Cloud Computing and Data Science
https://ojs.wiserpub.com/index.php/CCDS/

UNIVERSAL WISER PUBLISHER

Research Article

Pongo: Efficient Lossless Floating Point Compression

Yufeng Liu[1], Yao Shen[1*], Fenghua Zhang[1], Feiteng Huang[2]

[1]School of Computer Science, Shanghai Jiao Tong University, Shanghai, China
[2]Huawei Cloud Database Innovation Lab, China
E-mail: yshen@cs.sjtu.edu.cn

Received: 27 April 2025; **Revised:** 9 July 2025; **Accepted:** 9 July 2025

Abstract: A large amount of time series data is increasingly being collected in different fields. In order to make good use of this large amount of time series data, it is necessary to solve the problems of high storage costs and transmission bandwidth that the data bring. The general compression algorithms effectively reduce the size of data at the cost of a large amount of computation. However, due to the huge time cost and batch processing mode of the general compression algorithms, Time Series Management Systems (TSMSs) often use streaming compression algorithms to replace the general compression algorithms for compressing time series data. For floating-point data, the most prevalent streaming compression algorithms, such as those based on exclusive OR (XOR) operations, offer relatively fast processing and high compression ratios compared to conventional general-purpose compression algorithms. Among them, the Elf algorithm proposes the idea of first erasing and then compressing, achieving the best compression ratio among existing streaming compression algorithms. This paper proposes a new lossless streaming compression algorithm, Pongo, for floating-point numbers, which uses a carefully designed erasing method different from Elf. The Pongo algorithm employs a novel erasing technique that transforms the binary representation of fractional parts to decimal, leveraging a newly proposed algorithm that enhances the efficiency of this conversion process. To demonstrate the superior performance of Pongo, we conducted extensive experiments comparing it with ten leading compression algorithms across twenty-two different datasets. On average, Pongo achieves a compression ratio that is 14% better than Elf and 58% better than Gorilla, making it the top-performing algorithm among all those tested, as shown through both mathematical analysis and practical testing.

***Keywords*:** floating point compression, lossless compression, time series data, decimal native numbers

1. Introduction

Time series data is one of the most important data types, and it is increasingly being collected in many different fields [1]. This includes but is not limited to Aviation, Computers, Energy, Finance, Logistics, and Healthcare. The increased deployment of sensors for monitoring extensive industrial systems has led to a rise in the availability of data that can now be efficiently analyzed, enabling automation and remote management at an unprecedented scale [2]. For example, the sensors of the Boeing 787 can generate half a terabyte of data per flight [3]. Therefore, in order to make good use of these large amounts of time series data, it is necessary to solve the problems of high storage costs and transmission bandwidth that the data bring.

DOI: https://doi.org/10.37256/ccds.6220257102

Compression is the most effective means of addressing these challenges by reducing the overall space requirements. However, general compression algorithms such as LZ4 [4] and Xz [5] have significant time costs and do not leverage the internal characteristics of these time series data. In addition, most of these general compression algorithms are performed in batch processing mode, so general compression algorithms cannot be directly applied to streaming time series data. Recently, Time Series Management Systems (TSMSs) [6] often use streaming compression algorithms instead of general compression algorithms to compress streaming time series data. These streaming algorithms are very fast compared to general compression algorithms and also provide a high compression ratio. Moreover, these streaming algorithms can compress streaming data, which is very suitable for application in TSMSs.

Most streaming compression algorithms for floating-point time series data are XOR-based, and Gorilla [7] is the most representative algorithm among them. This type of XOR-based compression algorithms utilize the similarity between two values in time series data, so it can balance compression time and compression ratio. Gorilla [7], Chimp [8], and Erasing-based Lossless Floating-point compression (Elf) [9] are three state-of-the-art XOR-based lossless floating-point compression algorithms. These methods leverage temporal locality and XOR encoding to achieve high throughput, but often struggle with decimal-native numbers and high-precision floats, either introducing minor errors or leaving redundancy unexploited.

In this paper, we propose our *Pongo* algorithm. As shown in Figure 1, we analyze the double number 3.18. The red part consists of the sign, component, and integer parts in 3.18, while the black part represents the fractional part of 3.18 (i.e. 0.18). Our idea is to use $(10010)_2$ instead of representing 0.18. In fact, $(10010)_2$ is first reversed to $(01001)_2$, and the reason for this will be explained in detail in Section 3.2.1. So 3.18 will ultimately be transformed into a binary representation that leads to the XOR result with many trailing zeros. As shown in Figure 2, if we XOR two adjacent numbers 3.17 and 3.18, we will obtain the result depicted in the figure, which has 16 leading zeros and 46 trailing zeros. This example demonstrates that *Pongo* can enhance the compression ratio of floating-point data by re-encoding it, leading to a significant number of trailing zeros. Unlike Elf, which directly erases the last few bits of a floating-point number, *Pongo* re-encodes the entire number, and the re-encoded number's last few bits are filled with zeros. Consequently, floating-point numbers processed by *Pongo* will also result in a large number of trailing zeros, similar to Elf.

3.18: 0100000000001001011100001010001111010111000010100011110101110001

re-encode

3.18: 0100000000001100100000000000000000000000000000000000000000000000

inversion

3.18: 0100000000001010010000000000000000000000000000000000000000000000

Figure 1. *Pongo*'s erasing process for floating-point numbers

3.18(erasered): 0100000000001010010000000000000000000000000000000000000000000000

XOR

3.17(erasered): 0100000000001100010000000000000000000000000000000000000000000000

XOR Result: 0000000000000110000000000000000000000000000000000000000000000000

Figure 2. An example of *Pongo* compressing adjacent numbers

At the same time, we find that there are errors in the accuracy of Elf that are not resolved (Elf's code implementation avoids this error issue by checking whether the number before and after erasing is the same), and *Pongo* also has such problems. As shown in Figure 3a and b are two floating-point numbers of double type. 3.18 converting to the corresponding double type floating-point number is a. The only difference between a and b is that the last digit of a is 1, while the last digit of b is 0. For a, converting a to decimal is 3.18, but for b, converting b to decimal is also 3.18. The occurrence of this problem is due to a natural error between binary and decimal numbers. We will discuss this issue in detail in Section 4. In this paper, we analyze and solve this error problem and propose a new algorithm that can determine whether a floating-point number is a decimal native (detailed in Section 2.1) number and convert binary numbers to decimal numbers. This algorithm significantly improves the efficiency of converting binary numbers to decimal numbers compared to traditional methods.

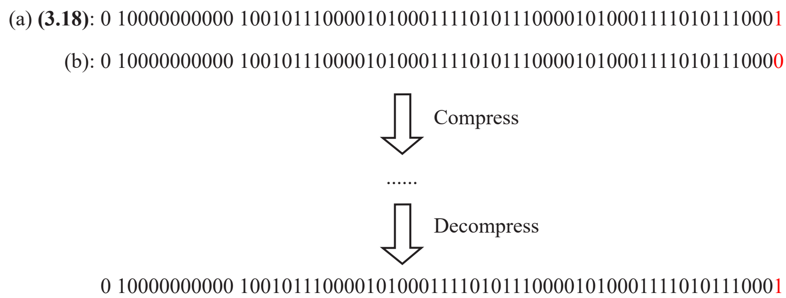

Figure 3. An example of the error problem

The motivation of this work stems from the need for more efficient streaming compression of floating-point time series data, particularly in scenarios where values originate from decimal representations. These values tend to have decimal-native characteristics—meaning they can be exactly represented in base-10 but not necessarily in binary. However, most existing streaming compressors, such as Gorilla [7], Chimp [8], Chimp_{128} [8], and Elf [9], operate entirely in the binary domain, without considering the semantic structure of the source data. As a result, they may risk irreversible transformation when attempting compression. Elf, in particular, erases last few bits skillfully before XOR encoding, significantly improving the compression ratio over Gorilla. However, Elf's erasure is based purely on binary rounding rules and does not distinguish between decimal-native and non-native values. This leads to potential reversibility issues and limits its ability to eliminate redundancy when handling decimal-heavy datasets. Motivated by these limitations, we propose *Pongo*, a novel decimal-aware lossless floating-point compression algorithm. *Pongo* introduces a reversible erasure mechanism that precisely identifies decimalnative number. This erasure is followed by a standard XOR-based delta compression, achieving better compression ratios without sacrificing precision. Furthermore, we design an optimized binary-to-decimal conversion algorithm that accelerates the decimal-native detection process, reducing computation time while maintaining correctness. Then we implement our *Pongo* compression algorithm using Java and compare it with 11 other compression methods for floating point data.

