\mathbf{P}} \rightarrow 01101100 \rightarrow \boxed{0010}\boxed{1110} \rightarrow \boxed{00}\boxed{10} \rightarrow \boxed{0}\boxed{0} \rightarrow 0 \\
\text{v} \xrightarrow{\text{in}} \boxed{\mathbf{P}} \rightarrow 01110110 \rightarrow \boxed{0100}\boxed{1111} \rightarrow \boxed{10}\boxed{11} \rightarrow \boxed{0}\boxed{1} \rightarrow 1 \\
\text{e} \xrightarrow{\text{in}} \boxed{\mathbf{P}} \rightarrow 01100101 \rightarrow \boxed{0000}\boxed{1111} \rightarrow \boxed{00}\boxed{11} \rightarrow \boxed{0}\boxed{1} \rightarrow 1 \\
\text{d} \xrightarrow{\text{in}} \boxed{\mathbf{P}} \rightarrow 01100100 \rightarrow \boxed{0000}\boxed{1110} \rightarrow \boxed{00}\boxed{10} \rightarrow \boxed{0}\boxed{0} \rightarrow 0
\end{array}\right\rangle 11110110 \equiv \text{F6h} \equiv \ddot{\text{o}} \xrightarrow{\text{store to}} \boxed{\mathbf{C}}$$

Fig. A.14: An example of the generated outcome delivered by the FBAR prototype

Consequently, further compression values within a certain range of the grid between memory grid row address # 1x1x1x1 and 16x16x16x16, from Table A.7, give predictable reference values on original char combinations prior to the ANDed and ORed columns via 4-bit flags set. This memory row number is static, and each address is self-embedded for at least 4×1-bit flag representative. Each char representative gives a 2-Char output as a reconstructed data to the original. However, the grid file size is dynamic when it comes to occupying space (size), as explained for grid file specifications in § A.1.2. This makes data quite portable between machines, and program access during compressions from layers 1 to 4 of the FBAR algorithm. In continue, the closures of Boolean states of 01 and 10 as 1 and 0 for the 2nd product of the 4th layer, respectively, alleviate dictionary index complications upon recalling the memory address to reconstruct data during the decompression phase of the algorithm. For explicit recall of values on this level of compression, for the lower bound of the table on the memory address, between index # 47 and 95, requires a pointer to the relevant index of the grid file raising the particular relevant bit that once was a pair of either, 00, 01,10, or 11, before the further-generated closure states. The result, according to Eq. (7), at this stage is between, 1-to-2 bit/byte, expectably.

A.2 Data Compression

Data Compressors produce compressed input values for their input chars from a targeted document loaded to the program. The set of possible input types is large, ranging from simple data types such a numbers, to more complex data such as a combination of numbers, strings and binary. Also the domain of input values for a specific type can be quite large. Finding appropriate values, is a very important although, a difficult task. In this section we present a possible approach for the compression of input string serving FBAR logic for a lossless compression.

A.2.1 Characteristics of a Lossless Data Compressor

A Lossless Data Compressor has to cover a large domain of types and input values. It must compress data and decompress successfully in terms of no data loss during bitwise conversions. One could say: a lossless main characteristic is to preserve well-defined entropies between data I/O points. It is probably too much for a single prototype to cover all these different possibilities. Therefore, a major design decision was to have different functions programmed for specific problems e.g., data conversion, memory allocation, encoding , etc. to fulfill a final step of a lossless data compression. The responsibility lies with each single bit generated or inputted as our quantitative input/output (I/O) values. Since

$$\therefore b \in H_{\mathcal{A}}, m = 216 \text{ thus } H_{\wedge\vee(b)} = 7.755 \text{ bits/byte} \quad (8)$$

The English alphabet letters in total possess a length of 216 bytes, its entropy rate of probability concentration is thus 7.755 bits/bytes when dealt with binary *b* values only. This measurement is inappropriate to some extent for a full sequence of letters within Shannon standards, and not the plausibility of each character that subsist on its combinatorial binary AND/OR logic. However, if the x86 machines were to be a quantum machine, this scenario would have changed dramatically, handling impure 01 and 10, and pure 11 and 00 states simultaneously i.e. 8 different probable states on sequences of binary per byte. In other words, $H_{\wedge\vee(b)}$ as H_q would give 2.585 for a base 8, convenient to apply the FBAR model with least order of probability in counting bits compared to chars. So, by applying FBAR, we can say that the probability of binary *b*, decreases its entropy *H* significantly, giving last orders of 0 bits/byte for each reconstructible character, according to Eq. (7). Therefore, to tackle such drastic odds, like the one given in Eq. (8), performed on an x86 machine, we define the problem specific losslessness of a data compression in terms of having a comparator, and a compressed grid file as the main components of our process design.

A.2.2 Defining problem-specific losslessness of a data compression

When writing a new lossless compressor, there a few things that a developer must consider. Every lossless compressor must include the compaction technique of single flag bits. For our grid **G** file, during compression, in C, we leave the compiler to compact our possible bit fields, pertinent to **ip** and **zn** dimensions of **G** for every corresponding compressed char in it, prior to the dynamic **C** file.

```
    Struct packed_struct {
                    unsigned int f1:1; // for ip dimensions of the G file
                    unsigned int f2:1; // ...
                    unsigned int f3:1; // ...
                    unsigned int f4:1; // ...
                   unsigned int f5:1; // for zn dimensions of the G file
                   unsigned int f6:1; // ...
                   unsigned int f7:1; // ...
                   unsigned int f8:1; // ...
    } pack;
```

Here, the `packed_struct` contains 8 members: four 1 bit flags f1…f4, for probable **ip** combinations, the remaining flags for a negation possibility upon the previous flags if, and only if, raised per combination. The total bit resultant is usually 5 to 6 bits, and maximally, considering the **C** file char constituents, 7 bits. The average score of bits normally corroborates with our spatial LDC estimate of 37.5%. The C compiler automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case then some compilers may allow memory overlap for the fields whilst other would store the next field in the next word (see comments on bit files portability below). Access members as usual via:

```
    pack.type = 6; // access member no.6
```

Note: Only *n* lower bits will be assigned to an *n* bit number. So `type` cannot take values larger than 15 (4 bits long). Bit fields are always converted to integer type for computation. We are allowed to mix "normal"' types with bit fields. The unsigned definition is important - ensures that no bits are used as a ± flag.

A.2.3 FBAR Compressor compared to Standard Data Compressors

The FBAR algorithm broadly comprises of two major components: **1-** the dictionary, and **2-** the compressed grid file or **GC**. The dictionary on its own, consists of 4x1-bit flag addresses, and is the key to the grid's file content's identity. Once this identity is compared and linked with the right identity in the dictionary, original data is reconstructed at the LDD phase of the algorithm. Other lossless compressors don't have this feature available, and thus act differently based on probabilistic factors for character detection and identification.

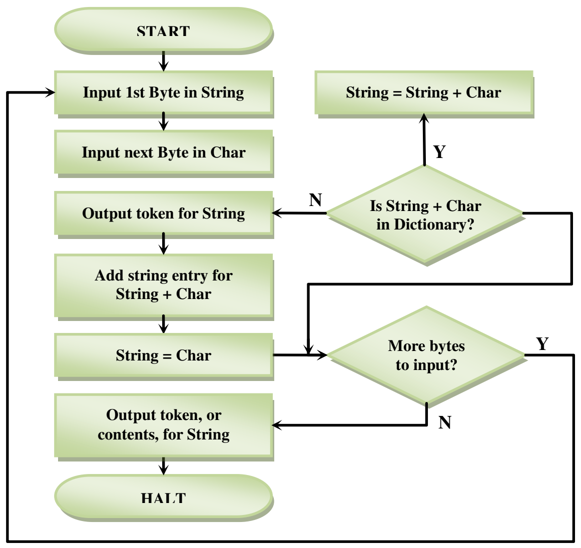


Fig. A.15: Flowchart of an LZW algorithm for an LDC. (Courtesy of [40] and [47])

As shown in this flowchart, Fig. A.15, the LZW compression algorithm is designed to input data, accumulate it, generate a dictionary that assigns tokens and outputs them into a compressed format. Lempel "Ziv" Welch (LZW) [6, 7], is a lossless data compression technique that was created back in 1984 by Terry Welsh [8], as an improvement to the popular LZ77 compression algorithm. The following is a pseudocode extracted from the LZW fundamentals. A quick examination of the algorithm shows that LZW is always trying to output codes for strings that are already known, and each time a new code is output, a new string is added to the string table [46]:

```
STRING = get input character
WHILE there are still input characters DO
    CHARACTER = get input character
    IF STRING + CHARACTER is in the string table then
        STRING = STRING + character
    ELSE
        output the code for STRING
        ADD STRING + CHARACTER to the string table
        STRING = CHARACTER
    END of IF
END of WHILE
output the code for STRING
```

A sample string used to demonstrate the algorithm is shown in Table A.9. The input string is a short list of English words separated by the '/' character. Stepping through the start of the algorithm for this string, you can see that the first pass through the loop, a check is performed to see if the string "/W" is in the table. Since it isn't, the code for '/' is output, and the string "/W" is added to the table. Since we have 256 characters already defined for codes 0-255, the first string definition can be assigned to code 256. After the third letter, 'E', has been read in, the second string code, "WE" is added to the table, and the code for letter 'W' is output. This continues until in the second word, the characters '/' and 'W' are read in, matching string number 256. In this case, the code 256 is output, and a three character string is added to the string table.

Input String = /WED/WE/WEE/WEB/WET

Character Input	Code Output	New code value	New String
/W	/	256	/W
E	W	257	WE
D	E	258	ED
/	D	259	D/
WE	256	260	/WE
/	E	261	E/
WEE	260	262	/WEE
/W	261	263	E/W
EB	257	264	WEB
/	B	265	B/
WET	260	266	/WET
EOF	T		

Table A.9: The LZW compression process. (Extracted from [46])

The process continues until the string is exhausted and all of the codes have been outputted. The sample output for the string is shown in Table A.9 along with the resulting string table. As can be seen, the string table fills up rapidly, since a new string is added to the table each time a code is output. In this highly redundant input, 5 code substitutions were outputted, along with 7 characters. If we were using 9 bit codes for output, the 19 character input string would be reduced to a 13.5 byte output string. Of course, this example was carefully chosen to demonstrate code substitution. In real world examples, "*compression usually doesn't begin until a sizable table has been built, usually after at least one hundred or so bytes have been read in*." On the other hand, the FBAR does not construct a new table for each new file load execution. The fact is, since values are prefixed as self embedded 1-bit flags, hence it would not necessary to reconstruct data according to new dictionary versions to the grid code. The pseudocode for an FBAR LDC is given in terms of:

```
1.  WHILE there are still input characters DO
2.  CHARACTER = get input character
3.  CONVERT CHARACTER to BIN CHARACTERS
4.  PACK 1-bit FLAGS from any conversion level
5.  IF PACK + CHARACTER is in the Reference Table
6.  THEN
7.  PACK = PACK + CHARACTER in the G file
8.  ELSE
9.  OUTPUT the code for PACK as NEW STRING
10. ADD NEW STRING + BIN CHARACTER to the C file
11. NEW STRING = CHARACTER
12. END of IF
13. END of WHILE
14. OUTPUT the code for PACK in G file
15. OUTPUT the code for NEW STRING in C file
```

This delivers characters irrespective to any repetition of them in a given input string to the program. A simple glance at the pseudocodes, one could realize that the FBAR approach is in contrast with LZW and operates autonomous from LDC predecessors. The main difference is in the abstraction of the code in binary with respect to input string, rather than chars as string constituents. The building of a table is not as same as LZW, and as illustrated in Fig. A.9, is compressing the logic states of input string into a 4D logic constructor grid for each occupant char related to occupant chars in Table A.7.

A.3 Data Decompression

To reconstruct information from the prototype, we dereference data by program's LDD subroutine. The subroutine comprises of an interpreter filled with essential if-else statements, comparing the stored data in the **GC** file with the dictionary file containing the translation table (Table A.7). The dereferencing function within the program, once finds a match between the two files, returns char values in a new file as reconstructed characters, just like before as it's suppose to be in the original file.

As shown in Fig. A.16 below, through the use of an iterative process, the decompression algorithm is responsible for reading in the compressed data and converting it back to its original form by *dynamically* replicating the compression program's dictionary. In contrast, the *static approach* from LDC to LDD is later given in four steps after Fig. A.16, below. The decompression program starts with the first entries of its dictionary initialized to all the characters in the original data. It then reads each character from the compressed data, which are merely pointers to the dictionary, and uses each pointer to retrieve uncompressed character strings from its dictionary and writes them to a decoded output buffer. It also builds its dictionary in the same way as the compression program.

The lossless FBAR decompression model for an x86 machine upon pure 'unitary states' to reconstruct data, i.e. the 0 state building 00, the 1 state building 11, considering that at least we have the FBAR impure states 01 and 10, is pursued upon recalling 1-bit flags according to the following program and matrix relationships:

To reconstruct data, three procedures are taken consecutively. Each procedure comes from a separate constructor associated with the dictionary. The first constructor representing the first procedure is the 'logic constructor grid' **G**, which during the LDC timeframe recorded the impure and pure state logic with respect to the closure states of 1 and 0 during compression. This timeframe should have covered in its recorded data, the AND/OR conversions, which are columns 4 to 6 of the above matrix flow. The basics of the "constructor grid" in conducting a 'triangular logical operation', has already been explained in § A.1.2. Nevertheless, a logic constructor grid excerpt on the string '`resolved`' gives about an experienced release over LDD according to the dereferencing procedure (denoted by an "*" in C-like languages) on data. The 8-char word, '`resolved`', is now compressed into an 8-bit binary, equivalent to a single length character ö, from the 8-bit extended ASCII table, so we expect the FBAR logical procedures have already been carried out before decompression.

