Elf: Erasing-based Lossless Floating-Point Compression

Ruiyuan Li
Zheng Li
Chongqing University, China
ruiyuan.li@cqu.edu.cn
zhengli@cqu.edu.cn

Yi Wu
Chao Chen
Chongqing University, China
wu_yi@cqu.edu.cn
cschaochen@cqu.edu.cn

Yu Zheng
JD iCity, JD Technology, China
JD Intelligent Cities Research, China
Xidian University, China
msyuzheng@outlook.com

ABSTRACT

There are a prohibitively large number of floating-point time series data generated at an unprecedentedly high rate. An efficient, compact and lossless compression for time series data is of great importance for a wide range of scenarios. Most existing lossless floating-point compression methods are based on the XOR operation, but they do not fully exploit the trailing zeros, which usually results in an unsatisfactory compression ratio. This paper proposes an Erasing-based Lossless Floating-point compression algorithm, i.e., *Elf*. The main idea of *Elf* is to erase the last few bits (i.e., set them to zero) of floating-point values, so the XORed values are supposed to contain many trailing zeros. The challenges of the erasing-based method are three-fold. First, how to quickly determine the erased bits? Second, how to losslessly recover the original data from the erased ones? Third, how to compactly encode the erased data? Through rigorous mathematical analysis, *Elf* can directly determine the erased bits and restore the original values without losing any precision. To further improve the compression ratio, we propose a novel encoding strategy for the XORed values with many trailing zeros. *Elf* works in a streaming fashion. It takes only $O(N)$ (where N is the length of a time series) in time and $O(1)$ in space, and achieves a notable compression ratio with a theoretical guarantee. Extensive experiments using 22 datasets show the powerful performance of *Elf* compared with 9 advanced competitors.

PVLDB Reference Format:
Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. *Elf*: Erasing-based Lossless Floating-Point Compression. PVLDB, 16(7): 1763 - 1776, 2023.
doi:10.14778/3587136.3587149

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Spatio-Temporal-Lab/elf.

1 INTRODUCTION

The advance of sensing devices and Internet of Things [33, 45] has brought about the explosion of time series data. A significant portion of time series data are floating-point values produced at an unprecedentedly high rate in a streaming fashion. For example, there are over ten thousand sensors in a 600,000-kilowatt medium-sized thermal power generating unit, which produce tens of thousands of real-time monitoring floating-point records per second [52, 53]. Additionally, the sensors on a Boeing 787 can generate up to half a terabyte of data per flight [23]. If these huge floating-point time series data (abbr. time series or time series data in the following) are transmitted and stored in their original format, it would take up a lot of network bandwidth and storage space, which not only causes expensive overhead, but also reduces the system efficiency [30, 31] and further affects the usability of some critical applications [53].

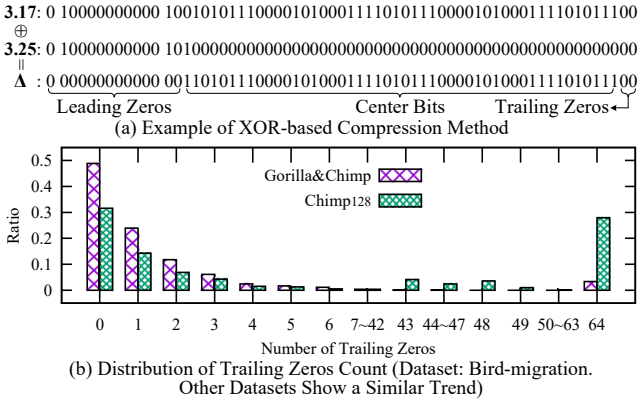
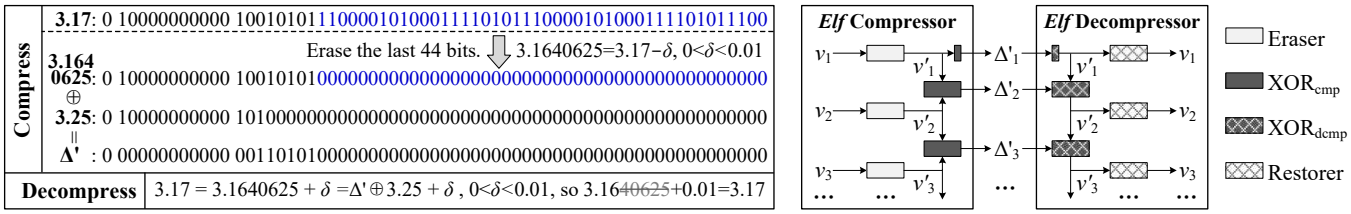


Figure 1: Motivation.