We summarize the key contributions of our proposed compression algorithm here, and we:

• Design and implement *Pongo*, a new lossless compression algorithm tailored for floating-point time series data, which integrates a novel reversible decimal-aware erasure technique with a theoretical compression ratio that surpasses that of existing methods.

• Discover and rigorously analyze a conversion error inherent in binary-to-decimal fraction algorithms existing in both Elf and *Pongo*, then propose a corrected and efficient alternative.

• Provide a more efficient algorithm than traditional methods for converting binary fractions to decimal numbers, which addresses the error problem, enabling practical deployment.

• Conduct extensive experiments on 22 real-world and synthetic datasets, demonstrating that *Pongo* consistently achieves higher average compression ratios than other lossless compressors, including Elf, while maintaining competitive compression and decompression speeds.

In the rest of this paper, related works are reviewed in Section 2. We present some preliminaries in Section

3. In Section 4, we give our design and implementation of *Pongo* algorithms, including the design concept and implementation details of *Pongo* Eraser and *Pongo* Restorer, as well as some key points during the preprocessing steps of floating-point numbers. Section 5 elaborates on the error problems that exist in both Elf and *Pongo*, and analyzes the reasons for the occurrence of this kind of error. Then we provide our solution for avoiding the impact of such errors in lossless compression and propose our optimized *Pongo* algorithm. The background details ofthe experimental setting and the experimental results are presented in Section 6. Finally, we conclude the paper and discuss future research directions in Section 7.

2. Related works

Many lossy compression algorithms for floating-point data, such as ZFP [10] and other examples [11-18], have been developed with a focus on specialized scientific applications. However, these algorithms are generally unsuitable for database applications that demand strictly lossless storage; therefore, they fall outside the scope of this study.

Thus, lossless compression of floating-point data has attracted extensive attention due to its importance in scientific and real-time applications. General-purpose compressors such as LZ4 [4], Zstandard (Zstd) [19], and Xz [5] are widely used, but they treat binary data as opaque byte streams and fail to utilize the internal structure of IEEE 754 floating-point numbers. Consequently, these dictionary-based, batch-oriented methods are effective for a broad range of data types but often fail to exploit temporal and structural patterns inherent in floating-point time series data, resulting in suboptimal performance in streaming scenarios.

To overcome these limitations, dedicated streaming lossless compression algorithms for floating-point sequences have been proposed. Predictor-based techniques, such as Finite Context Method predictors (FCM) [20], Differential FCM (DFCM) [21], and Lorenzo [22-26]. Alternatively, XOR-based approaches—such as Gorilla [7], Chimp [8], Chimp_{128} [8], and Elf [9]—use bitwise differencing and zero-run encoding to facilitate efficient real-time compression. Gorilla [7], Chimp [8], and Elf [9] are three state-of-the-art XOR-based lossless floating-point compression algorithms. Gorilla assumes that the XOR result of two adjacent floating-point numbers is highly likely to have a large number of leading and trailing zeros simultaneously. However, Chimp's survey indicates that most XOR results exhibit a significant number of leading zeros but are unlikely to have a significant number of trailing zeros [8]. And Chimp provides a very space saving approach based on the actual distribution of leading and trailing zeros. Chimp_{128} [8] is an optimized version of Chimp, selecting one of the top 128 values to produce the XOR result with the most trailing zeros. This way, the XOR result has both a large number of leading and trailing zeros, allowing Chimp_{128} to achieve a significant improvement in compression ratio. And Elf builds upon the idea of Chimp_{128}, which is that increasing the number of trailing zeros in XOR results significantly improves the compression ratio of time series. These methods exploit the presence of leading or trailing zeros in the XOR results for compact encoding. However, they often overlook the inherent structure of decimal-native values. Elf [9] improves on XOR-based methods by introducing an erasure step that removes non-informative bits before XOR encoding. The core idea of Elf is to erase the last few bits of a floating-point number (i.e. set them to zero) to obtain a XOR result with a large number of trailing zeros. Through this approach, Elf has achieved a very high compression ratio, surpassing that of other algorithms. While it achieves excellent compression in many cases, Elf assumes a fixed erasure strategy and does not account for the semantics of decimal representations, which can lead to precision degradation or missed opportunities for compression. Our *Pongo* algorithm builds on the preprocessing methodology of Elf and further integrates an optimized XOR encoding scheme to enhance compressibility.

Recent research has advanced adaptive strategies for time series compression. For instance, Adaptive lossless Floating-point Compression (AFC) algorithm [27] selects appropriate compression strategies according to data characteristics, An efficient lossless Compression algorithm for Time series Floating-point data (ACTF) [28] incorporates preprocessing and flexible coding techniques, and Adaptive Lossless floating-Point compression (ALP) [29] utilizes vectorized procedures to improve both throughput and compression ratio. In contrast, *Pongo* is specifically tailored for high-throughput database environments, ensuring a robust trade-off between processing speed and compression efficiency by means of lightweight value preprocessing and efficient XOR-based coding, especially when handling large-scale, heterogeneous floating-point datasets.

3. Preliminaries

This section begins by defining essential terms and then delves into the IEEE754 standard's double-precision format, the XOR-based compression method, and the state-of-the-art Elf algorithm.

3.1 *Definitions*

Definition 1 Floating-Point Time Series. A floating-point time series TS is a sequence of data points, represented as times-tamp and value pairs, ordered by time in increasing order: $TS = <(t_1, v_1), (t_2, v_2), ...>$. For each pair of time series, t_i represents the timestamp and value v_i represents a floating-point number.

Definition 2 Decimal number. The representation of a decimal number is $\pm (d_{h-1}d_{h-2} \ldots d_0.d_{-1}d_{-2} \ldots d_l)_{10}$, so we have the value of the number:

$$v = \pm \sum_{i=l}^{h-1} d_i \times 10^i \tag{1}$$

where d_i is the i-th decimal digit with $0 \leq d_i \leq 9$, l and h are the indices of the least and most significant digits, respectively. The total number of significant decimal digits is $D_S = h - l$.

Definition 3 Binary number. The representation of a binary number is $\pm (b_{h-1}b_{h-2} \ldots b_0.b_{-1}b_{-2} \ldots b_l)_2$, so we have the value of the number:

$$v = \pm \sum_{i=l}^{h-1} b_i \times 2^i \tag{2}$$

where b_i is the i-th binary digit ($b_i \in \{0, 1\}$), and l, h have the same meaning as in decimal. The number of significant binary bits is denoted by $B_S = h - l$.

It is important to note that the representation range for binary numbers differs from that for decimal numbers. For example, the decimal number 0.3 is represented as an infinite binary sequence $(0.01001100110011...)_2$, indicating that no finite binary number can accurately represent 0.3. Additionally, it should be noted that floating-point numbers, being a form of binary representation, can accurately represent binary numbers within their valid range. However, they cannot represent all decimal numbers within this range.

Definition 4 Decimal native. A floating-point number f_1 is considered decimal native if, when converted to a decimal number $d = \pm (d_{h-1}d_{h-2} \ldots d_0.d_{-1}d_{-2} \ldots d_l)_{10}$, and then converted back to a floating-point number f_2, f_2 matches f_1 exactly. The decimal significant figure $DS = h - l$ of d must match the decimal significant figure of the floating-point number to maintain accuracy. For float precision type, DS is 6, and for double precision type, DS is 15.

It is worth noting that errors can occur during the conversion of floating-point numbers to decimal and vice versa. As depicted in Figure 3, a is decimal native because it is derived from a decimal number, whereas b is not decimal native.

3.2 *IEEE754 double precision floating point format*

IEEE standard 754 for binary floating-point arithmetic (IEEE 754) has been the most widely used floating-point arithmetic standard since the 1980s and is adopted by many CPUs and floating-point arithmetic operators [30]. This standard defines two fundamental formats: *float*, represented by 32 bits, and *double*, represented by 64 bits. Modern computer systems and programming languages mostly support these two floating-point formats, and the accuracy of the *double* type is higher than that of the *float* type. Due to the higher precision provided by the *double* type, the compression of *double* numbers is the primary focus of this discussion. The compression of *float* numbers can be derived from the techniques developed for *double* compression.