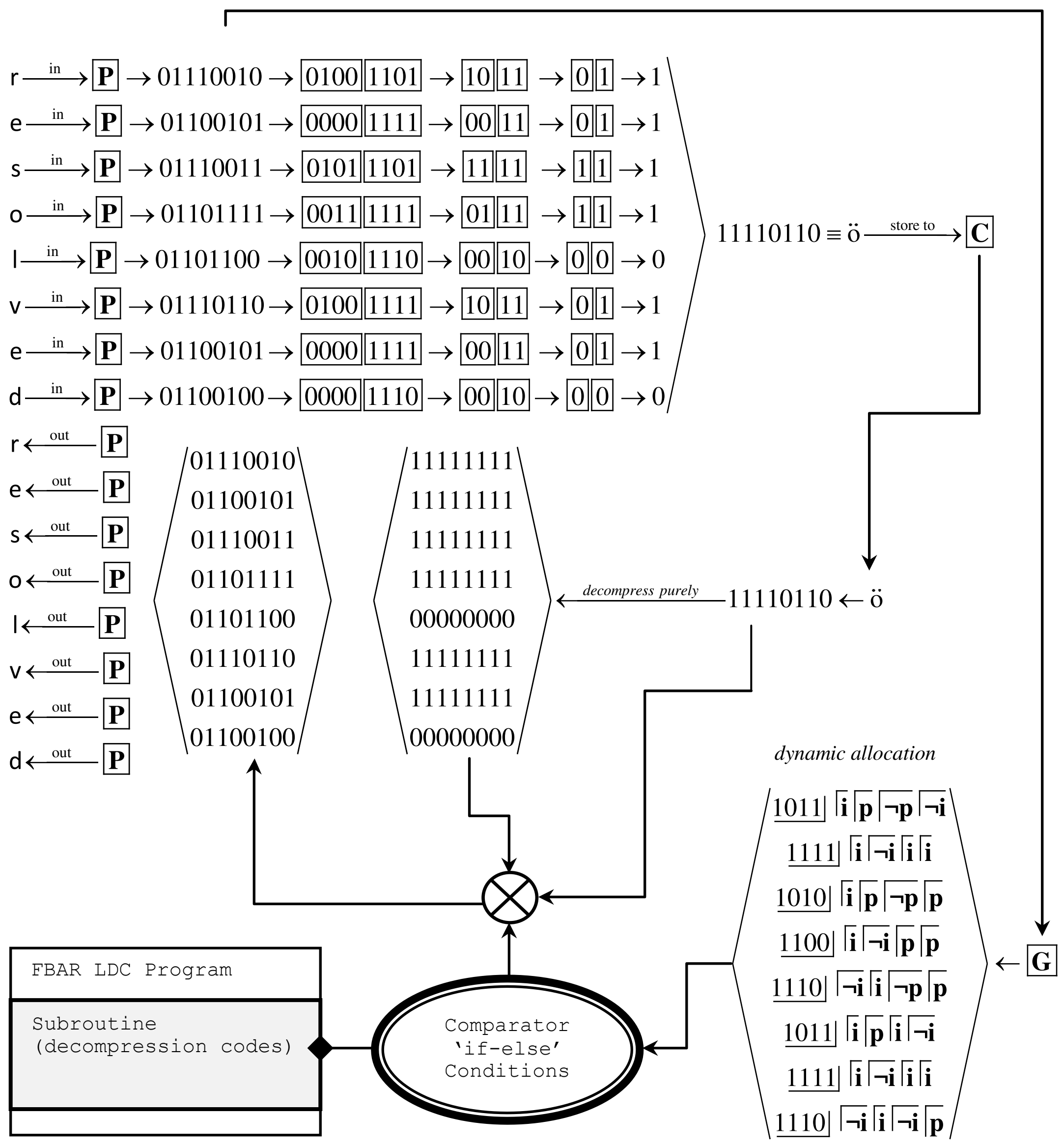


Fig. A.16. An FBAR LDD conducted by the program's interpreter, here as a comparator

The memory grid file for each bit of the 8 char entries, gave us a 4-bit length algorithmic product to decode and decompress data. Hence, for the sample, a total of 5 bits per char entry, denoting a 37.5% space savings reserved. The comparator of the source file deals with the last procedure, checking the logic between the compressed file and the grid address to decompress data as a subroutine of the program. An example of this could be given by the following pseudocode:

```
READ the GC file row-by-row from end-of-file
OUTPUT temporary ROW_CHARACTERS
OUTPUT temporary (ROW_NUMBER == ROW_ADDRESS)
CHARACTER = ROW_CHARCTER
WHILE reading CHARACTER by CHARCTER DO
READ ROW_NUMBER
IF CHARCTER is not in the (ROW_ADDRESS AND CHARACTER) of translation table with BIN
CHARACTER
THEN
STRING = get translation of OLD_CODE
STRING = STRING + CHARACTER
ELSE
STRING = get translation of NEW_CODE
END of IF
OUTPUT STRING
CHARACTER = 1st or 2nd or … or nth 2 characters in
STRING
REPLACE CHARACTER with 2 new characters from the
translation table
OLD_CODE = NEW_CODE
DELETE temporary ROW_NUMBER and ROW_CHARACTERS
END of WHILE
```

In reality, each dimension of the grid solely requires 4 bits per char entry and intersects with the other dimensions as other char entries. With the compressed char in the compressed file **C**, 1 bit is allocated. Therefore, 5 bits per character denoting a 37.5% space savings is required. To avoid occupying an 8 bit full occupation, it suffices to put two neighboring compressed chars, each with 4 bits in the grid file **G**, into one byte character, thus for 2*chars* giving 1*byte* + 2*bits* = 1*char* + ¼ *char*. A more efficient programming would even reduce this dependency of a single bit allocation for every char in the **GC** version (Fig. A.9). That is, to make the **G** file totally independent of the dynamic version (**C** file) when the comparator is engaged during decompression. So, by condensing the nested 'if loop' conditions of the comparator, e.g., coding 0000000 as 0 and 11111111 as 1, with better flag combinations to create the original char, would create a self-embedded dictionary as a robust reference point of reduced packs of bits and nibbles. Interpedently, the more self-embedded references conditioned in the comparator, the more flags are managed, and thus higher states of LDC values. However, this must not hamper the way in which the temporal behavior on memory management issues is handled due to hardware architecture and instructions limit. In fact, for a quantum model, compared to the current 4D model on an x86 machine, combined with a 360^o Bloch sphere [42] satisfies any state of logic in pairwise forms. Meaning that, the newly-emerging *n*D-model becomes very useful to compute data for a decompression. In other words, the compressed file, with the dismissed bits of the tabs and spaces of the grid, occupying 8 bits per se, is embedded into the extra information space (the grid) required to reconstruct the original data, is thus guaranteed with a quantum CPU followed by its instructions unit. We have considered this in our design for maximum LDCs of the algorithm. The simplistic steps of data reconstruction (a successful LDD), by setting a default value in the dynamic compressed file (or source code), at the LDC phase, for the sequencer achieving double-efficient *C*'s ≥ 50%, is given as follows:

1. The following is a pure sequence for the input chars. We set this always as default in the FBAR program

   11111111 (*LDC upwards*)

2. Suppose the original input char is

   @

3. In binary, according to ASCII is

   01000000

   (*the goal is to manipulate 11111111, to obtain 01000000*)

4. So the combination in terms of **znip** relative to pure sequence closures on each pair from MSB to LSB, is

   i p p p (11 11 11 11) → 01 11 11 11 → *then* (*static allocation*)
   z n n n (01 11 11 11) → 01 00 00 00 → @ (*LDD, achieved*)

Therefore, we say, the key for an LDD is having the unique combination **ippp** **znnn** to return the @ character. One could decipher this flag combination in terms of address *xxy* (two dimensions out of four dimensions) by referring to the 4D bit-flag model (Fig. A.7). Of course, the @ example must be followed with another original input char to have a pure double-efficiency at the LDD phase of FBAR. This of course is detected by the program's interpreter/comparator once a translation table (Table A.7) read is made by the program. The collapsed version of the previous pseudocode, obeying the double-efficient four steps, on the string '`resolved`', would be

```
WHILE reading CHARACTER by CHARCTER AND compressed BIN CHARACTER is '1' DO
READ CHARACTER as last block character
IF CHARACTER is a block character THEN
READ CHARACTER prepositioned to block character
READ ROW_NUMBER
GET ROW_ADDRESS from translation table
IF (CHARCTER ='d' AND ROW_ADDRESS = '1x13x2x7')
OUTPUT STRING ='ed'
ELSEIF (CHARCTER = 'c' AND ROW_ADDRESS = '6x6x4x15')
OUTPUT STRING ='lv'+'ed' = 'lved'
ELSEIF (CHARCTER = 'b' AND ROW_ADDRESS = '12x14x6x13')
OUTPUT STRING ='so'+'lved' = 'solved'
ELSEIF (CHARCTER ='a' AND ROW_ADDRESS = '7x11x1x13')
OUTPUT STRING ='re'+'solved' = 'resolved'
ELSE
PRINT no data or null compressed
END of IF
ELSE
PRINT no block character in range
End of IF
END of WHILE
```

The same comparison of an inclusion technique via negation, pure, impure logic is consistent to the previous sub-layers of the FBAR compression for the encodings, when the program attains layer # 4 of the LDC. So, we use the 'bit field' approach rather than 'bit array', lees than 8 bit space occupation. The hypothetical outline of these relationships between the increase of complexity of the comparator on x86 machines with memory allocation limits, leading to hybrid versions of FBAR as FBAR/FQAR and pure FQAR are given in § B.3, Appdx. B. Of course, for either the hybrid or pure version, we had to contemplate the impure pairs 01 and 10 in terms of a quantum encoding procedure for an arbitrary future-state quantum computer (QC), constructing quantum trees on logic states and their relationships. The rationale to this structuring of data is in with accordance to Eq. (1)'s interrelatedness behavior of states, and the prime objective is in achieving values of Eq. (7) on x86 machines.

A.4 Test Cases

The test case generator is the core of conducting comparison experiments. It first collects information from the universe of discourse on information input observed to an LDC algorithm, irrespective to the encoded, decoded and decompressed data. It could be done manually by selecting *random documents*, thereby *conducting LDC by a relevant package of choice*. The results of comparisons are ranked afterwards, through Freidman's test as a nonparametric comparisons method (Appdx. B). However, an automated version of package selector as a comparator could be coded for accurate test case generators on sample documents.

We have selected those documents that were compatible with the evolutionary pattern of our algorithm. Basically, we needed to reconstruct char-based data at first (Table A.10), so to investigate character reconstruction possibility within the context of FBAR logic. Of course, after fulfilling this feature of the algorithm, it is possible to advance the algorithm for its reception on any data type even compatible with the machine language and its respective compiler. In Appdx. B, we report our results on small and large input data for our statistical test. Small input data allows accurate comparisons between original chars during the input phase, compression and decompression. The following were our selective choice of samples (twelve in total), pertinent to the algorithmic evolutionary requirements:

No	File	Type	Size (bytes)
1	text	.txt	61608
2	book1	.txt	678244
3	book2	.txt	1772074
4	paper1	.txt	52516
5	paper2	.txt	117493
6	paper3	.txt	10262
7	web1	.htm	747766
8	web2	.htm	598125
9	log	.txt	1840924
10	cipher	.txt	777654
11	latex1	.tex	209212
12	latex2	.tex	155641

Table A.10: Relevant sample documents for LDC algorithmic comparisons

To see whether data is reconstructed successfully, the output is therefore compared with its original. From there, it is logical to make test-runs on large input data or file(s), since data integrity evaluations are conducted during small sample runs.

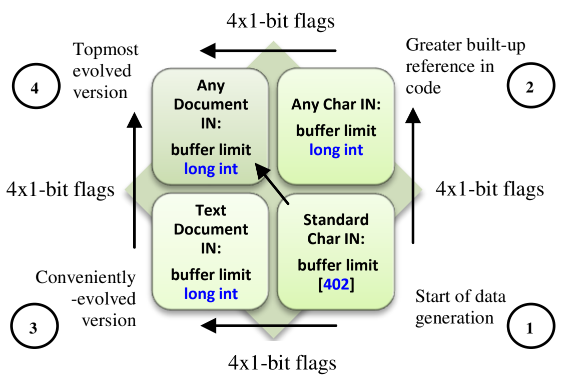

4x1-bit flags
Topmost evolved version
4
Greater built-up reference in code
2
Any Document IN: buffer limit long int
Any Char IN: buffer limit long int
4x1-bit flags
4x1-bit flags
Text Document IN: buffer limit long int
Standard Char IN: buffer limit [402]
Conveniently -evolved version
3
Start of data generation
1
4x1-bit flags

Fig. A.17: Input data types used for a set of test-runs

Working with large samples on the first runs would be extremely complicated and almost impossible to manage per input document. Once char integrity evaluated on the smallest scales possible with certain buffer limit, assigning string size to the counter variable, building up the sample, would result in manageable flows, and easy validation on data results. The '`long int`' limit in Fig. A.17, is integrated within 'code loops' to store occupant chars in the **G** file, as 4-bit flag representatives. In case of an LDD with any size input, through proper access and comparisons of values from the translation table (Table A.7), with the occupant chars within the grid, an evolution of different versions starting with textual type to any data type is achievable. The current FBAR subsists on the three, upper-right, lower-left and lower-right (starting point) of the matrix, evolving toward the last version of any document type beyond the level of chars.

Appendix B

Appendix B describes the steps of the comparisons experiment with FBAR and presents the results. Section B.1 introduces the experiment. The test samples are presented in Section B.2. The results are presented, evaluated and discussed in Section B.3.