One of the best ways is to compress the time series data before transmission and storage. However, it is challenging for the compression of floating-point data, because they have a rather complex underlying format [26]. General compression algorithms such as LZ4 [19] and Xz [12] do not exploit the intrinsic characteristics (e.g., time ordering) of time-series data, although they could achieve good compression ratio, but they are prohibitively time-consuming. Moreover, most of them run in a batch mode, so they cannot be applied directly to streaming time series data. There are two categories of compression methods specifically for floating-point time series data, i.e., lossy compression algorithms and lossless compression algorithms. The former [27, 35–38, 54, 55] would lose some information, and thus it is not suitable for scientific calculation, data management [22, 29, 32, 50, 52] or other critical scenarios [53]. Imagine the scenes of thermal power generation [53] and flight [23], any error could result in disastrous consequences. To this end, lossless floating-point time series compression has attracted extensive interest for decades. One representative lossless algorithm is based on the XOR operation. As shown in Figure 1(a), given a time series of double-precision floating-point values, suppose the current value and its previous one are 3.17 and 3.25, respectively. If not compressed, each value will occupy 64 bits in its underlying storage (detailed in Section 2.2). When compressing, the XOR-based compression algorithm performs an XOR operation on 3.17 and 3.25, i.e., $\Delta = 3.17 \oplus 3.25$. When decompressing, it recovers 3.17

*Corresponding author: Chao Chen

Proceedings of the VLDB Endowment, Vol. 16, No. 7 ISSN 2150-8097.
doi:10.14778/3587136.3587149



(a) Intuition of Erasing-based Lossless Floating-Point Compression (*Elf*)

(b) Overview of *Elf* Compression for Time Series

Figure 2: Main Idea of *Elf* Compression.

through another XOR operation, i.e., $3.17 = \Delta \oplus 3.25$. Because two consecutive values in a time series tend to be similar, the underlying representation of Δ is supposed to contain many **leading zeros** (and maybe many **trailing zeros**). Therefore, we can record Δ by storing the **center bits** along with the numbers of leading zeros and trailing zeros, which usually takes up less than 64 bits.

Gorilla [46] and Chimp [34] are two state-of-the-art XOR-based lossless floating-point compression methods. Gorilla assumes that the XORed result of two consecutive floating-point values is likely to have both many leading zeros and trailing zeros. However, the XORed result actually has very few trailing zeros in most cases. As shown in Figure 1(b), if we perform an XOR operation on each value with its previous one (just as Gorilla and Chimp did), there are as many as 95% XORed results containing no more than 5 trailing zeros. Instead of using the exactly previous one value, Chimp_{128} [34] selects from the previous 128 values the one that produces an XORed result with the most trailing zeros. As a result, Chimp_{128} can achieve a significant improvement in terms of compression ratio. The lesson we can learn from Chimp_{128} is that, increasing the number of trailing zeros of the XORed results plays a significant role in improving the compression ratio for time series. However, as shown in Figure 1(b), when we investigate the trailing zeros' distribution of the XORed results produced by Chimp_{128}, there are still up to 60% of them having no more than 5 trailing zeros.

This paper proposes an Erasing-based Lossless Floating-point compression algorithm, i.e., *Elf*. The intuition of *Elf* is simple: if we erase last few bits (i.e., set them to zero) of the floating-point values, we can obtain an XORed result with a large number of trailing zeros. As shown in Figure 2(a), if we erase the last 44 bits of 3.17, we can transform it to 3.1640625. By XORing 3.1640625 with the previous value 3.25 (itself already has a lot of trailing zeros), we can get an XORed result Δ', which contains as many as 44 trailing zeros (only 2 before erasing as shown in Figure 1(a)).