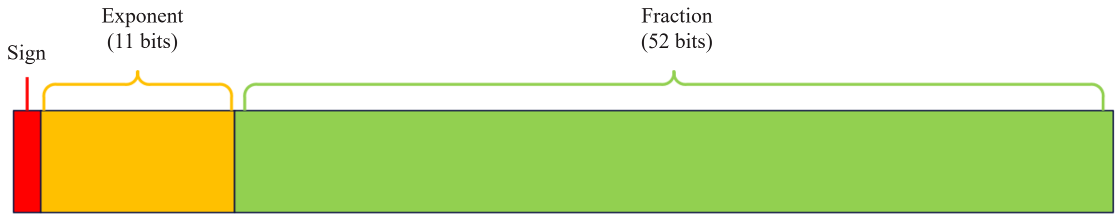

Figure 4. IEEE754 double type format

Figure 4 shows the IEEE 754 double precision floating-point data type, which consists of *sign* bit, *exponent* field, and *fraction* field. The *sign* bit S represents the positive or negative value of this floating-point number, where 0 represents positive and 1 represents negative. The *exponential* field E contains 11 bits, representing the exponent of this floating-point number. The *fractional* part F is 52 bits, which contains the significant number of this number. The value of a double precision floating-point number with this format is:

$$v = (-1)^{S} \times 2^{(E-1,023)} \times 1.F \tag{3}$$

where S is the sign bit (0 for positive, 1 for negative), E is the 11-bit exponent with bias 1,023, and F is the 52-bit fractional part. The term $(1.F)$ represents the normalized mantissa with an implicit leading 1.

3.3 *XOR-based compression method*

The current mainstream streaming floating-point data compression methods are based on XOR methods. Figure 5 shows the process of XOR two floating-point numbers, and the middle part of the resulting sequence is known as the *center bits*. The preceding zeros are collectively referred to as *leading zeros*, while the following zeros are collectively referred to as *trailing zeros*. Since floating-point time series possess the characteristic that two consecutive numbers in the time series tend to be similar, and performing an XOR operation on a floating-point number with its predecessor, the result could contain many leading zeros. Besides, *Pongo* is also an erasing-based algorithm, which means the result could also have many trailing zeros. Thus, the use of XOR-based compression methods can significantly improve the compression ratio and is highly suitable for floating-point time series data.

3.75: 0100000000001110000000000000000000000000000000000000000000000000

XOR

3.17: 0100000000001001010111000010100011110101110000101000111101011100

||

XOR Result: 0000000000000111010111000010100011110101110000101000111101011100

Figure 5. An example of XOR methods

3.4 *Elf compression*

Elf [9] introduces a streaming compression for floating-point data. The Elf algorithm has a higher compression ratio compared to Gorilla and Chimp.

As shown in Figure 6a, the compression phase of Elf involves *ElfEraser* and $ElfXOR_{cmp}$, while the decompression phase involves *ElfRestorer* and $ElfXOR_{dcmp}$. The specific details of Elf are provided in Algorithm 1, which focuses on the compression phase (denoted by v for the value to be compressed and *out* for the output stream).

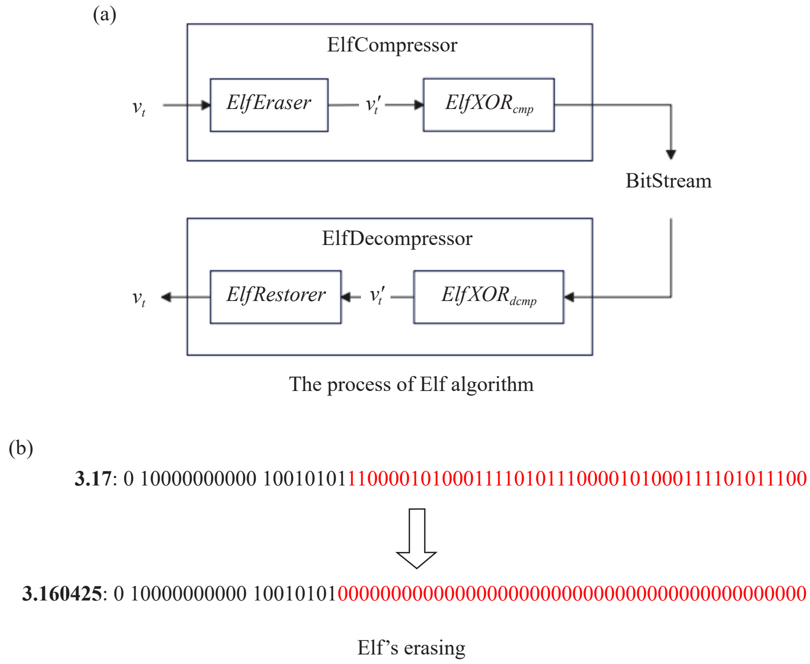

Figure 6. The main idea of Elf

ElfEraser and *ElfRestorer* are responsible for erasing and restoring floating-point numbers. As depicted in Figure 6b, considering the number 3.17, *ElfEraser* reduces it to 3.1640625 by erasing the last 44 bits, and *ElfRestorer* restores it to 3.17. Since $\delta = 3.17 - 3.1640625 < 0.01$, the recovery process involves only the calculation $3.16 + 0.01 = 3.17$ to obtain the original value of 3.17.

Algorithm 1 ElfCompression

Input: v – original floating-point number, *out* – output stream

Output: Compressed representation of v written to *out*

1 $v' \leftarrow ElfEraser\ (v, out)$;

2 $ElfXOR_{cmp}\ (v', out)$.

The specifics of $ElfXOR_{cmp}$ are detailed in Algorithm 2. $ElfXOR_{dcmp}$ is the decompression counterpart of $ElfXOR_{cmp}$, which will not be delineated in detail here. Elf uses '0', '01', and '1' to identify three different situations:

• '01' signifies that the XOR result is zero, indicating that the current floating-point number is identical to the preceding one (line 7).

• '00' indicates that the XOR result is non-zero, yet the number of leading zeros is constant (line 14).

• '1' indicates that the XOR result is non-zero with differing leading zeros. In such cases, the complete information of the number must be recorded. Depending on the number of center bits, there are two methods of recording (line 16 and line 19).

We find that *Pongo* and Elf share similar effects on floating-point erasing, and the designs of $ElfXOR_{cmp}$ and $ElfXOR_{dcmp}$ are highly effective, leading *Pongo* to incorporate them internally.

Algorithm 2 $ElfXOR_{cmp}\ (v_t', out)$

Input: v_t' – erased value, *out* – output stream

Output: XOR-encoded result written to *out*

1 **if** v_t' is the first value **then**

2 $\quad lead_t \leftarrow \infty$; $trail_t \leftarrow numOfTrailingZeros\ (v_t')$;

3 $\quad out.write\ (trail_t, 7)$;

4 $\quad out.write\ (nonTrailingBits\ (v_t'), 64 - trail_t)$;

5 **else**

6 $\quad xor \leftarrow v_t' \oplus v_{t-1}'$;

```
    if xor = 0 then
        out.writeBit ("01");
        lead_t ← lead_{t−1}; trail_t ← trail_{t−1};
    else
        lead_t ← binNumOfLeadingzeros (xor);
        trail_t ← numOfTrailingzeros (xor);
        center ← 64 − lead_t − trail_t;
        if lead_t = lead_{t−1} and trail_t ≥ trail_{t−1} then
            out.writeBit ("00");
        else if center ≤ 16 then
            out.writeBit ("10");
            out.write (lead_t, 3); out.write (center, 4);
        else
            out.writeBit ("11");
            out.write (lead_t, 3); out.write (center, 6);
        end
        out.write (center Bits (v_t′), center);
    end
end
```

4. Our Pongo algorithm

We introduce our *Pongo* algorithm in this section, which is a streaming floating-point time series compression algorithm. Similar to Elf, *Pongo* comprises four main components: *PongoEraser*, *PongoRestorer*, $ElfXOR_{cmp}$ and $ElfXOR_{dcmp}$. In the following text, we provide the details of *Pongo*.

4.1 *Overview*

During the compression phase, each v_t in the time series is transformed into v_t' by the *PongoEraser*, and the associated flag bits are appended to the compressed stream. Subsequently, $ElfXOR_{cmp}$ compresses v_t' and the resulting XOR output is written into the compressed stream. The complete algorithm workflow is illustrated in Algorithm 3 and Algorithm 4.