B.1 LDC Comparisons Experiment

In this experiment, we test the developed tool (prototype) from Appdx. A, on a number of different test compressors. The design, method and implementation have been all discussed in the previous sections of Appdx. A, which further led to the current LDC comparisons experiment. According to our methods from § A.1.1.2, we want to see whether the presented approach has difference when compared with other compressors used today. The experiment should show, if the tool is applicable and in which achieve a rank. The test samples have been given in Table A.10. The outcome of this experiment is compared with the results obtained by statistical test, and each test sample is tested multiple times. We tested samples in the small and in the large, inclusively.

B.2 Test samples

Test samples are selected to cover certain requirements of our prototype. These requirements are function-based in program code. These functions conduct FBAR logic and must maintain consistency in performance and data conversion factors. Once these factors are being dealt with per program execution, the integrity of input sample is tested for assurance at the point of delivery. In this case, the reconstruction point or where decompression occurs. Test samples could be presented in two forms: **1-** strings and binary, and **2-** documents; each having their own purpose of usage by the program. These test samples are already listed within the test cases context, or see, Table A.10.

B.2.1 Strings and binary

The string sample is a short representation for a good testing strategy carried out by a software prototype. String samples consist of characters, whereby convertible to other data types in programming. A code snippet used in many applications for string and binary conversions could assist in coding the FBAR hypotheses *H.1* to *H.5*, from § A.1.1, once we adapt the code to FBAR encoding standards, and thereby the static dictionary coder indices quantifying binary results back to 8-bit chars. The FBAR prototype satisfied compression products when a user investigates LDC products from its menu options. The enabled/available options are denoted by a YES, otherwise, a NO in Fig. B.1. These options were programmed to satisfy certain objectives of the FBAR algorithm as follows:

```
*****************************************************************************
*** The FBAR Data Compression/Decompression MENU ***
*****************************************************************************
 [1] 1 char-->2 digit-->8 bits...String to Hex/Binary Conversion Display: YES
 [2] 8 bits-->4 parallel bits...Bitwise AND/OR Encoding Product: YES
 [3] 4 parallel bits-->2 parallel bits...A Compression Product: YES
 [4] 2 parallel bits-->1 compressed parallel bit...A Compression Product: YES
 [5] 1 compressed parallel bit-->1 compressed bit...A Compression Product: YES
 [6] Compressed bits-->(n)decompressed bits...A Decompression Product: YES

*****************************************************************************
*** The FBAR File Compression/Decompression MENU ***
*****************************************************************************
 [7] 1 file-->1 compressed file ...A Compression Product: YES
 [8] 1 compressed file--> 1 initial file...A Decompression Product: YES
 [9] (n)files--> 1 compressed file...A Compression Product: NO
 [10] 1 compressed file--> (n)initial files...A Decompression Product: NO

*****************************************************************************
Enter your choice on one of the programmed hypotheses:
```

Fig. B.1: FBAR prototype menu options supporting implementation and simulation versions

The stateful flow representation (denoted by an arrow -->) for each menu option (ranging from 1 to 10) is explicit, and the prototypic implementation followed certain empirical rules of logic to compress data losslessly on a standard x86 machine. The upper section of the menu comprises of conditions assigned to an input string, and thereby after necessary data conversions, its compression product. The lower section of the menu, however, comprises of conditions assigned to an input document, and thus its compression product. Samples of the latter form are the result of carful implementation on the former type, maintaining the integrity of the bit's position and its state dependent on the FBAR static dictionary coder. Of course, this is done on a much larger scale and further explicated in §§ B.2.2 and B.3. One could rephrase this as testing our FBAR compression technique **in the large** indeed (recall, the last paragraph of § A.4).

The reason for using a string as the input in the former type, is to initially demonstrate our compression technique **in the small**, observing data integrity and quantitative constructs, addressing bits, thus verifying their FBAR logic according to our hypotheses before reaching *H.6*. It is at the later stage(s), or, lower frames of the menu, we then load a file (the code would be followed with slight changes containing duplicate calls and method invocations of relevant functions within), as we convert the contents to binary via textualization or batch filing e.g., concrete text format which for processing (compression and decompression) taken in as a set of strings. A very good example is the I/O string, '`resolved`', previously validated in § A.1.2.3. To compose other strings, we simply refer to the ASCII input chars in Table A.8, and expect low-level conversions (encoding) to high-level LDCs, occur in our prototype. These expectations are performed by prototype's menu options 1 through 5.

B.2.2 Documents

Common documents, is an overly large but simple application to test a compression technique's eligibility for delivering data as either lossy or lossless. The lossy type, of course, is never appropriate for textual documents, since information is lost or less detailed in case of converting text to a lossy image, saving more space. This results in more steps of data computation during data conversions resulting significant latency for a successful data delivery in such compressors. Lossless compressors, on the other hand, focus on data integrity and information entropy during any data conversion states. Once the "in the small" samples are tested for each portion and step of program code, then loading documents in the large becomes imminent. One could then investigate the integrity and entropy factors of I/O data with respect to bitrate performance and memory usage.

B.3 Results and Discussion

B.3.1 Test cases and algorithmic characteristics

Table B.1, compares our implementation of algorithm with three other compressors, chosen for their wide availability and their ranked compression ability. The same twelve files were compressed individually with each algorithm, and the results totaled. The bits per character values are the means of the values for the individual files. This metric was chosen to allow easy comparison with figures given via a nonparametric test technique. The choice of this technique is justified as we further explain the test in § B.3.2.

Document	#	WinZip	GZip	WinRK	FBAR	FQAR *
text	1	70.00%	85.70%	87.87%	50.00%	87.5%
book1	2	70.80%	69.00%	80.04%	49.47%	86.57%
book2	3	65.40%	63.80%	77.11%	48.95%	85.66%
paper1	4	65.60%	64.70%	73.58%	50.00%	87.5%
paper2	5	62.80%	61.60%	69.00%	50.00%	87.5%
paper3	6	60.00%	59.50%	68.25%	50.00%	87.5%
web1	7	72.20%	71.40%	75.37%	48.95%	85.66%
web2	8	53.80%	53.60%	54.57%	49.47%	86.57%
log	9	95.59%	95.37%	96.43%	48.95%	85.66%
cipher	10	73.30%	70.30%	77.82%	48.95%	85.66%
latex1	11	70.00%	69.00%	78.28%	50.00%	87.5%
latex2	12	66.52%	66.53%	75.70%	50.00%	87.5%

Table B.1: Test case LDCs based on *space saving* values

* FQAR is the fuzzy quantum version of FBAR, whereas the latter as fuzzy binary, is the predecessor to FQAR, displaying 87.5% C_r's.

The equivalent form to Table B.1, is the following bar chart giving a clearer compression ratio comparisons picture for our chosen algorithms. As we can see, we could distinguish the FBAR and FQAR versions from others, as the most aligned and correlated versions of C_r's. By comparison, this makes the new algorithm more reliable in LDC results, consistence and thus its spatial efficiency factors on compression.

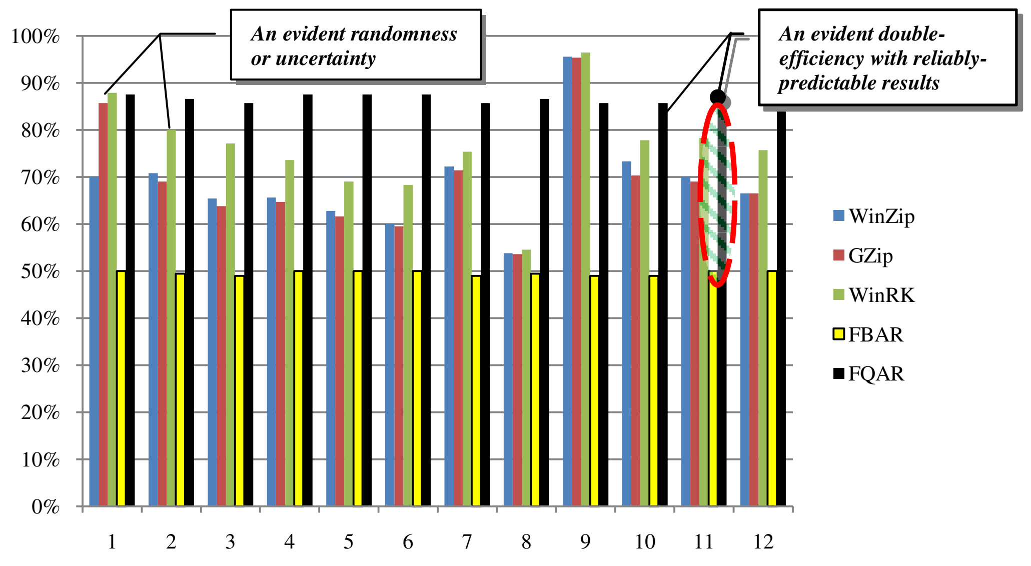


Fig. B.1: LDC ratio comparisons between FBAR/FQAR and other algorithms

The selection of an LDC package depends on the following criteria as applicable characteristics to all LDCs:

1. The ability to compress input data losslessly regardless of type, content size and complexity. If data type matters, e.g. being of textual type or otherwise, must compress textual information losslessly i.e. the decompressed data after compression must be identical to the original data.
2. Use memory for data access and management issues efficiently, e.g., data rate and spatial occupation of bits during compression i.e. when encoded, and referenced upon…
3. Must have a dictionary coder for validating data, referencing and dereferencing them during the reconstruction phase of data i.e. decompression.

As we can see, from Fig. B.2, based on the above characteristics, the selection of FBAR (fuzzy binary type) is oriented to FQAR (fuzzy quantum binary type) during implementation. Its simulation grade on x86 machines, reaches 87.5% LDC scenarios which are all fixed. The zone indicating x86 limits for the hybrid version (denoted as FABR~FQAR), inclusive of the ordinary FBAR versions, continues to expand within the purely-FQAR territory. This means, the structural integrity of the FBAR dictionary (or the 4D grid model) at $H = 0$ bits/byte final version on x86, is significantly changed in favor of FQAR before entering the quantum machine territory. In one word, FBAR mutates from version to version with uniformly-fixed values on space savings. In FQAR, negative entropy denoting universal predictably giving values ≥ 93.75% compression is estimated. This model could be considered as a solution to *complex negentropy problems* [43] in Signal Processing and Information Theory indeed.

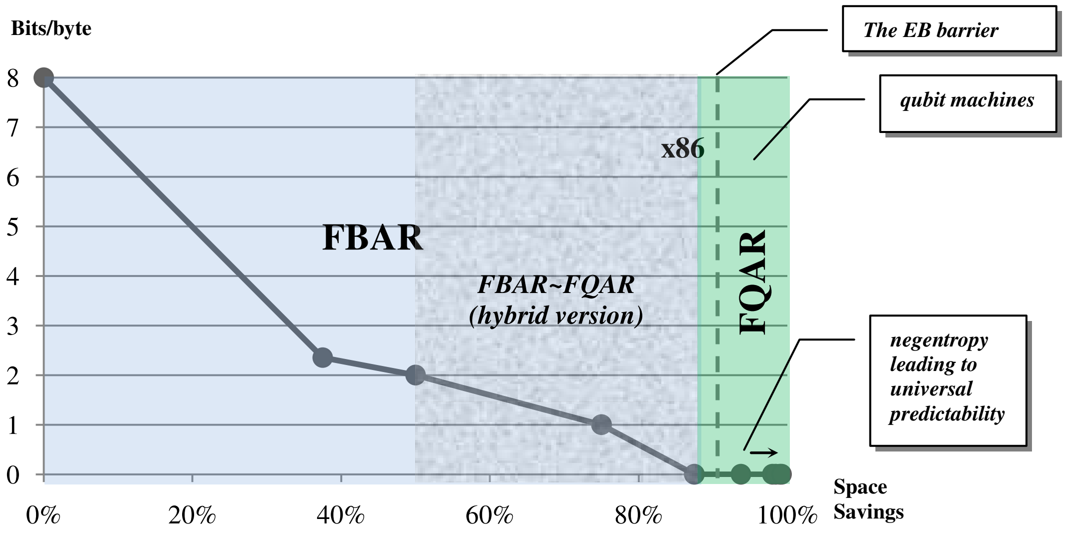


Fig. B.2: The pure FBAR mutating to a pure FQAR via its hybrid version on x86 machines

Twelve documents were given to four different LD compressors (in random order), relative to their bitrate performance for each LDC execution. Process time to a test and percentages of compression were measured. The resulted data are listed and discussed in the following section. For the EB barrier in Fig. B.2, refer to the explanations provided after Table A.7, which concern C's > 87.5% scenarios.

B.3.2 Nonparametric comparison test cases

Our motive using Freidman test: Ordinal data (countable data) gathered from repeated organization of LDC algorithms, i.e., repeated measures, employing a rating scale are commonly deployed in field and laboratory studies. If there are a large number of subjects (e.g., n>30), the assumptions of parametric approach, namely, normality and homogeneity of variance are usually met. Therefore, parametric analysis of variance (ANOVA) methods are frequently adopted to analyze these data. However, in field and laboratory trials conducted, situations frequently encounter in which small number of subjects, e.g., n < 15, are tested in repeated measure experiments. The more measures conducted relative to a growing number of samples, the more encountering of probable miscarriage of accuracy in the generated scores. In such cases, due to the relatively small sample size, the violation of assumptions of an ANOVA is usually inevitable. In FBAR, however, the sample size varied from, as small as 10,000 bytes, to much larger sizes ≈1.8 MBs, as char-based documents. In fact, the focus is on the results of how long the computation lasts per sample, its spatial consumption i.e. the percentage of compression relative to sample's rank. The main motive of using nonparametric comparison test cases is in contradicting the quality and quantity assessments done in t-test scenarios. The latter subsists on assumptions that form $T = Z/s$, where Z and s are functions of the data: Typically, Z is designed to be sensitive to the alternative hypothesis (i.e. its magnitude tends to be larger when the alternative hypothesis is true), whereas s is a scaling parameter that allows the distribution of T to be determined.

In the FBAR case, the assumption is severed from any sensitivity to alternative hypothesis or data distribution (e.g., *normality*). It is confined to the distribution of repeated observations on LDC I/O samples like many non-parametric tests, based on the ranks of the data rather than their raw values to calculate the statistic. In summary, we reason that this nonparametric choice was due to

a) The number of samples were < 20;
b) The data type was knows as char-based, hence the number of data types was limited (no extra assumptions like parametric methods were made.)
c) Not subject to normality measurements, unlike parametric and t-test cases.

In the following sections, we aim to use this method to evaluate our algorithm compared to other LDCs. Therefore, we wanted to make sure that its results were statistically significant and not obtained by chance. Thus, we considered the following null hypothesis:

> Let X contain our FBAR technique as well as a selection of state-of-the-art compression techniques. Furthermore, let Y contain a representative sample of documents of different type. Therefore,
>
> ***H.6-*** A difference exists in the performances of the techniques in X as measured on Y by computation rate and space savings.
> ***H.6₀-*** The difference in performances of the techniques in X as measured on Y by computation rate and space savings is zero.

In continue, since our number of samples is small and (n < 20), we use Freidman test to analyze the data and test the hypothesis (reference to Table A.10). The statistical test, involved the ranking of the data in the rows, then comparing the mean rank in each column. Thus the values of LDC would be ranked across each row as shown below. We derived these rankings collaboratively based on Fig. B.3, Tables B.1, B3-B4 results.

Document	#	WinZip	GZip	WinRK	FBAR	FQAR
text	1	4; **3**	3; **2**	1; **1**	5; **4**	2
book1	2	3; **2**	4; **3**	2; **1**	5; **4**	1
book2	3	3; **2**	4; **3**	2; **1**	5; **4**	1
paper1	4	3; **2**	4; **3**	2; **1**	5; **4**	1
paper2	5	3; **2**	4; **3**	2; **1**	5; **4**	1
paper3	6	3; **2**	4; **3**	2; **1**	5; **4**	1
web1	7	3; **2**	4; **3**	2; **1**	5; **4**	1
web2	8	3; **2**	4; **3**	2; **1**	5; **4**	1
log	9	2; **2**	3; **3**	1; **1**	5; **4**	4
cipher	10	3; **2**	4; **3**	2; **1**	5; **4**	1
latex1	11	3; **2**	4; **3**	2; **1**	5; **4**	1
latex2	12	4; **3**	3; **2**	2; **1**	5; **4**	1

Table B.2. Current test case LDC ranks on space savings

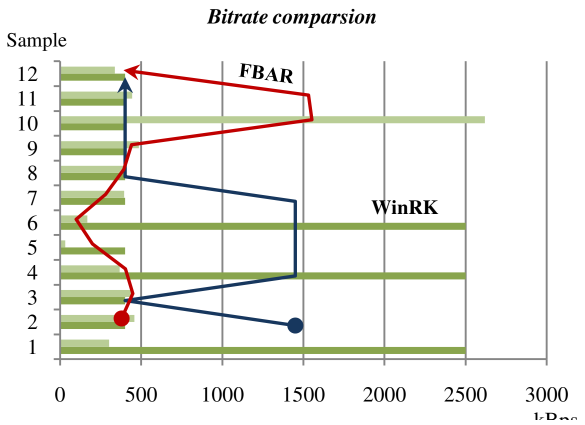
In Table B.2, we consider the rankings to be valid relative to the fuzzy quantum version (the FQAR column), while if dismissed, we consider the ranks to be distributed between 1-to-4 instead of 1-to-5. This is applied to observe the four first columns from the left relative to FBAR, in **bold** values. Now we start testing

Decision rule: Reject *H.6₀* if $F_r \geq$ critical value at $\alpha = 0.05$ or 0.01, corresponding to 5% or 1% probability *P*. Otherwise, stay consistent with null hypothesis *H.6₀*.

Calculation method: The differences between the sum of the ranks is evaluated by calculating the Friedman test statistic from the formula

$$F_{\mathrm{r}} = \left[\frac{12}{nk(k+1)} \sum_{i=1}^{k} R_i^2 \right] - 3n(k+1), \quad (9)$$

where k is the number of columns ('performance of algorithms'), n is the number of rows, and R_i is the sum of the ranks from columns. In compliance with our decision rule, the results on F_{r} which rejects *H.6₀*, are given in Table B.4, since *p*-value < α. The critical *p-value* of F_{r} for {4 observed columns + 1 hypothetical column} and 12 rows at $\alpha = 0.05$ or 0.01, is 0.0001. The distribution of the $F_{\mathrm{r}(4)}$ statistic is chi-square with k–1 degrees of freedom (df) or, df = 4. The *p*-value for the Freidman test is $P(F_{\mathrm{r(df)}} \geq F_{\mathrm{r\ observed}})$, the probability of observing a value at least as extreme as the test statistic for a chi-square distribution with df = 4. We thus conclude that the bitrate and space saving performances have had a significant result on the LDCs for the randomly loaded documents compared to FBAR. By conventional criteria, the *P*-value = 0.0001 < 0.01 rejects *H.6₀*, since this difference is considered to be extremely statistically significant. Although, dismissing the hypothetical column on FQAR results-in rank change on algorithms, we still observe $P = 0.0001 < 0.01$ rejecting *H.6₀*.



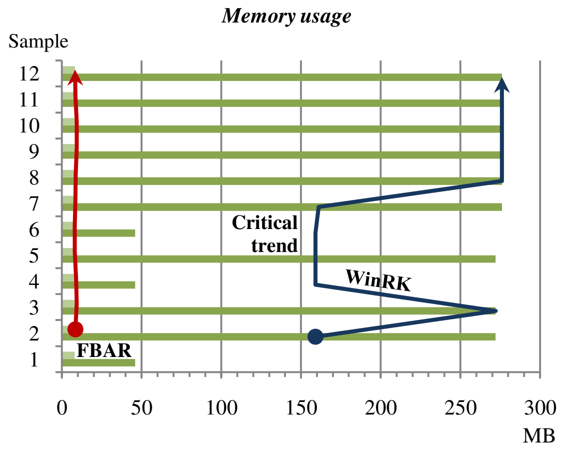


Fig. B.3. Bitrate comparisons and memory usage

Fig. B.3 shows the bitrate and memory performance on 12 test documents, with their critical and optimal trends. The results are elicited from Table B.3. The bitrate relative to memory usage was observed between the high and low ranked algorithms on 'space savings' (Table B.2): WinRK vs. FBAR. As we can see, for higher bitrate performances, WinRK has a critical usage of memory per input sample. In some cases, even having 10 kBps for encoding and decoding data, required 800 MB memory on a 2GHz Athlon CPU. This ranks WinRK's memory performance lower than expected, as 4th, compared to FBAR. When we associate values of the upper chart with the lower chart, it is evident that the empirical data relative to memory usage on FBAR is optimal, and uniformly correlated except, the jump of bitrate on sample # 10 called "cipher" (contents of some ciphered text). This is due to the excessive repetition of characters within the sample grid. The original input chars were ignored due to their pattern simplicity, forming simplistic forms of storable data. Therefore, the algorithm is not submissive for taking in too much information and thus its computation. The average base of bitrate was estimated 475 kBps for FBAR, and 925 kBps for WinRK on the 12 samples.

From the bar charts, it is possible to see that in some cases, there is already, right at the beginning, a major difference between the two results. There is also a difference observable at the end, where the mean coverage achieved by FBAR over memory usage is least critical than the mean coverage of the other compressor. This shows that there are significant differences between algorithms on their performances.

		t_L = CPU time/s				Compressed size (bytes)	bits/char
File	Size (bytes)	LLLC	HLLC	LDC	LDD	(**C**+**G** files) – 64 K LLLC vs. HLLC	LLLC vs. HLLC
text	61608	0.18	0.02	0.2	0.24	31124.8+**1**: 7701.00	[2.1,2.4] : [0,2]
book1	678244	1.31	0.14	1.45	1.40	342654.5 + **1**: 84780.50	[2.1,2.4] : [0,2]
book2	1772074	3.2	0.75	3.95	3.43	895266.5+ **1**: 221509.25	[2.1,2.4] : [0,2]
paper1	52516	0.13	0.01	0.14	0.20	26531.5+ **1**: 6564.50	[2.1,2.4] : [0,2]
paper2	117493	0.26	0.115	0.375	0.32	59358.4+ **1**: 14686.63	[2.1,2.4] : [0,2]
paper3	10262	0.05	0.01	0.06	0.10	5184.4+ **1**: 1282.75	[2.1,2.4] : [0,2]
web1	747766	1.63	0.22	1.85	1.71	377777.6+ **1**: 93470.75	[2.1,2.4] : [0,2]
web2	598125	1.29	0.2	1.49	1.36	302177.7+ **1**: 74765.63	[2.1,2.4] : [0,2]
log	1840924	3.43	0.27	3.7	3.58	930050.1+ **1**: 230115.50	[2.1,2.4] : [0,2]
cipher	777654	0.25	0.04	0.29	0.32	392877.28+ **1**: 97206.75	[2.1,2.4] : [0,2]
latex1	209212	0.43	0.03	0.46	0.49	105695.6+ **1**: 26151.50	[2.1,2.4] : [0,2]
latex2	155641	0.42	0.03	0.45	0.92	78631.1+ **1**: 19455.13	[2.1,2.4] : [0,2]
translator	≈ 8 MB	N/A	N/A	N/A	N/A	N/A	2 bits/char read
Total	7021519	12.58	1.835	14.415	14.07	3547342: 877689.88	Avg. 2.25:1

Table B.3: Estimates on compression with rate performance on FBAR's LDC and LDD

Friedman's test for repeated measures:

Document	WinZip	GZip	WinRK	FBAR	FQAR
sum of ranks	35	43	22	60	14
(sum of ranks)^2	1225	1849	484	3600	196
#of columns, k	$(4_{\text{real}} + 1_{\text{hypothetical}}) = 5$				
# of rows, n	12				
$\sum R^2$	*formulaic calculation* :	1225 + 1849 + 484 + 3600 + 196 = 7734			
$12/nk(k+1)$	*formulaic calculation* :	12/(12×5×6) = 0.033			
$3n(k+1)$	*formulaic calculation* :	3×12×6 = 216			
Test statistic F_r	*formulaic calculation* :	0.033 × 7734 – 216 = 39.22,		df = 4	P=0.0001 < 0.01
F_r without FQAR	*formulaic calculation* :	0.05 × 4280 – 180 = 34,		df = 3	P=0.0001 < 0.01

Table B.4: Rank sum and mean ranks via Freidman's test on hypothetical and real observed data with *P*-values

Relevant to the algorithmic computation process, the results in Tables B.3 and B.4 for an FBAR/FQAR LDC, were all **GC** file and dictionary coder-dependent. The grid file dimensions, each comprised of 16 fixed length code combinations, making 65,536 possible outcomes. From there, the translation table of the 95 printable and 1 nonprintable character block was used to make comparisons when the resultant document was converted back to binary for decompression. Table B.3, shows the difference between all layers being implemented from the lowest layer(s) of lossless compression (LLLC) to the highest (HLLC).

Interestingly, the lowest layers perform logic with expected bitrate. The LDC time parameter is the result of (LLLC + HLLC) time t_L, measured in seconds. On the other hand, having the highest layer with only '1 byte sequencer' equal to '1', according to the example given on pseudocode sample II, gives optimum performance. In other words, in total, **C** = '1' in content, makes the interpreter to interpret '11111111' for the whole document, otherwise, '00000000' on the first char input encounter. From there, applying self-embedded flags, altogether performs good bitrates by comparison. This is given by the additional byte (in **bold**) added to the HLLC column of the table.

We determine the limits of the application to be mostly on hardware constraints in design, rather than FBAR logic per se. To tackle this, we eliminated issues related to single bit usage of flags, considering their unique combination in **G** file is indeed avoiding 'bit array' models in programming. In fact, hard-coding 65,536 grid units via 'if loop statements', reading line-by-line with 95 printable char replacements, is more useful than the currently-available tools utilized for an x86 compiler. This enabled us to have all flags embedded in our marked-grid units by a standard char.

After verifying the theoretical estimates of 37.5%, 50%, 75% and 87.5% fixed size compressions, we began to compute the bitrate factor of our FBAR LDC. The result on randomly chosen documents for performing an LDD is listed in Table B.3. The bitrate for both LDC and LDD relative to CPU time/s are computed and listed in the same table. We then included specific test results in form of Freidman's mean ranks and rank sum in recognition of hypothesis *H.6* of this paper.

According to the sequencer approach mentioned above, it takes 5 to 6 levels of conversions with a CPU time t_L = {long + short + shorter + shortest} session to conduct all four FBAR LDC layers. Therefore, the HLLC version would practically engross one layer involvement during data computation. Hence, the logical results would give t_L on HLLC $\ll$ LLLC. This occurs relative to accessing the 'translation table' on 4×1-bit flags identity on each **G** row for an LDD. Typically giving t_L on LDD > LLLC intervals, due to data access, read and write operations during data reconstruction from the **GC** file and translation table.