Algorithm 3 *PongoCompression* (*v*, *out*)
Input: *v* – original float, *out* – output stream
Output: Compressed representation of *v* written to *out*

```
v′ ← PongoEraser (v, out);
ElfXOR_cmp (v′, out).
```

Algorithm 4 *PongoDecompression* (*in*)
Input: *in* – compressed input stream
Output: *v* – restored floating-point number

```
flag ← in.read (1);
if flag == '0' then
    if in.read (1) == '0' then
        flag ← '00';
        flag ← flag + in.read (4);
    else
        flag ← '01';
    end
end
v′ ← ElfXOR_dcmp (in);
```

11 $v \leftarrow$ *PongoRestorer* $(v', flag)$.

The essence of the *Pongo* algorithm lies in its methods for erasing and restoring the original floating-point numbers, which will be explored in detail in Sections 3.2 and 3.3.

4.2 *PongoEraser*

The primary goal of *PongoEraser* is to re-encode floating-point numbers in a manner that significantly increases their trailing zero count. Figure 2 and Figure 3 show the initial idea of *PongoEraser*, which is to save the fractional part of a floating-point number as an integer. For the fractional part $(0.18)_{10}$ of $(3.18)_{10}$, saving it in binary format requires a very long number of bits. However, if $(0.18)_{10}$ is treated as an integer $(18)_{10}$, only a small number of bits are needed to save it. For the integer $(18)_{10}$, it can be encoded as $(10010)_2$, which requires fewer bits. Indeed, this encoding method is not applicable to all floating-point numbers, such as $(3.18555555555555)_{10}$, which requires a very high number of bits after re-encoding according to this encoding method. That is, this encoding method is beneficial for floating-point numbers with small decimal places but not for floating-point numbers with large decimal places. Next, we will explain *PongoEraser*'s other designs.

4.2.1 *Reverse fraction*

As illustrated in Figure 2, for $v_t = (3.18)_{10}$, during the process of *Pongo*, the decimal portion of v_t is converted to $(10010)_2$. However, when attempting to restore v_t' to v_t, it becomes challenging to differentiate between $(10010)_2$, $(100100)_2$, or $(1001000)_2$ by sequentially examining the bits of v_t'. Recognizing that the first bit of an integer is always 1, we reverse $(10010)_2$ to $(01001)_2$ and save it accordingly. Consequently, $v_t = (3.18)_{10}$ is ultimately transformed into the form depicted at the bottom of Figure 2. During the restoration process, we only need to read forward from the 64th bit until we encounter the first '1' to recover the original fractional part of v_t', which plays an important role in reducing the overhead time consumption of the decompression process in *Pongo*.

4.2.2 *A special situation*

Figure 7 shows a special case, when we compare the results of re-encoding 3.18 and 3.018, we find that their re-encoding results are exactly the same. This is because the fractional part of 3.018 is $(018)_{10}$, which is treated as an integer $(18)_{10}$, resulting in the two numbers being re-encoded with the same result. To resolve this issue, we introduce flag bits to denote the special case, and we use 4 bits to encode the count of trailing zeros (for 3.018, this count is 1). For numbers with more than 16 trailing zeros, they are not re-encoded.

3.18(erasered): 0100000000001010010000000000000000000000000000000000000000000000

3.018(erasered): 0100000000001010010000000000000000000000000000000000000000000000

Figure 7. A special situation

4.2.3 *Flag assign*

As depicted in Figure 8, the re-encoding of v_t can occur under three scenarios, distinct encoding are assigned to each of these scenarios: (1) When v_t has no zeros after the decimal point, it undergoes normal re-encoding. We use only 1 bit and set it to '1' to represent this scenario since it is the most common case. (2) If v_t has zeros after the decimal point, we use 2 bits and set them to "00" to represent this scenario. Meanwhile, an additional 4 bits are used to encode the count of these zeros. (3) In cases where reencoding would result in information loss (i.e., v_t cannot be restored in a lossless manner after re-encoding), v_t is not re-encoded and enters $ElfXOR_{cmp}$ directly. We use 2 bits and set them to "01" to represent this scenario. The determination of this scenario will be addressed in Section 4.

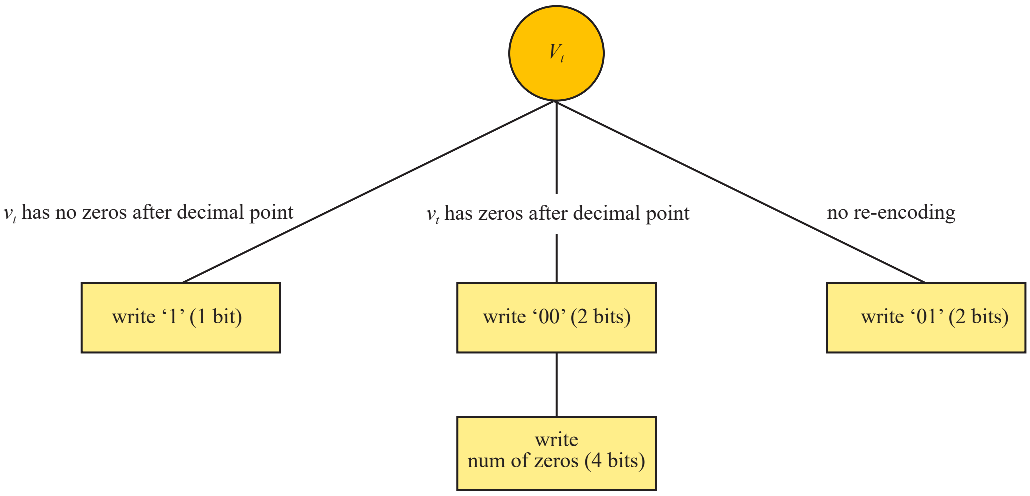

Figure 8. Flag assign

4.2.4 *Summary of PongoEraser*

Following the above analysis, we have developed the *PongoEraser*. The specific details of this component are outlined in Algorithm 5.

Algorithm 5 *PongoEraser* (*v*, *out*)
Input: *v* – original float, out – output stream
Output: *v′* – erased value for XOR compression

```
exp ← getExponentpart (v) − 1,023;
if exp < 0 then
        exp = 0;
end
fraction ← v << (12 + exp);
isDecimalNative <<
optimizedFractionToDecimal (fraction, decimal); //result is stored in decimal, which is an array with a length of 16
if !isDecimalNative then
        out.writeBit ("01");
        return v;
end
if decimal[0] == 0 then
        out.writeBit ("00");
        out.write (numofZeros (decimal), 4);
else
        out.writeBit ("1");
end
fraction ← decimalToBinary (decimal);
fraction ← reverse (fraction); //reverse from 0 bit to 63 bit
v′ ← ((0xffffffffffffffffL << (52 − exp)) & v) | (fraction >> (12 + exp));
return v′.
```

4.3 *PongoRestorer*

Algorithm 6 describes the entire process of *PongoRestorer*. *PongoRestorer* is the complementary process to *PongoEraser*, serving to reverse the transformations applied by *PongoEraser*. Firstly, according to the flag bits added by *PongoEraser*, a compressed number v'_t which has the flag "01" does not need to be restored. Except for this situation, all of the compressed numbers v'_t are converted to reversed integer. Secondly, take a reverse process from what *PongoRestorer* did to get the original decimal portion of v'_t. Next, if v'_t has flag "00", we should pad zeros after v'_t according to the number of zeros recorded in the flag bits, while the compressed number with the flag '1' will skip this step. Finally, the compressed number v'_t is restored to v_t.

Algorithm 6 *PongoRestorer* (v', *flag*)

Input: v' – erased float, *flag* – encoding metadata

Output: v – restored original float

```
if flag == "01" then
    return v';
end
exp ← getExponentPart (v') – 1,023;
if exp < 0 then
    exp = 0;
end
fraction ← (0xffffffffffffffffL >> (12 + exp)) & v';
fraction ← reverse (fraction); //reverse from 0 bit to 63 bit
decimal ← binaryToDecimal (fraction); //decimal is an array with a length of 16.
if flag! = "1" then
    numofZeros ← flag & 0xf;
    addZeros (decimal, numofZeros);
end
fraction ← integerToFraction(decimal);
v ← ((0xffffffffffffffffL << (52 − exp)) & v) | (fraction >> (12 + exp));
return v.
```

5. Error analysis and solution

In this section, we will introduce the error problems that exist in both Elf and *Pongo*, and then we provide our solution and propose our optimized *Pongo* algorithm.

5.1 *Error analysis*

Figure 3 shows an example of this error problem where the initial *Pongo* mistakenly interpret one floating-point number for another. The root cause of this error problem lies in the algorithm for converting binary fractions to decimal fractions. Elf also encounters this problem, and its code implementation addresses this issue by checking whether the two numbers before and after erasing are identical, a detail not included in the original Elf paper. The algorithm for converting binary fractions to decimal fractions is described in Algorithm 7. It actually multiplies the fractional part by $(1010)_2$ (i.e. 10) and carries it to obtain the decimal fractions.