B.3.2.1 FBAR and other lossless data compressors

The number of documents builds up the test case and comparisons attribute of our nonparametric test. On the one hand, the number of arbitrary documents relative to packages is proportional to the random increase and decrease of percentages of LDCs, spatially. Moreover, the temporal state on the computation and processing of bits with their dictionary codes to decode, remains consistent with the dimensional expansion on spatial occupation of bits in the memory matrix, proportional to the accumulation factor of the bits number. If this factor increases exponentially, the temporal state prolongs i.e., latency in compressing data, otherwise, efficient bitrate computation is recognized in the LDC's data analysis records. Any LDC must appreciate this behavior regardless of quantity and complexity with respect to RAM usage, CPU and LDC package switches returning values with reference to dictionary. FBAR, in addition to all of these expectations, must maintain logic prior to the leading closures of 0 and 1 states, attaining levels of efficiency within the LDC processing.

B.3.2.2 Evaluation of packages or LDC algorithms

The evaluation of packages strictly depend on the LDC selection criteria outlined in § B.3.1. Once, each algorithm is evaluated for all characteristics, we rank it according to its space savings, memory usage and bitrate performances.

By referring to relevant sources giving details on LDC packages [22], one could outline the basis of the statistical test for results. Table B.1 contains these packages with their respective ranks, reflected in Table B.2. We run our statistical test for comparing three or more related LDCs, as a result of their space savings, which makes no assumptions about the underlying distribution of the C_r data. The data is set out in a table comprising *n* rows by *k* columns. The data is then ranked across the rows and the mean rank for each column is compared. Bitrate ranking is statistically compared between the highest and lowest ranked algorithms, further constituting our null hypothesis. The comparisons data is given in Fig. 9.

Based on the characteristics from § B.3.1, the ranking of the package is given through percentages of C_r for each package. The C_r is not fixed for each package and merely based on probability and character letter counts or frequent reoccurrence for the Shannon entropy, used to conduct a lossless compression. The only package that deviates from this behavior is FBAR, which exhibits predictable C_r ratios regardless of content size and complexity. The selected packages were on the bases of best case probable scenarios in compressing data above 90% as a maximum LDC, 50% as an intermediate LDC, and below 50% > 0, as a classic LDC (the ordinary well-known LDC techniques embedded in such packages). Contradictorily, for the fixed C_r generated by the FBAR package, is conveniently more reliable in predicting C_r ratios compared to the probabilistic C_r's by WinZip, GZip, WinRK and LZW LDC packages. The ranking is further evaluated when package evidence of random sample inputs are measured non-parametrically.

B.3.3 Evaluation of packages or LDC algorithms

The following subsections, shall address issues related to Software Engineering, which aims at the performance-related issues, risks, confidence, usability, etc., based upon the resulting products of the FBAR algorithm:

Introductorily, the potentials of the algorithmic/package evaluation, lie-in the way the compressor compresses data in form of fixed sizes and predictable ranges of compression output. This makes it more reliable to compress all sorts of document size, regardless of content type. The testability for performing such characteristics remains iteratively correct under different testing criteria, or, applications. An FBAR evaluation of a data compression, has already been motivated throughout the previous sections, and therefore, comparison tests were made with other compression algorithms. Now we shall discuss the evaluation thematic results of the FBAR algorithm as follows.

B.3.3.2.1 Usability:

We discuss the uniqueness of the FBAR data compressor compared to other remaining compressors used today. We also point out the potentials for demanding this product as follows:

The potentials lie in the way the compressor compresses data in form of fixed sizes and predictable ranges of compression output. Unlike other LDCs, a fixed table forming a fixed size dictionary is always in store for the FBAR compressor, others, however, build a different one during data type conversions, every one-time execution. This makes FBAR more reliable to compress all sorts of document sizes regardless of content type. The testability for performing such characteristics remains iteratively correct under different testing criteria or applications. An FBAR evaluation of data compression has already been motivated throughout the previous sections, and comparison tests were made with other compression algorithms.

In addition, based on our performance comparisons, we deduce that *overhead information* in FBAR is unlikely to happen due to dependency of the program-read on the translation table as a static portable object between different driver/network locations (see also, robustness in § B.3.3.2.4). Furthermore, FBAR would not create overly occupied tasks in queues and overwhelm memory against user's will. This is evident according to our performance comparisons made in e.g., Fig. B.3. Therefore, multiple API's or *k*-thread executions for process management engaging the user, is none of our concern during FBAR I/O operations. Reasoning that, the FBAR functionality plays a key role in demarcating the usability aspects of the algorithm from its interface core to surface, for each one-time application run.

B.3.3.2.2 Functionality:

The development of new data compression software is a competitive activity and the time available to bring a product to market is often limited. Furthermore, the complexity and size of software systems have increased in recent years especially when it comes to lossless data compressions. There are diverse techniques to perform compression based on mere probabilities. In principle, they benefit from e.g., Shannon entropy [15, 16, 18] to compute similarities in data objects and their recursive pattern recognitions. More specifically, they base their compression on repeated patterns of input data to bit sequences (*frequently encountered*), [32]; hence their frequency varies in their compressed version (an *uncertainty*). In order not to fail on the market, it is important to also achieve a high quality with intact data integrity when studying the output data. The usability of FBAR is due to its logic as almost being independent of an uncertainty, thus its potential demandability on the market side increases in number of its users in the future. The main reason is that, satisfactory compressions with predictable compression ratios, gives the user (customer) more assurance in compressing his/her data right on the spot without being concerned about his/her machine's spatial management issues. For example, how much RAM is required; how much space is needed on this HDD for this particular compressed version of the original file; will there be enough space after compression on this driver after compression, or so to speak, will we be free to store more records on the driver, etc. Having fixed values, gives a definite answer on such spatial limitations to its user.

B.3.3.2.3 Reliability:

Mostly, on complex systems, FBAR generates fixed value identities, compression results and reference points. Hence, the FBAR design for x86 machines, performs with the purpose of reliability testing based on its firm logic with confidence to produce fixed values with finite number of lines of code in structure. In aim of discovering potential problems within the design, as early as possible and, ultimately, provide confidence that the system meets its reliability requirements, the less confidence is not an issue to bargain with, in FBAR. The rationale to this is, since handling all states of logic inclusive of fuzzy is quite complex in current applications, and when reaching levels of quantum states parallel to binary logic is virtually continuous, the weak point in confidence would therefore abort to exist. In fact, the latter factor becomes unsustainable due to the sustainability of the confidence itself within the representations of impure and pure logic, keeping product resultants intact with one another, in data structure, efficiency and logical consequence in design. The limitations in the circuitry design of the hardware system and its corresponding components, of course have an influence (affect) on performance and confidence factors from the lowest LDCs (37.5% and 50%) to the highest LDCs (87.5%) on x86 machines. The FBAR components, in context, are a set of conversion functions as useful tools that mainly rely on a comparator component (see Fig. A.3), which itself is an 'interpreting function' possessing a series of 'if and else' statements to compare data from one another. Its job is to compare the logical combinations and their consequences for decompressing data, losslessly. These functions are incorporated in the subprograms of the algorithm, and their layout is already given in Fig. A.3, Appdx. A. In summary, we rate FBAR algorithmic product confidence, as 'quite high' in our evaluation method. The reasons behind this evaluation are:

A. Because FBAR values are predictable, and the confidence is rated based on the predictability of spatial and temporal rates, displaying reliably-fixed bitrate results per lossless data compression (recall, Fig. B.3). In other words, this confidence is proportional to the growth of predictability. The more predictable, the grater the confidence or,

$$FBAR\ confidence \propto FBAR\ predictability \times 2n,\ \ n \in \mathbb{N} \tag{10}$$

   where $2n$ comes from the double-efficiency property of the algorithm, obeying the natural numbers' interval result set, from Eq. (7), for each predictable state of compression, according to Fig. B.2. Therefore, in time, confidence grows double-efficiently for each progressive version of FABAR.
B. Thus, FBAR is least likely to fail at all, in logic, design and principle.
C. We have done this with the new 4D bit flag model, and its algorithmic representation (pseudocodes).
D. Why? Well, FBAR is here to perform maximal and thus ultimate LDCs, prior to the LDC algorithms we know today.

The mean time to failure (MTTF), and other factors, such as, confidence interval (CI) estimates, which we have not conducted in the FBAR project, are deemed important when our algorithm is tested in the large, and not in the small, or, under its prototypic release (§ B.2.1). However, the reliability analysis by running surveys, when the package is tested in the future, gives concrete results to compute such CI estimates, relevant to algorithmic space savings' products. The only potential problem that we could point out from is, handling the logic package between x86 machines and those of which are strongly dependent on quantum encryption methods in information theory, e.g., 'superdense coding' [34], in their instructions set and algorithmic responses. The latter, however, is available under laboratorial conditions aiming for future technological developments done by organizations such as IBM, like the *qcl quantum computer simulator* and similar algorithms testing quantum computation at large [41].

B.3.3.2.4 Robustness:

In cryptography, according to Kerckhoffs' principle (assumption, axiom or law) [48], a *cryptosystem* should be secure even if everything about the system, except the key, is public knowledge. Throughout this thesis, we have introduced and discussed FBAR, both on its encryption and decryption properties for an arbitrary data I/O. FABR has its own cryptographic translations as specified in the paper section, in terms of a translation table inclusive of [2] as a reference table employed for the lowest layer of dynamic data conversions. The *public key* as either the translation table for the static approach, or the reference table for the dynamic approach, abides by the encryption design principles inclusive of a mirror technique for fundamental exclusions of data corruption. This is how we apply this new mirror technique to FBAR during e.g., peer-to-peer transmissions in Fig. B.4:

Foremost, since we know what specific occupant chars (e.g., Table A.6), are occupying the compressed file contents, it is evident with a *fuzz testing* e.g., giving invalid and valid I/O, the program could thus detect and exclude error chars from the compressed version, allowing it to return the original chars at the decompression phase. It is further evident, that even some of the error chars, if they fall in the range of our 95 standardized occupant char range, the zone that gives an additional size expectancy of the file, would thus indicate an error has been occurred. Reason that: based on FBAR I/O conversions, we totally expect the compressed version (the **GC** file) if done through 50% compression, must only carry half of the original file size excluding the 64KB size of the grid file. If this varies significantly, the program on the other side (point B), would know that there has been an error occurrence during transmission. To mimic these kinds of scenarios, we purposely input random characters to the compressed version offline as part of our experiment. We conduct our robustness in terms of manipulating data as the corrupt version of **GC** whilst creating a mirror file, which contains characters that are not in the scale of our standard occupant-chars of the **GC** file. The mirror file is always 0 bytes in content, unless error occurs during transmission, and thus in complement with the **GC** file, increases in size. A self-generating *recheck char loop* algorithm (in structure, inherits FBAR's comparator subroutine statements), for *error detection during transmission* (a real-time patch), which rechecks the **GC** file periodically is anticipated along the mirror file generation. Bear in mind, mirror file is only generated, once an error occurs, thus detected by the real-time transmission loop algorithm. Further, we convert the corrupted zone of the **GC** file to certain out of occupant char range i.e. 95 printable chars, (or recall Table A.6) and store it in the mirror file. The mirror file could be deemed as a complementary object to the **GC** file, reflecting error chars outside the *pristine boundary of data* (the zone where data must not be of corrupted type), where at the end, gets deleted once all corrupt data are detected, extracted and compared with the FBAR dictionary (reference table, translation table and ASCII). If there are any remaining errors i.e. in case of the mirror file is received with a size > 0 bytes, a further error detection is executed when an LDD is run by user # 2. This detection is done via program's interpreter/comparator. Then, at the point of LDD delivery, we expect the decompressed version, of course after the use of the translation table, to return the expected original file identically without corrupted chars in it i.e. the error version. Ergo, we say that FBAR, due to its uniqueness in char deliveries and translation table comparisons technique, makes it quite robust compared to the probabilistic versions where many patches are designed for particular set of errors, making user to depend on many other factors when such issues are confronted on his/her machine.

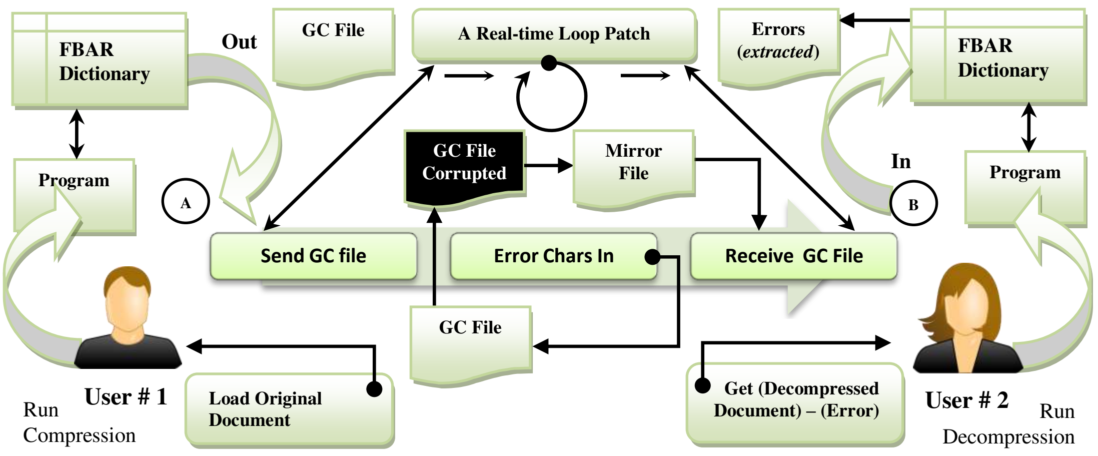


Fig. B.4. An example to examine the level of FBAR robustness between transmission points

The algorithm, during its transmission protocol on e.g., a network, from point A to point B might be tampered with its compressed result or LDC product. One could, based on different observations or experience, class this as overhead information within the communications system.

Fig. B.4, above, shows how robust the algorithm performs under such circumstances when an FBAR sender, sends the compressed data to the recipient, while a network error or some intrusion casing corrupts user's data. Of course, a physical error like connection problems, etc., could be tackled with certain additional software programs (patches) that maintain data transmission between lines until connection is fixed to resume the delayed data transmission.

Regarding the integrity being altered in the compression content, it is evident by creating a mirror of the compressed file along the way (previous paragraphs), the encrypted message i.e. the contents of the **GC** file in terms of occupant chars (Table A.6), are strictly dependent on the contents of the translation table on both sides, sender A, and receiver B. Thus, whatever returned at the decompression phase, if the initial file before compression in any shape, with error or not, is to be returned as it is. Meaning that, the program's interpreter/comparator installed at point B, works on returning the original chars and exclude chars of error type, coming from either file content zone, within the transmission layers, or, right after source before destination (point x in A-B).

Of course, further technical discussions relating to this extension of the algorithm is out of the current thesis scope, and requires a separate work in progress parallel to the current topic.

B.3.3.2.5 Efficiency:

Table B.3 efficiency ranks on memory usage, is the result of expressing the ranking of FBAR or even FQAR as quite consistent with uniform efficiency factors compared to other LDCs. Despite of FBAR suffering on space savings, but since it is performing uniformly, even on the lowest standards, the structure of logical consequence mutates from one version to another (Fig. B.2). Ergo, the performance is also mutable for every upgraded version from FBAR to FQAR, either on bitrate, space saving or memory usage factors. Therefore, we deduce that the efficiency of FBAR is of a reliable type on any versions of it, since the structure is strongly grid content-dependent, and the reconstruction of data, stands firm on its combinatorial logic (impure and pure states of binary).

Document	#	WinRK	FBAR
text	1	2	1
book1	2	1	2
book2	3	2	1
paper1	4	2	1
paper2	5	2	1
paper3	6	2	1
web1	7	2	1
web2	8	2	1
log	9	2	1
cipher	10	2	1
latex1	11	2	1
latex2	12	2	1

Table B.3. Current test cases between FBAR and WinRK LDC ranks on memory usage

B.3.3.2.6 Completeness relative to efficiency:

For current x86 machines, the FBAR model design suffices to generate 37.5% to 50%, and potentially, 87.5%, if indeed grid file-dependent with a comparator. The 50% case, proving an absolute superdense coding technique [34] in Coding Theory and Cryptography [3], is also complete with confidence, since the grid file could be recalibrated to exponentially act on behalf of a compressed file. This file contains the position number of the representative char entry akin to an actual compressed file generated by the program. Furthermore, contrary to the attempts being made in quantum information theory for compressing 2bits via 1 qubit during data transmissions, the 50% reduces probable quantum states as self-contained in the grid file with no state probability (0 state probabilities). Ergo, the confidence rate for *doubling the efficiency* (superdense) in reconstructing chars is absolute and self-explanatory. Extending the grid file addresses in format, of course, as described in §A.1.2, Appdx. A, results in 87.5% compression. Once the logic constructor grid acknowledges the results by the comparator relative to the compressed file, such fixed size estimates are realized during decompression. Furthermore, the application could be executed under Windows and UNIX platforms despite of memory efficiency factor or rate differences and hardware constraint problems handling FBAR logic, i.e. application's data computation performance on spatial and temporal localities.

B.3.3.2.7 Portability:

The FBAR logic could be implemented under different platforms, even programming languages. Generally, the application could be implemented in C, .Net or any equivalent programming language using the right functions to perform FBAR logic. However, there are a number of customized functions that are required for implementation as specified within the context of FBAR logic, in case of not benefiting from a programming language standardized functions per release. Furthermore, the application could be executed under Windows and UNIX platforms despite of memory efficiency or rate differences e.g., application's data computation performance on spatial and temporal factors. It is also important to connote that the algorithm contains portable components such as **GC** and dictionary files, making it portable from one compiler to another. The execution of commands and translation, from one computation level to another, is standardized and manageable on different machines.

Also, the cumulative results in terms of a compressed file in context are in accordance with an expected size performed by the algorithm. It is based on the firm self-embedded flags, packed into 4-bits per reconstructible char. This shows us that all results are at least, half of the original input size of some random document. Ergo, portability in form of compressed files, such as **GC** type, with its decompression package, simply, by choosing the relevant option, on the user's side, reconstructs data expectably. Portability of the algorithm, its components and executables, e.g., "file-faces upon new and old data during LDC and LDD phases", within this context, is evident, and thus easy to envisage from a developer's side delivered to the customer-client side, on nowadays machines.

B.3.3.2.8 Validity threats:

Due to the evident nature of Shannon entropy, practiced within the four packages of our choice, and thence justified, "in the choice itself," based on their ability of compression according to their global rankings [22], and logic, e.g., [18], we thus deem all-LDC package selections to be indeed universal. Hence, the ASCII representation for all data conversions is too universal including other LDCs' *cause* and *effect* of the internal systems, as an *internal validity threat* due to extreme variance or randomness of logic, i.e., their implementation of logic. Therefore, the selection comes about based on these evident logical conversions, whilst we conduct our version of logic, FBAR, for its provability aspect in being different, i.e., not being based on repeated patterns of symbols, makes this threat to be evidently reduced due to internal functions, operation of logic. So the speak, due to FBAR design being merely based on pure circuitry logic in a combinatorial manner i.e., Fuzzy Binary AND/OR with certain universal operators, as newly-defined in terms of **znip** operators, makes this algorithm in pre-test conditions to confine threats in its context of universality relative to post-test conditions where generality of the application is conducted on e.g., computers, networks, etc. (values are, herein, self-embedded or, self-contained). We could even see this for higher degrees of FBAR experimentations within the *external validity* context, which engulfs threats relative to users using FBAR. For example, we map such participation of users to real-world relationships, such as discussing the robustness of the algorithm in § B.3.3.2.4. Ergo, the selections, based on the available data from [22], inclusive of our own extensive analyses on the encoding/decoding layers, before the algorithm's universal translation table, are all 'universal' indeed. This made the selection imminent prior to logic during sampling of the char-based type documents, which further maps to our version in terms of data-type, and NOT THE LOGIC per se.

Since our logic is mathematically self-contained (impregnable) for single and groups of bit values, each potential threat is thus self-contained. Therefore, we admit by contrast that, the remaining LDCs are practicing some '*discrete logic*' without a combinatorial extension to it, with loosen threats spread all over the e.g., network. This makes developers frustrated in designing package extensions, and thereby multiple applications to tackle each problem occurring for each probabilistic LDC, compared to FBAR. On the one hand, FBAR does not claim to possess an *independent variable*, yet all validity threats remain intact with it. On the other hand, in FBAR, since the logic itself results in *one universal variable*, it self-contains all *dependent variables,* regardless of complexity and uncertainty factors within its qualitative and quantitative relationships. This means that, the former situation is not in effect (self-contained) for all transmissions, majoring the predictability aspect with respect to participants and groups of LDCs.

B.3.3.2.9 Risks in summary:

FBAR only fails if the program functions and their subroutines are not implemented according to the 4D-read/write bit-flag model. In other words, debugging and validation issues, is always the case during implementation. Moreover, the EB barrier, illustrated in Fig. B.2, and explicated earlier in the ending parts of § A.1.2.3, handled by the 64-bit microprocessor for $C_r > 87.5\%$, (hypothesis *H.5*), must abide by the rules of complex matrix computation (*the comparator matrix*) for the 4-distributed tables, in reality. Meaning that, the very first translation table representing the double-efficiency of 50% compression, is the ultimate solution of the intersection of the values with those values associated in its identical tables from the program code (LDC/LDD subroutines). This has been illustrated in the later parts of § A.1.2.3. Ergo, no matter how variant the LDC result in other compressors, FBAR remains relatively fixed, due to its universal translation table being intact with its package supporting double-efficiency for each release. The translation table respectively is 1, 2, and 4 in quantity, supporting the simplistic to complex orders of Eq. (7) for the hybrid and non-hybrid versions of FBAR/FQAR.

Appendix C

Appendix C concludes the thesis with the discussion (Section C.1) of the key findings with respect to the central research question, highlights the strengths and limitation of the study, and presents some possible direction of future work following its expected applications (Sections C.2 and C.3).

C.1 Discussion

Data compression is all around us. We see it in a variety of products, such as, high definition televisions, DVDs, MP3 players, cell phones, digital cameras, fax machines, automobiles, etc. When we look at all the embedded products out there, we will quickly see that data compression is an integral part of their operation [40]. In this study, we presented FBAR logic, prototype, performance, usability and its applications. The FBAR was tested in an experiment in which the outcome was compared with the results of other LDC algorithms.

In this study, the prototype was built to perform a lossless data compression, and necessary comparisons were made with other lossless data compressors. The results respectively concluded that the FBAR compression regardless of file content, type and size, generated fixed size compression ratios with double-efficiency, a factor in which other compressors are incapable of performing in practice. This precludes the fact that other compressors rely on certain fixed ratios for their space savings on a computer system. They embark on uncertainty over fixed file sizes. When file content or type changes, the data compression ratio also changes with respect to the compression technique incorporated, mainly performing probability pattern checks of recurring characters within that file. So, FBAR logic poses to call-in the 'logic' itself rather than taking in recurrences of the same character to compress data in a limited space and timeframe.

The compressible space however, could be unequally partitioned in a discrete manner which itself generates an uncertain loci of compressed bits i.e. where and what size are there in the compressed pattern. One case though, rarely plausible to occur for a 99% compression, and that is, assuming all file content is built-up of one recurring character, e.g., *aaaaaaaaaa* …. *a*, resulting in $k[a] = 3 + x$ characters in the compressed version; where k is the size in bytes, each byte represents 1 reoccurring character, and x depends on how k is presented for the new length e.g., if $k = 10$ then $x = 2$, if $k = 1000$, $x = 4$ bytes and …, spatially occupied. On the other hand, FBAR by contrast, makes all data compressed, irrespective of content and type, based on a single combinatorial logic. This is the strength of FBAR, its universal applicability to any information object, regardless of size, type and structure.

The string values in FBAR contrary to the string exemplified in the previous paragraph, were treated as binary during the encoding and lossless compression procedures. The strings were compressed into equivalent characters from the ASCII table into file **C**, thereby to a 4D grid file **G**. From there, the translation table of the 95 printable and 1 nonprintable character block was used to make comparisons when the resultant document was converted back to binary for decompression. We evaluated the difference between all layers that were being implemented from the lowest layer(s) of lossless compression (LLLC) to the highest (HLLC).

We determine the limits of the application to be mostly on hardware constraints in design, rather than FBAR logic per se. To tackle this, we eliminated issues related to single bit usage of flags, considering their unique combination in **G** file, is indeed avoiding 'bit array' models in programming.

After verifying the theoretical estimates of 37.5%, 50%, 75% and 87.5% fixed size compressions, we began to compute the bitrate factor of our FBAR LDC.

The bitrate for both LDC and LDD relative to CPU time/s were computed and listed in Table B.3. We then included specific test results in form of Freidman's mean ranks and rank sum in recognition of hypothesis *H.6* of this paper.

We point out a possible threat to external validity which could be the selection of test samples, since there are LDC packages under development and those that are yet unknown to general public, not labeled as the most popular ones like WinRK, GZip, etc. This makes the selection of packages less absolute in algorithmic evaluations and comparisons made upon their performance.

This also entailed other risks related to performance factors such as memory usage, discussed earlier as resolutely intact for any versions of FBAR compared to other algorithms. For example, not all embedded systems support large blocks of RAM memory that can be allocated to a striving compression algorithm as well as WinRK for giving high C_r's yet requires lots of memory space to encode/decode data. For embedded designs with limited RAM memory, the challenge is to find software that can achieve acceptable efficiencies within a small memory footprint. We propose FBAR operates in a more reasonable boundary of memory usage compared to other LDCs.

This thesis described a lossless compression algorithm based on the popular LZW compression standard and its opponents such as FBAR/FQAR. Broadly, the well-known LDCs suffer in consistency in acting less probabilistic, thus depending on many random factors and redundancies when it comes to data types, finding a char match within the context of LDC.

Concerning memory size management and thus its efficiency, there are ways to reconfigure data access and allocation through compression techniques as explained in the previous chapters, e.g. §A.1.2, through a dynamic versus static approach.

The current version of FBAR compresses data up to 87.5% in a unique manner, whereas other compressors do not, since they generate random values depending on data content. FBAR is independent of data content, since it follows one and only one concrete rule of logic, applicable to data content itself. This logic is combinatorial, and defined as fuzzy with binary *and-or* logic promotive into quantum logic and vice versa. That is why this attribute of FBAR ranks it as the highly-information conversive to all logical standards discovered today. In other words, a new way of compressing data losslessly amongst other compressors, is itself a novel approach. The most efficient FBAR compression would be to implement its quantum application with fuzzy, inclusive of binary for performing robust calculations between the absolute discrete state boundaries (purely binary), and dual state boundaries (quantum like) applications. This would allow decision making systems to compute data efficiently, with all of its complexity intact within the information object.

FBAR future applications lie onto implementing its quantum logic through signal processing and physical applications i.e. a new hardware design and organization, briefly explained as follows:

C.2 The future of FBAR's ultimate compressions

Simulating a quantum computer (QC) on a traditional classical computer, e.g., an x86 machine is a hard problem. The resources required increase exponentially with the amount of quantum memory under simulation, to the point at which simulating a QC with even a few dozen quantum bits (*qubits*) is well beyond the capability of any computer made today [41]. Maneuvering with FBAR combinatorial logic, reversing ANDed and ORed results inclusive of their in-detailed threshold states of fuzzy type, allows future interpreters and dictionary coders, act most efficient in shortest signals possible. Despite of hypothesizing now on the primitive grounds of this compressor on current computers, its future would promulgate possible advances in quantum technology to handle extreme complexities of logic compared to nowadays LDCs. The reason that LDCs today, consume space, and thus clumsy in delivering data in shortest time possible compared to lossy types, which is apparent in their management of converting multidimensional limits of space in the shortest forms possible.