Algorithm 7 *fractionToDecimal* (*fraction*, *decimal*)

Input: *fraction* – binary fraction, *decimal* – output array

Output: *decimal* – array filled with decimal digits

```
fraction ← fraction >> 4;
i ← 0;
while fraction ! = 0 AND i < 16 do
    fraction ← fraction ∗ 0b1010;
    decimal[i] ← fraction >> 60;
    fraction ← fraction & 0x0fffffffffffffffL;
    i ← i + 1;
end
```

Consider the following scenario involving two binary fractions, v_1 and v_2, where v_1 is derived from the decimal number 0.17. The only difference between v_1 and v_2 lies in their least significant bit:

$$v_1: \ 0.0010101110000101000111101011100001010001111010111000$$

$$v_2: \ 0.0010101110000101000111101011100001010001111010111001$$

Despite this minor difference, both v_1 and v_2 convert back to the decimal number 0.17 when represented with 15 decimal places, as illustrated in Table 1 which details the carry operations during the conversion.

Table 1. Carry situation

i	a_i	b_i	i	a_i	b_i
0		0.0010101110000101000111101011100001010001111010111000	0		0.0010101110000101000111101011100001010001111010111001
1	1	0.1011001100110011001100110011001100110011001100110000	1	1	0.1011001100110011001100110011001100110011001100111010
2	6	0.1111111111111111111111111111111111111111111111100000	2	7	0.0000000000000000000000000000000000000000000001000100
3	9	0.1111111111111111111111111111111111111111111011000000	3	0	0.0000000000000000000000000000000000000000001010101000
4	9	0.1111111111111111111111111111111111111111001110000000	4	0	0.0000000000000000000000000000000000000001101010010000
5	9	0.1111111111111111111111111111111111111000001100000000	5	0	0.0000000000000000000000000000000000010000100110100000
6	9	0.1111111111111111111111111111111110110001111000000000	6	0	0.0000000000000000000000000000000010100110000001000000
7	9	0.1111111111111111111111111111110011110010110000000000	7	0	0.0000000000000000000000000000011001111100001010000000
8	9	0.1111111111111111111111111110000101111011100000000000	8	0	0.0000000000000000000000000100000011011001100100000000
9	9	0.1111111111111111111111101100111011010011000000000000	9	0	0.0000000000000000000000101000100001111111101000000000
10	9	0.1111111111111111111101000001010000111110000000000000	10	0	0.0000000000000000000110010101010011111100010000000000
11	9	0.1111111111111111100010001100101001101100000000000000	11	0	0.0000000000000000111111010101000111011010100000000000
12	9	0.1111111111111011010101111110100000111000000000000000	12	0	0.0000000000001001111001010011001010001001000000000000
13	9	0.1111111111010001011011110001001000110000000000000000	13	0	0.0000000001100010111100111111100101011010000000000000
14	9	0.1111111000101110010101101011010111100000000000000000	14	0	0.0000001111011101100001111011110110000100000000000000
15	9	0.1110110111001111011000110001101011000000000000000000	15	0	0.0010011010100111010011010110011100101000000000000000
16	9	0.0100101000011001110111110000101110000000000000000000	16	1	0.1000001010001001000001100000011110010000000000000000

This binary-to-decimal conversion process can cause certain floating-point numbers—such as v_2—to be compressed and then decompressed into a similar but not identical floating-point number. Therefore, the original *Pongo* compression method is not strictly lossless for these numbers.

Figure 9 depicts the positions of these two numbers on the real number line. Neither v_1 nor v_2 exactly represents 0.17, due to their inherent representation errors, quantified by

$$\delta_1 = 0.17 - v_1 = 7.0 \times 10^{-17}, \ \delta_2 = v_2 - 0.17 = 1.5 \times 10^{-16}.$$

Since $\delta_1 < \delta_2$, v_1 is considered the *decimal native* representation of 0.17, as it more closely approximates the original decimal value.

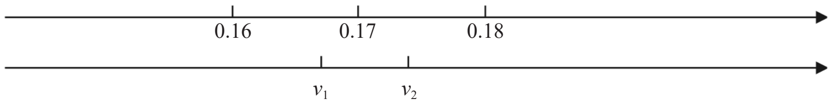

Figure 9. The positions of these numbers on the number axis. The number axis above represents decimal decimals, while the number axis below represents binary decimals

5.2 *Determining decimal native numbers*

The current challenge is to ascertain whether a given floating-point number is decimal native and to identify its corresponding decimal representation. That is to say, given a floating-point number x, if there exists a decimal number y such that x is the number converted from y to a floating-point number, then the floating-point number x is decimal native, and y is the corresponding decimal number for x.

Given that integer-to-integer conversions between binary and decimal are error-free, we equate the integer parts of y and x. Subsequently, we focus solely on the fractional parts of x and y, where x and y now exclusively refer to their fractional components. According to Theorem 1, we consider the fractional part of x as b_0, and after the i-th operation, $x = (0.a_1a_2 \ldots a_i)_{10} + b_i \cdot 10^{-i}$. Consequently, we can assume y to be either $(0.a_1a_2 \ldots a_i)_{10}$ or $y = (0.a_1a_2 \ldots a_i)_{10} + 10^{-i}$. In the first assumption (i.e. $y = (0.a_1a_2 \ldots a_i)_{10}$), the absolute error is $\delta = x - y = b_i \cdot 10^{-i}$. According to Theorem 2, in order to satisfy $\delta < 2^{-(\alpha+1)}$, we have $\delta = b_i \cdot 10^{-i} < 2^{-(\alpha+1)}$, which is equivalent to:

$$b_i \cdot 2^{\alpha} < 2^{-1} \cdot 10^{i} \tag{4}$$

In the second assumption (i.e. $y = (0.a_1a_2 \ldots a_i)_{10} + 10^{-i}$), the absolute error is $\delta = y - x = (1 - b_i) \cdot 10^{-i}$. According to Theorem 2, in order to satisfy $\delta < 2^{-(\alpha+1)}$, we have $\delta = (1 - b_i) \cdot 10^{-i} < 2^{-(\alpha+1)}$, which is equivalent to:

$$\left(1-b_i\right) \cdot 2^{\alpha} < 2^{-1} \cdot 10^{i} \tag{5}$$

where b_i is the remaining binary fraction after i digits of conversion, and α is the number of bits in the fractional part of the floating-point format (52 for double precision).

Theorem 1 Given a binary fraction b_0, where $0 \le b_0 < 1$, multiply b_{i-1} by $(1010)_2$ ($i >= 1$) each time. After the i-th operation, set the integer part of the result to a_i (decimal) and the decimal part to b_i (binary). Then, after the i-th operation, $b_0 = (0.a_1a_2 \ldots a_i)_{10} + b_i \cdot 10^{-i}$.

Proof. Due to $0 \le b_i < 1$, $b_i \cdot (1010)_2 = b_i \cdot (10)_{10} < 10$, therefore ai belongs to 0-9. When $i = 1$, the integer part of the result of $b_0 \cdot (1010)_2$ is a_1 (decimal), and the decimal part is b_1 (binary). There is $b_0 \cdot (1010)_2 = a_1 + b_1$, so $b_0 = a_1/10 + b_1/10 = (0.a_1)_{10} + b_i \cdot 10^{-1}$, which holds true. Assuming that for the i-th calculation, $b_0 = (0.a_1a_2 \ldots a_i)_{10} + b_i \cdot 10^{-i}$, then for the $i + 1$st calculation, $b_i \cdot (1010)_2 = a_{i+1} + b_{i+1}$, $b_i = (0.a_{i+1})_{10} + b_{i+1} \cdot 10^{-1}$, so $b_0 = (0.a_1a_2 \ldots a_i)_{10} + b_i \cdot 10^{-i} = (0.a_1a_2 \ldots a_i)_{10} + (0.a_{i+1})_{10} + b_{i+1} \cdot 10^{-1}) \cdot 10^{-i} = (0.a_1a_2 \ldots a_ia_{i+1})_{10} + b_{i+1} \cdot 10^{-i+1}$, it is proven. □

Theorem 2 Given a binary number x and a decimal number y, where $x = (\ldots x_2x_1x_0.x_{-1}x_{-2} \ldots x_{-\alpha})_2$, if $\delta = |y - x| < 2^{-(\alpha+1)}$, then x is the number where y is converted to binary.