Strong dependency on probabilities rather than entangling them in terms of 1-with-0 and 0-with-1, rather than treating them totally discrete in their representation, demands to create more reference points and uncertainties in data reconstruction and calculations. Ergo, having an FBAR model in possession, makes computation from the most discrete states to the most continuous, and in overall, connected time frames retrievable from one information content to another

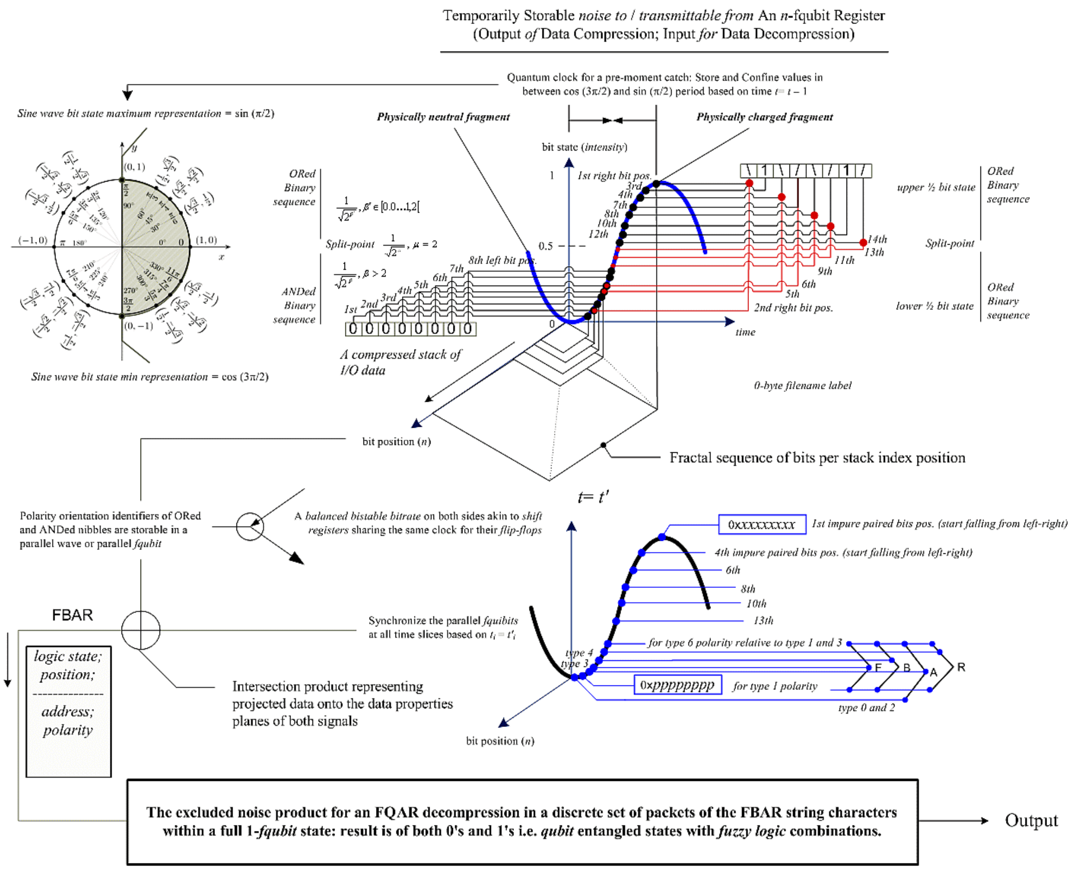

Temporarily Storable *noise to* / *transmittable from* An *n*-fqubit Register
(Output *of* Data Compression; Input *for* Data Decompression)
Quantum clock for a pre-moment catch: Store and Confine values in between cos (3π/2) and sin (π/2) period based on time t = t − 1
Sine wave bit state maximum representation = sin (π/2)
Sine wave bit state min representation = cos (3π/2)
Physically neutral fragment
Physically charged fragment
bit state (intensity)
1st right bit pos.
8th left bit pos.
2nd right bit pos.
ORed Binary sequence
Split-point
ANDed Binary sequence
upper ½ bit state
lower ½ bit state
time
A compressed stack of I/O data
0-byte filename label
bit position (n)
Fractal sequence of bits per stack index position
Polarity orientation identifiers of ORed and ANDed nibbles are storable in a parallel wave or parallel fqubit
A balanced bistable bitrate on both sides akin to shift registers sharing the same clock for their flip-flops
t = t'
0xxxxxxxxx
1st impure paired bits pos. (start falling from left-right)
4th impure paired bits pos. (start falling from left-right)
FBAR
logic state; position;
address; polarity
Synchronize the parallel fquibits at all time slices based on t_i = t'_i
Intersection product representing projected data onto the data properties planes of both signals
for type 6 polarity relative to type 1 and 3
0xpppppppp
for type 1 polarity
type 0 and 2
F B A R
The excluded noise product for an FQAR decompression in a discrete set of packets of the FBAR string characters within a full 1-fqubit state: result is of both 0's and 1's i.e. qubit entangled states with fuzzy logic combinations.
Output

Fig. C.1. An overview of two co-local parallel signals, in total forming 1-bit revolution whilst a set of *equipartitioned fragments* of signal occupied by binary bits of data. (Extracted from [2])

The transformation of FBAR to its highest levels of compression within its four-layer encodings is done via qubit registers. In this regard, a *seclusive proposal* in § 3.5.1 [2], is given for their new hardware design principles. The design *in theory*, with its practical aspects of an *n*-fqubit register, is briefly outlined as follows:

C.2.1. New memory architecture for future FBAR LDCs with sub-bit handling

As we know by now, a number of *entangled qubits* taken together is a qubit register. Quantum computers perform calculations by manipulating qubits within a register. A *qubyte* is a collection of eight entangled qubits. It was first demonstrated by a team at the Institute of Quantum Optics and Quantum Information at the University of Innsbruck in Austria, in December 2005.

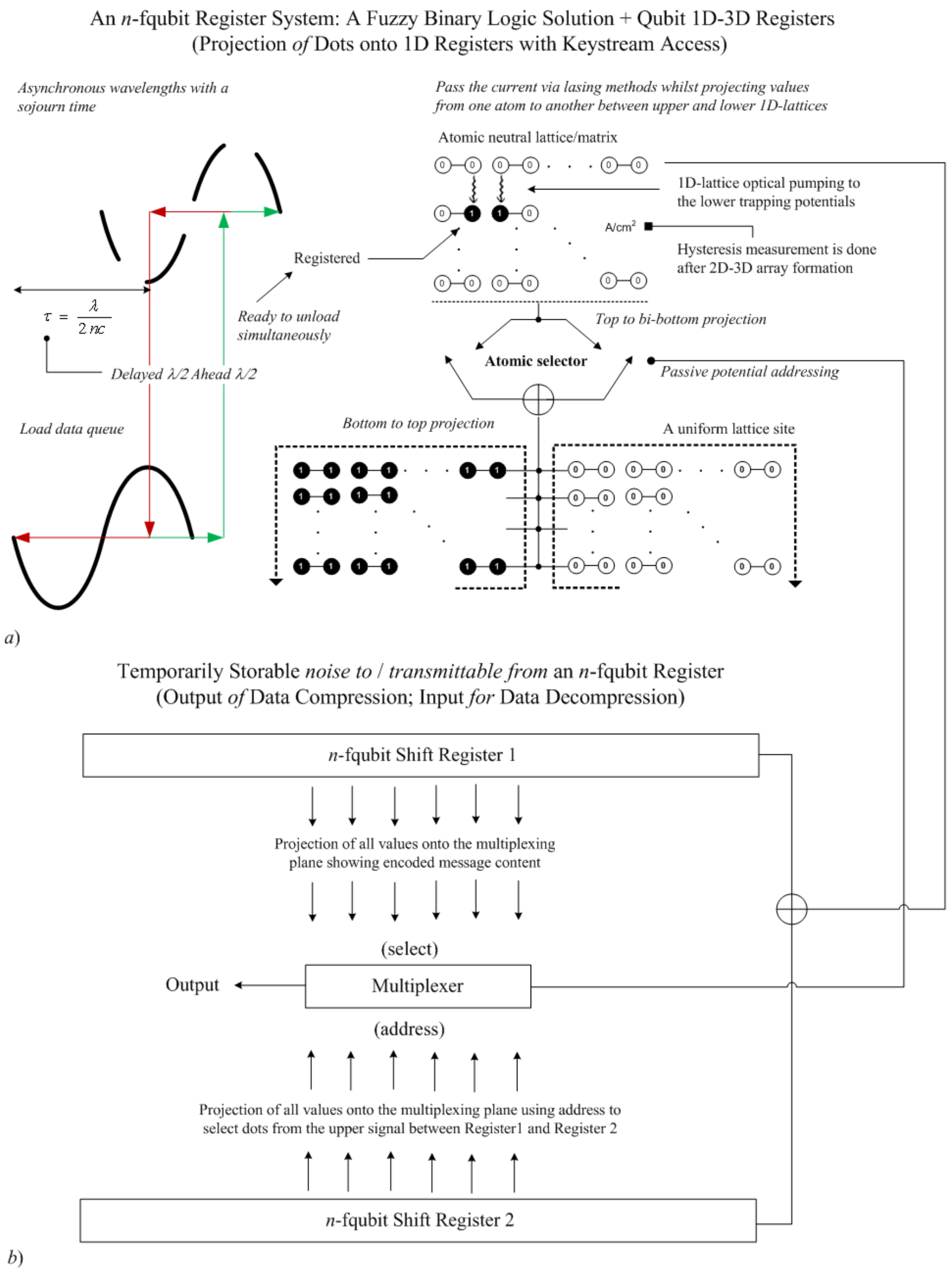

An n-fqubit Register System: A Fuzzy Binary Logic Solution + Qubit 1D-3D Registers
(Projection of Dots onto 1D Registers with Keystream Access)
Asynchronous wavelengths with a sojourn time
Pass the current via lasing methods whilst projecting values from one atom to another between upper and lower 1D-lattices
Atomic neutral lattice/matrix
1D-lattice optical pumping to the lower trapping potentials
A/cm²
Hysteresis measurement is done after 2D-3D array formation
Registered
Ready to unload simultaneously
τ = λ / 2nc
Delayed λ/2 Ahead λ/2
Top to bi-bottom projection
Atomic selector
Passive potential addressing
Load data queue
Bottom to top projection
A uniform lattice site
a)
Temporarily Storable noise to / transmittable from an n-fqubit Register
(Output of Data Compression; Input for Data Decompression)
n-fqubit Shift Register 1
Projection of all values onto the multiplexing plane showing encoded message content
(select)
Output
Multiplexer
(address)
Projection of all values onto the multiplexing plane using address to select dots from the upper signal between Register1 and Register 2
n-fqubit Shift Register 2
b)

Fig. C.2. (*a*) An *n*-fqubit register system with *overcoming latency options* between clocked quantized signals; (*b*) A 2*n*-fqubit multiplexer generator taking in signals from the upper image for data decompression purposes. (Extracted from [2])

By referring to our new 4D grid model from Appdx. A, a superposing character representing at least two characters, indicating a 50% pure LDC on an x86 machine is itself a significant catch for a quantum machine. By combining the famous Bloch sphere model [42] representing multiple quibit states with the 4D hypercube grid, the building of universal chars would grow exponentially, since all bit flags are self-embedded within the hypercube grid structure. Therefore, extracting flags based on 'char priority' in entanglement would be super-condensed into half a signal of a wave, containing the FBAR total code information with intact length into shortest forms possible. This new model incorporation in terms of concatenated addresses from base into the signal is illustrated in Fig. C.1.

Fig. C.1 extracted from Ref. [2], illustrates the concept of having signals refreshed for the fragmented data in form of storable data dots via fuzzy qubit lattice sites. Every dot represents a Boolean bit value memory address, bit polarity or bit position (see Fig. C.1) governing the possibility of having information so compressed beyond the classical limits of data computation.

Possessing memory address, bit location and polarity, permits the algorithm's later stages to decompress data successfully, without losing 1-bit of information. Data retrieval at the extreme compression level becomes imminent despite of having the two co-local parallel signals 'asynchronous' in behavior. The main reason is that the new bi-lattice sites refresh the signal by having necessary 'off and on' bits of information at upper lattice site's disposal. Once the signal, either of two being delayed, the refreshing process (an iterative loop) compensates signals in data integrity with equal fragments of data, adjacent to one another.

The principle aspect in theory obeys 'Bose-Einstein condensation' and 'photonic projection' from one upper linear layer to lower layer(s) of qubit registers with a necessary design alteration (upgrade) in the lattice site [23]. The practical aspect of presenting FQAR becomes quite feasible akin to Phillips' research team [24, 25], once we satisfy the design principles of the fqubit registers for an ultimate compression. Once the new design is implemented, the grand scale of its work could be applied to server architectures addressing memory organization, its management and communications. Optimum loads (based on bitrate) of maximally-self-compressed data in memory cells, fixates reliable paths in data access and transactions between databases and their applications.

In our next reports, we suggest physical methods of the hardware design (Figs. C.1, C.2), which uses optical projection of bits [23] and matter condensation [37] in form of atomic shift registers, like Whitlock *et al.* [38]. In our design, we commit **znip** bits (Appdx. A) to memory/grid according to FBAR projections, simplifying Alice and Bob entanglement concept by super-compressing an encoded message, thereby decode and decompress. The FBAR logic would then be called as FQAR or quantum *and-or* in its maximum performance of LDC. Hence, a negative entropy < 0 bits/byte of Eq. (6), denoting double efficiency above 87.5%, for a universal predictability, is not farfetched in reality.