Proof. For all binary numbers, which are in the form of $x = (\ldots x_2x_1x_0.x_{-1}x_{-2} \ldots x_{-\alpha})_2$, the difference between adjacent numbers is $2^{-\alpha}$, and the position of y is in the middle of some two binary numbers. Therefore, when $\delta = |y - x| < 1/2 \cdot 2^{-\alpha} = 2^{-(\alpha+1)}$, x is the number closest to y among these binary numbers, so x is the number where y is converted into binary. □

Taking the binary fractions v_1 and v_2 in Section 4.1 as an example, we perform Algorithm 7 on v_1 and v_2, and the results are shown in Table 1. We first analyze v_1. Following the first operation, where $a_1 = 1$, we initially set $y = 0.1$, but this choice does not meet the requirement $b_1 \cdot 2^{\alpha} > 2^{-1} \cdot 10^{1} = (5)_{10}$, so $y = 0.1$ is discarded. Next, we consider $y = 0.2$, but this choice does not satisfy the condition $(1 - b_1) \cdot 2^{\alpha} > 2^{-1} \cdot 10^1 = (5)_{10}$, so $y = 0.2$ is also discarded. After the second operation, with $a_2 = 6$, we attempt $y = 0.16$ first, but this does not meet the condition $b_2 \cdot 2^{\alpha} > 2^{-1} \cdot 10^2 = (50)_{10}$, so $y = 0.16$ is discarded. However, setting $y = 0.16 + 10^{-2} = 0.17$ satisfies the condition $(1 - b_2) \cdot 2^{\alpha} = (11111)_2 < 2^{-1} \cdot 102 = (50)_{10}$, indicating that $y = 0.17$ is valid. Consequently, v_1 is decimal native, and 0.17 is its corresponding decimal number. For v_2, it can be observed that no choice of y satisfies the condition for i from 1 to 15, suggesting that v_2 is not decimal native.

Algorithm 8 describes the aforementioned process and serves as a replacement for Algorithm 7. For a double-precision floating-point number, Algorithm 7 requires 16 iterations to determine the corresponding decimal representation. In contrast, Algorithm 8 typically completes the task with significantly fewer iterations—often fewer than 16 (e.g., 2 iterations for 0.17 and 3 for 0.172)—while still ensuring the reliability of the decimal conversion. Consequently, Algorithm 8 is more efficient than Algorithm 7. Moreover, Algorithm 8 can determine whether a number is decimal-native; it produces the corresponding decimal number only if the number is indeed decimal-native, whereas Algorithm 7 lacks the ability to identify decimal native numbers.

Algorithm 8 *OptimizedFractionToDecimal* (*fraction*, *decimal*)

Input: *fraction* – binary fraction, *decimal* – output array

Output: Returns true if decimal-native, otherwise false

```
fraction ← fraction >> 4;
i ← 1;
while fraction ! = 0 AND i ≤ 16 do
    fraction ← fraction * 0b1010;
    decimal[i] ← fraction >> 60;
    fraction ← fraction & 0x0fffffffffffffffL;
    b_i ← fraction >> (60 − α);
    if b_i < 10^i/2 then
        return true;
    end
    if ((1 << α) − b_i) < 10^i /2 then
        decimal[i] ← decimal[i] + 1;
        return true;
    end
    i ← i + 1;
end
if i == 17 then
    return false;
end
```

6. Experimental evaluation

We have implemented our *Pongo* compression algorithm in Java and have compared its performance against ten other compression methods for time-series floating-point data. Our testing code is based on the Elf experimental code, with the addition of *Pongo* code implementation and testing. We will delve into the experimental setup and present our conclusions in the subsequent sections.

6.1 *Experimental setting*

6.1.1 *Experimental operating environment*

Our experiment was conducted on a Windows 11-based computer equipped with an Intel ® Core™ i7-10700 CPU operating at 2.90 GHz and 32 GB of memory. The Java Development Kit (JDK) version 1.8 was used as the runtime environment.

6.1.2 *Datasets*

To validate the performance of the *Pongo* algorithm, we tested a total of 22 datasets, which include 14 time-series datasets and 8 non-time-series datasets. These datasets were previously used in the Elf experiment [9]. Among them, the data in the time-series datasets are sorted by timestamp, while the data in the non-time-series datasets is sorted by the default order given by the publisher. The datasets we use are listed in Table 2.

Table 2. Dataset information

	Dataset	Recoreds	Decimal digit	Time span
Time series	City-Temp (CT)	2,905,887	1	25 years
	Wind-Speed (WS)	199,570,396	2	6 years
	IR-bio-temp (IR)	380,817,839	2	7 years
	PM10-dust (PM10)	222,911	3	5 years
	Dewpoint-Temp (DT)	5,413,914	3	3 years
	Air-Pressure (AP)	137,721,453	5	6 years
	Stocks-UK (SUK)	115,146,731	1	1 years
	Stocks-USA (SUSA)	374,428,996	2	1 years
	Stocks-DE (SDE)	45,403,710	3	1 years
	Bird-Migration (BM)	17,964	5	1 years
	Bitcoin-Price (BP)	2,741	4	1 month
	Air-Sensor (AS)	8,664	17	1 hour
	Basel-Wind (BW)	124,079	7	14 years
	Basel-Temp (BT)	124,079	9	14 years
Non time series	Food-Prices (FP)	2,050,638	4	-
	Vehicle-Charge (VC)	3,395	2	-
	SD-Bench (SB)]	8,927	1	-
	Blockchain-tr (BTR)	231,031	4	-
	City-Lat (CLat)	41,001	4	-
	City-Lon (CLon)	41,001	4	-
	POI-Lat (PLat)	424,205	16	-
	POI-Lon (PLon)	424,205	16	-

6.1.3 *Baselines*

We compare the performance of the *Pongo* algorithm against five state-of-the-art lossless floating-point compression algorithms and five general-purpose compression algorithms. These five lossless floating-point compression algorithms are Gorilla [7], Chimp [8], Chimp128 [8], FPC [24] and Elf [9]. These five general compression algorithms are Xz [5], Brotli [31], LZ4 [4], Zstd [19], and Snappy [32].

6.1.4 *Metrics*

We assess the performance of the *Pongo* algorithm using three metrics: compression ratio, compression time, and decompression time. The compression ratio is defined as the ratio of the compressed data size to the original data size.

6.2 *Experimental result*

The results of the experiment are presented in Table 3. Next, we analyze the results in detail.

6.2.1 *Compression ratio*

We draw conclusions by analyzing the compression ratio performance of 11 compression algorithms across 22 datasets. The formula for calculating the compression ratio is:

$$\frac{\text{Total Size}_{\text{Compressed}}}{\text{Total Size}_{\text{uncompressed}}} \tag{6}$$

Prior to *Pongo*, Elf was nearly the algorithm with the best compression ratio among all tested algorithms. Upon examining the data in Table 3, we observe that *Pongo* achieves a superior compression ratio compared to Elf. On time series datasets, *Pongo* outperforms Elf by 14%, Gorilla by 58%, and Chimp_{128} by 24%. For non-time series datasets, *Pongo* surpasses Elf by 9%, Gorilla by 43%, and Chimp_{128} by 21%. With the help of low-decimal precision floating-point data, and large number of trailing zeros, *Pongo* can achieve such excellent results.

Comparing *Pongo* with general compression algorithms, we observe that for the first time, *Pongo* surpasses the Xz general compression algorithm as a streaming compression algorithm. On time series datasets, the average compression ratio for Xz is 0.33, Elf is 0.37, and *Pongo* is 0.32. For non-time series datasets, the average compression ratio for Xz is 0.51, Elf is 0.55, and *Pongo* is 0.50. Regardless of whether the dataset is time series or non-time series, *Pongo* consistently outperforms Xz, suggesting that *Pongo* possesses the highest average compression ratio among the 11 tested algorithms.

We then analyzed several datasets with suboptimal *Pongo* performance (i.e., AS, BT, PLat, and PLon), and discovered that these datasets contain data with significant decimal places. Specifically, AS has a decimal place of 17, BT has a decimal place of 9, and both PLat and PLon have a decimal place of 16. This analysis suggests that as the number of decimal places in a dataset increases, the compression ratio of *Pongo* tends to decrease. In fact, analyzing the internal details of the *Pongo* algorithm can better understand this point. As the number of decimal places in the data increases, the fractional parts that are converted to integers also increase. For instance, in BT, the data 2.6105285, which has a decimal place of 7, is converted to the integer $(6105285)_{10} = (10111010010101000110000101)_2$, which occupies 23 bits. The larger the decimal place of the dataset, the more bits it takes to convert the fractional part to an integer, and the fewer trailing zeros, resulting in poorer compression ratio.

6.2.2 *Compression time*

Table 3 shows the average time (μs) required to compress 1,000 values across all 22 datasets in our experiment, which is the average of multiple executions. From the compression time data in Table 3, we can compute the compression rate of the *Pongo* algorithm as $r = 1{,}000 \times 64/(118 \times 10^{-6}) \approx 5.42 \times 10^{8}$ bits/second. When comparing *Pongo* with Elf, we find that *Pongo*'s compression time is comparable on most datasets but is less efficient on a few (AS, BT, PLat, and PLon). Specifically, *Pongo* exhibits the poorest performance on the AS dataset, with an average compression time that is approximately six times that of Elf. We note that the compression time performance of *Pongo* is particularly affected by datasets with a high number of decimal places.

Table 3. Experimental result

		Dataset	Time series															Non time series								
			CT	WS	IR	PM10	DT	AP	SUK	SUSA	SDE	BM	BP	AS	BW	BT	AVG.	FP	VC	SB	BTR	CLat	CLon	PLat	PLon	AVG.
Compression Ratio	Floating	Gorilla	0.85	0.83	0.64	0.48	0.83	0.73	0.58	0.68	0.72	0.79	0.84	0.82	0.99	0.94	0.76	0.58	1.00	0.63	0.74	1.03	1.03	1.03	1.03	0.88
		Chimp	0.64	0.81	0.59	0.46	0.77	0.65	0.52	0.64	0.67	0.72	0.77	**0.77**	0.88	0.85	0.70	0.47	0.86	0.55	0.67	0.92	0.98	**0.90**	**0.99**	0.79
		Chimp128	0.32	0.23	0.24	0.21	0.35	0.54	0.29	0.23	0.27	0.50	0.72	**0.77**	0.71	**0.47**	0.42	0.34	0.36	0.27	0.55	0.78	0.85	**0.90**	**0.99**	0.63
		FPC	0.75	0.85	0.61	0.50	0.82	0.67	0.74	0.70	0.73	0.75	0.81	0.82	0.92	0.90	0.75	0.62	0.91	0.59	0.69	0.96	1.00	0.95	1.00	0.84
		Elf	0.25	0.25	0.21	**0.16**	0.31	0.31	0.22	0.24	0.26	0.42	0.56	0.85	0.59	0.58	0.37	0.23	0.34	0.27	0.36	0.56	0.63	0.96	1.06	0.55
		Pongo	**0.21**	**0.22**	**0.14**	0.17	**0.24**	**0.27**	**0.19**	**0.17**	**0.21**	**0.32**	**0.46**	0.87	**0.52**	0.49	**0.32**	**0.20**	**0.27**	**0.22**	**0.28**	**0.47**	**0.54**	0.97	1.07	**0.50**
	General	Xz	**0.18**	**0.15**	**0.16**	**0.11**	**0.27**	**0.47**	**0.16**	**0.17**	**0.19**	**0.43**	**0.63**	**0.79**	**0.57**	**0.35**	**0.33**	**0.23**	**0.23**	**0.13**	**0.40**	**0.60**	**0.63**	**0.93**	**0.96**	**0.51**
		Brotli	0.20	0.17	0.18	0.12	0.32	0.51	0.19	0.20	0.22	0.47	0.71	0.85	0.61	0.39	0.39	0.26	0.28	0.14	0.43	0.65	0.68	0.94	**0.96**	0.54
		LZ4	0.36	0.37	0.36	0.27	0.52	0.69	0.39	0.39	0.41	0.61	0.87	1.01	0.69	0.54	0.56	0.41	0.47	0.30	0.53	0.79	0.82	1.00	1.00	0.67
		Zstd	0.22	0.19	0.24	0.14	0.38	0.58	0.22	0.24	0.26	0.51	0.75	0.91	0.61	0.41	0.43	0.30	0.34	0.17	0.45	0.68	0.71	0.94	**0.96**	0.57
		Snappy	0.29	0.27	0.30	0.21	0.51	0.73	0.32	0.32	0.35	0.61	0.99	1.00	0.75	0.54	0.55	0.39	0.42	0.25	0.54	0.83	0.87	1.00	1.00	0.66
Compression Time	Floating	Gorilla	**21**	**20**	**20**	**16**	**19**	**73**	**18**	**18**	**19**	**19**	**20**	**34**	**25**	**26**	**25**	**17**	**23**	**18**	**19**	**23**	**22**	**23**	**23**	**21**
		Chimp	29	27	24	21	28	47	25	25	26	27	28	46	31	32	30	24	29	23	25	29	29	27	31	27
		Chimp128	31	27	27	25	31	65	30	26	28	34	38	79	43	43	38	30	32	29	35	42	42	37	43	36
		FPC	49	47	39	43	38	123	33	34	34	37	33	82	62	84	51	43	46	43	48	50	51	46	57	48
		Elf	56	63	53	54	60	114	58	60	63	70	65	128	77	84	71	53	58	51	69	69	73	77	104	69
		Pongo	48	56	48	56	59	161	49	53	60	79	83	688	117	127	118	47	58	45	74	77	78	159	186	91
	General	Xz	1,296	1,006	1,034	891	1,123	2,820	913	923	980	1,202	1,454	2,097	1,257	1,589	1,336	1,026	1,334	891	1,223	1,296	1,278	1,516	1,721	1,286
		Brotli	2,136	1,977	1,984	1,815	2,078	5,689	2,022	1,999	2,026	2,180	2,084	2,613	2,648	2,600	2,396	2,029	2,064	1,907	2,082	2,136	2,120	1,984	2,180	2,063
		LZ4	1,339	1,252	1,273	1,291	1,305	2,271	1,257	1,263	1,256	1,314	1,288	1,437	1,469	1,458	1,384	1,326	1,315	1,308	1,407	1,339	1,281	1,336	1,322	1,329
		Zstd	345	**108**	222	290	222	257	231	230	243	274	267	209	325	357	298	223	245	212	271	345	251	214	237	250
		Snappy	**209**	182	**203**	**233**	**207**	**221**	**179**	**176**	**188**	**203**	**189**	**138**	**216**	**224**	**198**	**190**	**197**	**193**	**211**	**210**	**225**	**187**	**191**	**201**
Decompression Time	Floating	Gorilla	**22**	21	21	19	**21**	51	**19**	21	**21**	**22**	**21**	53	**23**	**25**	**25**	**19**	24	21	**21**	**21**	**21**	**22**	**24**	**22**
		Chimp	26	55	23	21	26	**44**	23	25	25	26	26	57	29	32	31	22	28	22	27	27	27	27	31	26
		Chimp128	23	**19**	**19**	**18**	23	46	21	**19**	21	28	29	43	32	28	27	22	**23**	**20**	27	31	31	29	34	27
		FPC	32	29	28	31	28	47	25	26	26	30	24	48	34	48	32	33	36	30	38	34	42	35	39	36
		Elf	44	50	45	50	51	61	40	50	49	53	55	**35**	60	66	51	35	48	43	53	64	62	34	42	48
		Pongo	54	64	63	69	73	130	56	66	73	96	105	267	126	121	98	49	84	55	103	103	109	80	97	85
	General	Xz	147	120	139	103	215	467	158	135	147	326	454	604	422	274	265	190	181	119	292	435	438	652	639	368
		Brotli	60	42	57	57	73	125	46	48	52	164	87	107	96	82	78	64	69	58	83	85	86	117	74	80
		LZ4	25	**19**	**26**	**27**	**26**	**35**	**20**	**22**	**19**	**31**	29	32	**34**	**29**	**27**	**27**	**26**	**26**	**31**	**25**	**26**	**24**	**18**	**25**
		Zstd	40	30	41	34	40	58	32	34	34	45	42	46	52	159	49	71	41	38	51	41	42	39	31	44
		Snappy	32	21	29	28	31	41	41	**22**	21	34	**25**	**30**	39	82	34	31	31	28	33	97	29	25	22	37

6.2.3 *Decompression time*

The decompression time is defined as the average time needed to decompress 1,000 values, which is also the average across multiple executions. From this decompression time, we can calculate the decompression rate of the *Pongo* algorithm as $r = 1{,}000 \times 64/(121 \times 10^{-6}) \approx 5.29 \times 10^{8}$ bits/second. We observe that Gorilla and LZ4 exhibit the best decompression performance, while *Pongo*'s decompression time is approximately four times longer than theirs. When comparing *Pongo* with Elf, *Pongo* has an average decompression time that is 92% higher than Elf. Additionally, *Pongo*'s decompression performance on the AS dataset is notably poor, with a decompression time that is nearly five times that of Gorilla.