C.2.2. The extended grid model for ultimate LDCs

In the classical version of bit computation on current computers, however, the 'current challenge' was to implement the FBAR's 4th layer projections commencing with an 8-bit to 5-bit iterative compression, which yielded an estimate of 37.5% compression. From there, condensed techniques were applicable in form of a 4D hypercube model, practically satisfying 8-bit to 4-bit sequential compressions, which yielded as estimate of 50% compression. Further challenges meet those compression values generated from $2n$:1 ratios. The 50% compression is significant to prove the possibility of a guaranteed superdense coding technique [3, 34] in *quantum information theory*.

Future extreme compressions obeying this model deliver highest possible percentages of LDCs without worrying about quality loss and data integrity variations. Self-embedded efficient space and temporal limits of computation shall be granted as a result of the FBAR/FQAR models, yielding future advances in information technology. We aim to publish our next report in some quantum information and computation journal for the extended 4D model, whereby the current abstract is reformulated as follows (subject to change before publication):

Abstract – We report a new lossless data compression (LDC) algorithm for implementing predictably-fixed compression values on quantum computers. The *fuzzy qubinary and-or* (FQAR) algorithm, primarily aims to introduce a new model for superdense coding in quantum information theory. Classical coding on x86 machines would not suffice techniques for maximum LDCs, generating fixed values of $C_r \geq 2{:}1$. We have previously implemented the compression and simulated the decompression phase using *fuzzy binary and-or* (FBAR) logic. This compression was done through a 4D hypercube consisting of self-embedded 4×1-bit flags, for at least a 2-original character input, resulting in 1 character (a *superposing char*) representing the original. This resulted in a 50% compression proving a superdense transmission on 16 classical bits via 8 qubits (doubled efficiency) with a pure superposition of both 0 and 1 logic. In contrast with probabilistic LDCs that use Shannon entropy, the current model with `fuzzy qubinary' entropy, is presented in form of a combinatorial Bloch sphere hypercube (*hyper-sphericube*). This model is a stateful improvement on its predecessor to reconstruct 8 original chars via 1 compressed char, yielding a C_r= 8:1 or 87.5% compression. The current fuzzy qubit model shows an exponential compression growth for $2n$-char:1 input scenarios, denoting *universal predictability* with a *negentropy* < 0 bits/byte. We conclude that this model is a steppingstone to quantum information models solving complex negative entropies, giving LDCs > 87.5% space savings for an ultimate LDC.

C.3 Conclusion

In this study, we introduced and implemented FBAR logic, thereby evaluated its lossless compression ability compared to other known compressors.

We conclude that almost every LDC uses probabilistic Shannon entropy as its ‘logic base’ in conducting lossless compression. However, we conclude that our LDC performs fixed compression ratios, contrasting probabilistic standards of a typical LDC algorithm. We thus conclude that, our algorithm contains predictable values due to a self-embedded 4×1-bit flag structure for each two-character input.

We finally claim that this algorithm is novel in most aspects such as encryption, binary, fuzzy and quantum information methods. To this account, the fields of interest encompass the newly-born FBAR models useful to *quantum information theory mathematicians*, as well as *computer scientists* for its logic, and *software engineers* for its applications.

FBAR can be used for different kinds of input values. The system was tested on an x86 machine, under both UNIX and Windows platforms.

Test samples as char-based documents, e.g., HTML, LaTeX and plain text, were examined for our prototype and compared with other compressors, varying from low-average to high ranks.

FBAR could achieve higher space saving percentages, above 50% as estimated, simulated and discussed in theory from its quantum state protocol, once linked reciprocally with fuzzy and classical binary. Future generation computers, by using this model, e.g., combining the 4D grid model with the famous Bloch sphere in quantum information, could sustain a great deal of space and bitrate savings.

The “LDD simulation”, on the other hand, allowed us to study FBAR products from our experiment’s future challenges. Its implemental state performed a maximum possible compression based on FBAR logic orienting to FQAR, approximating a 0 space memory occupation, yielding a data compression greater or equal to 87.5%. FBAR is independent of data content, since it follows one, and only one, concrete rule of logic: *the AND/OR logic, impure-pure pairwise states applicable to data content itself.* In FQAR, negative entropy denoting *universal predictably* giving values ≥ 93.75% compression is estimated. We conclude that, this model could be considered as a solution to *complex negentropy problems* in signal processing and information theory. That is why this attribute of FBAR to FQAR, ranks it as a fresh way to compress data losslessly amongst other compressors used today.

References

[1] K. Gödel, *Über die Vollständigkeit des Logikkalküls*. Doctoral dissertation. University Of Vienna. The first proof of the completeness theorem, 1929.

[2] P. B. Alipour, ‘A Fuzzy Binary AND/OR Compressor’, *arXiv:0910.2066, Comp. Sci. and Math., Info. Theo.*, pre-print v.2, pp. 1-44, Oct 2009.

[3] D. Joiner (Ed.), ‘Coding Theory and Cryptography’, *Springer*, pp. 151-228, 2000.

[4] J. S. Walker, ‘A Primer on Wavelets and Their Scientific Applications’, 2nd ed., 2008.

[5] “Fuzzy Logic”. *Stanford Encyclopedia of Philosophy. Stanford University*. 2006. http://plato.stanford.edu/entries/logic-fuzzy. Retrieved in 2008.

[6] J. Ziv and A. Lempel, “A Universal Algorithm for Sequential Data Compression,” *IEEE Transactions on Information Theory*, Vol. 23, pp. 337–342, 1977.

[7] J. Ziv and A. Lempel, “Compression of Individual Sequences Via Variable-Rate Coding,” *IEEE Transactions on Information Theory*, Vol. 24, pp. 530–536, 1978.

[8] T. A. Welch, “A Technique for High-Performance Data Compression,” *Computer*, pp. 8-18, 1984.

[9] L. A. Zadeh *et al.*, Fuzzy Sets, Fuzzy Logic, Fuzzy Systems, *World Scientific Press* 1996.

[10] L. A. Zadeh, “Fuzzy algorithms”. *Information and Control* 12 (2): 94–102, 1968.

[11] L. A. Zadeh, “Fuzzy sets”. *Information and Control* 8 (3): 338-353, 1965.

[12] J. Miller, P. Flor, G. Berg, and J. G. Cabill´, on the “Pigeonhole principle”. In Jeff Miller (ed.) *Earliest Known Uses of Some of the Words of Mathematics*. Electronic document, retrieved in 2006.

[13] V. Engelson, D. Fritzson, and P. Fritzson. Lossless Compression of High-volume Numerical Data from Simulations. *In Proceedings of the Conference on Data Compression (March 28 - 30, 2000). DCC. IEEE Comp. Soci.*, USA, 574, 2000.

[14] S. W. Smith, “Digital Signal Processing: A Practical Guide for Engineers and Scientists”, Chap. 27, *Newnes/Elsevier.*, 2007.

[15] C. E. Shannon, “The Mathematical Theory of Communication,” *Univ. of Illinois Press, Champaign*. 1998.

[16] C. E. Shannon, “A Mathematical Theory of Communication”. *Bell System Technical Journal*, Vol. 27, pp. 379-423, 623-656, 1948.

[17] J. A. Patel, B. Cho, I. Gupta, Confluence: A System for Lossless Multi-Source Single-Sink Data Collection, Distributed Systems & Computer Networks, *Dept. of Comp. Sci., University of Illinois, USA*, pp. 1-3. http://hdl.handle.net/2142/13158 Accessed Jun 2009

[18] C. E. Shannon, Redirected from EFF: *Electronic Foundation Frontier group*, Home database: http://www.data-compression.com/index.shtml Accessed Jun 2009.
[19] P. Viana, A. Gordon-Ross, E. Barros, and F. Vahid, A table-based method for single-pass cache optimization. *In Proceedings of the 18th ACM Great Lakes Symposium on VLSI, Cat. Cryptography and Architecture*. GLSVLSI '08. ACM, New York, NY, pp. 71-76, 2008
[20] M. Czachor, "Notes on nonlinear quantum algorithms", Report No. *quant- arXiv.org*: ph/9802051v2. 1998.
[21] D. S. Abrams and S. Lloyd, "Nonlinear quantum mechanics implies polynomial-time solution for NP-complete and #P problems", *arXiv.org*: Report No. quantph/9801041, 1998.
[22] English text, *CIA World Fact Book*, Lossless data compression software benchmarks/comparisons, Maximum Compression, http://www.maximumcompression.com/data/text.php, Accessed Sep. 2009.
[23] S. Zhang, Z. Li, Y. Liu, R. Geldenhuys, H. Ju, M.T. Hill, D. Lenstra, G.D. Khoe and H.J.S. Dorren, "Optical shift register based on an optical flip-flop with a single active element", *Proceedings Symposium IEEE/LEOS Benelux Chapter*, Ghent, 2004.
[24] E. Arimondo, I. Bloch and D. Meschede, Atomic q-bits and optical lattices, Retrieved from ftp://ftp.cordis.europa.eu/pub/ist/docs/fet/qip2-eu-28.pdf, Sep. 29, 2009.
[25] B. Phillips, Talk from 10/6/2003, "Neutral Atoms as Qubits", *Fujitsu Lectures, University of Cambridge*, 2003, at: http://sms.cam.ac.uk/media/650257 Accessed Jun 2009
[26] C. D. Meyer, "Matrix Analysis and Applied Linear Algebra*", Society for Industrial and Applied Mathematics*, 2000.
[27] D. Schrader, I. Dotsenko, M. Khudaverdyan, Y. Miroshnychenko, A. Rauschenbeutel & D. Meshede (2004) Neutral Atom Quantum Register. Phys. Rev. Lett., 93, 150501.
[28] P. A. M. Dirac. The principles of quantum mechanics (Fourth Edition ed.). Oxford UK: *Oxford University Press*. p. 18 ff, 1982.
[29] Entisoft Developers, String Insertion, dedicated to Microsoft Visual Basic and Office development tools 1996-1999, at http://www.entisoft.com/estools/StringManipulations, Accessed Oct. 2009.
[30] L. Debnath, 'Wavelets and signal processing ', *Springer*, pp. 177-217, 2003.
[31] F. Pistono, "*Open Source implementations of encoding algorithms for video distribution in the HTML5 era*", Dept. of Comp. Sci., University of Verona, Italy, 2010.
[32] I. Veldman, A. Keijzer and M. van Keulen, Compression of Probabilistic XML Documents, *Cat. Comp. Sci., Springer Berlin/ Heidelberg*, Vol. 5785/2009, pp. 255-267, 2009.
[33] X. Xie and Q. Qin, Fast Lossless Compression of Seismic Floating-Point Data, *IEEE Comp. Soci*., pp. 235-238, 2009.
[34] C. Bennett and S. J. Wiesner. Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states. *Phys. Rev. Lett.*, 69:2881, 1992.
[35] R. R. Kowalski, "Algorithm = Logic + Control". *Communications of the ACM* **22** (7): 424–436, 1979.
[36] *CCSDS Green Book*. Informational Report Concerning Space Data System Standards, Lossless Data Compression, *CCSDS* 120.0-G-2, Section 3.1, 42 pages, 2006.
[37] S. Peil, J. V. Porto, and W. D. Phillips *et al.*, Patterned loading of a Bose–Einstein condensate into an optical lattice, *Phys. Rev.* A **67** 051603, 2003.
[38] S Whitlock1, R Gerritsma2, T Fernholz3 and R J C Spreeuw. Two-dimensional array of microtraps with atomic shift register on a chip, *New J. Phys.* **11** 023021, 2009.
[39] IBM Corporation, *Structure, Declaring and Using Bit Fields in Structures* http://publib.boulder.ibm.com/infocenter/lnxpcomp/v7v91/index.jsp?topic=/com.ibm.vacpp7l.doc/language/ref/clrc03defbitf.htm Accessed Feb. 2010
[40] R. Guastella, Lossless Data Compression for Embedded Systems, Courtesy of Embedded.com , at http://www.embedded.com/opensource/217800397Accessed Apr. 2010
[41] B. Huntting and DavidMertz, 'A guide to solving intractable problems simply', *Introduction to Quantum Computing, IBM, Brad Hunting University of Colorado and Gnosis Software, Inc.*, at: http://www.ibm.com/developerworks/linux/library/l-quant.html Accessed Oct. 2009.
[42] D. Chruściński, "Geometric Aspect of Quantum Mechanics and Quantum Entanglement", *Journal of Physics Conference Series*, **39**, pp.9-16, 2006.
[43] A. Hyvärinen and E. Oja, *Independent Component Analysis: A Tutorial, node14: Negentropy*, Helsinki University of Technology Laboratory of Computer and Information Science, 1999.
[44] G. Boole, *Cambridge and Dublin Mathematical Journal*, Vol. III, pp. 183-98, 1848.
[45] C. E. Shannon, *A symbolic analysis of relay and switching circuits,* Massachusetts Institute of Technology, Dept. of Electrical Engineering, 1940.
[46] M. Nelson, LZW Data Compression, *DJJ journal*, 1989, or see for instance: http://www.cs.cf.ac.uk/Dave/Multimedia/Lecture_Examples/Compression/lzw/lzw_docs/LZW_Data_Compression.htm , Accessed Apr. 2010.
[47] S. W. Smith, The Scientist & Engineer's Guide to Digital Signal Processing, *California Technical Pub. 1st Ed.*, Chap. 27, 1997.
[48] A. Kerckhoffs, "La cryptographie militaire," *Journal des sciences militaires*, vol. IX, pp. 5–83, Jan. 1883, pp. 161–191, Feb. 1883. (http://petitcolas.net/fabien/kerckhoffs/)