6.2.4 *Summary*

In conclusion, *Pongo* stands out for achieving the highest average compression ratio among algorithms for floatingpoint data. While its compression and decompression times are longer than those of other streaming floating-point compression algorithms, they remain significantly shorter than those of general-purpose compression algorithms. *Pongo* exhibits varied performance depending on dataset characteristics. For datasets with more than 9 decimal places, the additional flag bit overhead in the algorithm reduces *Pongo*'s compression ratio advantage. Furthermore, its compression and decompression times become noticeably longer than those of Elf, primarily due to the extra time required to convert decimal fractions to integers within the algorithm. Consequently, *Pongo* is best suited for datasets with fewer than 9 decimal places, offering the highest compression ratio within acceptable compression and decompression times. For datasets with more than 9 decimal places, *Pongo* still maintains a compression ratio advantage, and the increase in compression-decompression time remains within an acceptable range. To address this issue, we plan to carry out targeted algorithmic and data structure optimizations for high decimal-precision floating-point data, as well as develop parallelized versions of the algorithm to further reduce compression and decompression time costs.

7. Conclusion and future work

This paper introduced *Pongo*, a novel lossless compression algorithm designed specifically for floating-point numbers. To evaluate its performance, we conducted extensive experiments on 22 datasets covering a wide range of data types. The results demonstrated that *Pongo* achieves the highest average compression ratio among all tested lossless compression algorithms, including general purpose compressors. While *Pongo*'s compression time is longer than that of other streaming compression methods, it remains considerably faster than general-purpose algorithms. Overall, our findings indicate that *Pongo* provides a favorable trade-off between compression decompression time and compression ratio, particularly when processing high precision floating-point data.

In future work, we plan to further optimize the compression and decompression speed of the *Pongo* algorithm. Specifically, we are exploring parallel computation techniques and investigating specialized strategies for handling high-decimal precision floating-point numbers. These improvements aim to make *Pongo* more efficient and practical for real-time or large-scale data processing scenarios.

Conflict of interest

The authors declare no competing financial interest.

References

[1] Rokach L, Maimon O. *Data Mining and Knowledge Discovery Handbook*. 2nd ed. New York: Springer; 2010.
[2] Sharma AB, Ivančić F, Niculescu-Mizil A, Chen H, Jiang G. Modeling and analytics for cyber-physical systems in

the age of big data. *ACM SIGMETRICS Performance Evaluation Review*. 2014; 41(4): 74-77.

[3] Ronkainen J, Iivari A. Designing a data management pipeline for pervasive sensor communication systems. *Procedia Computer Science*. 2015; 56: 183-188.

[4] Yann C. *LZ4 - Extremely Fast Compression.* 2011. Available from: https://lz4.github.io/lz4/ [Accessed 28th June 2025].

[5] *The .xz File Format*. 2009. Available from: https://tukaani.org/xz/xz-file-format.txt [Accessed 28th June 2025].

[6] Jensen SK, Pedersen TB, Thomsen C. Time series management systems: a survey. *IEEE Transactions on Knowledge and Data Engineering*. 2017; 29(11): 2581-2600.

[7] Pelkonen T, Franklin S, Teller J, Cavallaro P, Huang Q, Meza J, et al. Gorilla: a fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*. 2015; 8(12): 1816-1827.

[8] Liakos P, Papakonstantinopoulou K, Kotidis Y. Chimp: efficient lossless floating point compression for time series databases. *Proceedings of the VLDB Endowment*. 2022; 15(11): 3058-3070.

[9] Li R, Li Z, Wu Y, Chen C, Zheng Y. Elf: erasing-based lossless floating-point compression. *Proceedings of the VLDB Endowment*. 2023; 16(7): 1763-1776.

[10] Lindstrom P. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics*. 2014; 20(12): 2674-2683.

[11] Lazaridis I, Mehrotra S. Capturing sensor-generated time series with quality guarantees. In: *Proceedings of the 19th International Conference on Data Engineering (Cat. No. 03CH37405)*. Bangalore, India: IEEE; 2003. p.429-440.

[12] Liu C, Jiang H, Paparrizos J, Elmore AJ. Decomposed bounded floats for fast compression and queries. *Proceedings of the VLDB Endowment*. 2021; 14(11): 2586-2598.

[13] Elmeleegy H, Elmagarmid A, Cecchet E, Aref WG, Zwaenepoel W. Online piecewise linear approximation of numerical streams with precision guarantees. In: *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB)*. Lyon, France; 2009. p.145-156.

[14] Lin J, Keogh E, Lonardi S, Chiu B. A symbolic representation of time series, with implications for streaming algorithms. In: *Proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD '03)*. New York, NY, USA: Association for Computing Machinery; 2003. p.2-11.

[15] Liu T, Wang J, Liu Q, Alibhai S, Lu T, He X. High-ratio lossy compression: exploring the autoencoder to compress scientific data. *IEEE Transactions on Big Data*. 2023; 9(1): 22-36.

[16] Zhao K, Di S, Dmitriev M, Tonellot TLD, Chen Z, Cappello F. Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation. In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. Chania, Greece: IEEE; 2021. p.1643-1654.

[17] Zhao K, Di S, Perez D, Liang X, Chen Z, Cappello F. MDZ: an efficient error-bounded lossy compressor for molecular dynamics. In: *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. Kuala Lumpur, Malaysia: IEEE; 2022. p.27-40.

[18] Sun G, Jun SW. ZFP-V: Hardware-optimized lossy floating point compression. In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. Tianjin, China: IEEE; 2019. p.117-125.

[19] Collet Y. *Zstd GitHub Repository from Facebook*. 2016. Available from: https://github.com/facebook/zstd [Accessed 28th June 2025].

[20] Sazeides Y, Smith JE. The predictability of data values. In: *Proceedings of the 30th Annual International Symposium on Microarchitecture*. IEEE; 1997. p.248-258.

[21] Goeman B, Vandierendonck H, de Bosschere K. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In: *Proceedings of the HPCA Seventh International Symposium on High-Performance Computer Architecture*. Monterrey, Mexico: IEEE; 2001. p.207-216.

[22] Ibarria L, Lindstrom P, Rossignac J, Szymczak A. Out-of-core compression and decompression of large n-dimensional scalar fields. *Computer Graphics Forum*. 2003; 22(3): 343-348.

[23] Ratanaworabhan P, Ke J, Burtscher M. Fast lossless compression of scientific floating-point data. In: *Data Compression Conference (DCC '06)*. Snowbird, UT, USA: IEEE; 2006. p.133-142.

[24] Burtscher M, Ratanaworabhan P. FPC: a high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers*. 2009; 58(1): 18-31.

[25] Lindstrom P, Isenburg M. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*. 2006; 12(5): 1245-1550.

[26] Burtscher M, Ratanaworabhan P. High throughput compression of double-precision floating-point data. In: *2007 Data Compression Conference (DCC '07)*. Snowbird, UT, USA: IEEE; 2007. p.293-302.

[27] Chen H, Liu L, Meng J, Lu W. AFC: an adaptive lossless floating-point compression algorithm in time series database. *Information Sciences*. 2024; 654: 119847.
[28] Wang W, Chen W, Lei Q, Li Z, Zhao H. ACTF: an efficient lossless compression algorithm for time series floating point data. *Journal of King Saud University - Computer and Information Sciences*. 2024; 36(10): 102246.
[29] Afroozeh A, Kuffo LX, Boncz P. ALP: Adaptive lossless floating-point compression. *Proceedings of the ACM on Management of Data*. 2023; 1(4): 1-26.
[30] Kahan W. IEEE standard 754 for binary floating-point arithmetic. In: *Lecture Notes on the Status of IEEE 754*. 1996. p.1-30.
[31] Alakuijala J, Farruggia A, Ferragina P, Kliuchnikov E, Obryk R, Szabadka Z, et al. Brotli: a general-purpose data compressor. *ACM Transactions on Information Systems (TOIS)*. 2018; 37(1): 1-30.
[32] Google. *Snappy: A Fast Compressor/Decompressor*. 2023. Available from: https://github.com/google/snappy [Accessed 28th June 2025].