There are three challenges for *Elf*. First, how to quickly determine the erased bits? Since there are a prohibitively large number of time series data generated at an unprecedented speed, it requires the erasing step to be as fast as possible. Second, how to losslessly restore the original floating-point data? This paper aims at lossless compression, but the erasing step would introduce some precision loss. It needs a restoring step to recover the original values from the erased ones. Third, how to compactly compress the erased floating point data? Since the distribution of trailing zeros has changed, it calls for a new XOR-based compressor for the erased values.

Figure 2(a) shows the main idea of *Elf*. For this example, during the compressing process, we find a small value δ satisfying $0 < \delta < 0.01$ to erase the bits of 3.17 as many as possible. Therefore, we can obtain an erased value $3.1640625 = 3.17 - \delta$, and encode the XORed result $\Delta' = 3.1640624 \oplus 3.25$ using few bits. During the decompressing process, since we know $3.1640624 = \Delta' \oplus 3.25 = 3.17 - \delta$ and $0 < \delta < 0.01$, we can losslessly recover 3.17 from Δ' and 3.25 (i.e., $3.16\cancel{40625} + 0.01 = 3.17$). This paper proposes a mathematical method to find δ in a time complexity of $O(1)$. Furthermore, we propose a novel XOR-based compressor to encode the XORed results containing many trailing zeros. As shown in Figure 2(b), *Elf* consists of Compressor and Decompressor, and works in a streaming fashion. In *Elf* Compressor, the original floating-point values v_i flow into *Elf* Eraser and are transformed into v'_i with many trailing zeros. Each v'_i (except for v'_1) is XORed with its previous value v'_{i-1}. The XORed result $\Delta'_i = v'_i \oplus v'_{i-1}$ is finally encoded elaborately in *Elf* XOR_{cmp}. In *Elf* Decompressor, each Δ'_i (except for Δ'_1) is streamed into *ELf* XOR_{dcmp} and then XORed with v'_{i-1}. Each $v'_i = \Delta'_i \oplus v'_{i-1}$ is finally fed into *Elf* Restorer to get the original value v_i.

To the best of our knowledge, this is the first proposal for lossless floating-point compression based on the erasing strategy. In particular, we make the following contributions:

(1) We propose an erasing-based lossless floating-point compression algorithm named *Elf*. *Elf* can greatly increase the number of trailing zeros in XORed results by erasing the last few bits, which enhances the compression ratio with a theoretical guarantee.

(2) Through rigorous theoretical analysis, we can quickly determine the erased bits, and recover the original floating-point values without any precision loss. *Elf* takes only $O(N)$ in time (where N is the length of a time series) and $O(1)$ in space.

(3) We also propose an elaborated encoding strategy for the XORed results with many trailing zeros, which further improves the compression performance.

(4) We compare *Elf* with 9 state-of-the-art competitors (including 4 floating-point compression algorithms and 5 general compression algorithms) based on 22 datasets. The results show that *Elf* has the best compression ratio among all floating-point compression algorithms in most cases (achieve an average relative improvement of 12.4% over Chimp_{128} and 43.9% over Gorilla). *Elf* even outperforms most general compression algorithms, and achieves similar performance to the best general one (i.e., Xz) in terms of compression ratio. However, *Elf* takes only about 4.86% compression time and 13.17% decompression time of Xz.

In the rest of this paper, we give the preliminaries in Section 2. In Section 3, we present the details of Eraser and Restorer. In Section 4, we elaborate on XOR_{cmp} and XOR_{dcmp}. We give some analysis and discussion in Section 5. The experimental results are shown in Section 6, followed by the related works in Section 7. We conclude this paper with future works in Section 8.

Table 1: Symbols and Their Meanings

Symbols	Meanings
$TS = \langle (t_1, v_1), (t_2, v_2), ... \rangle$	Floating-point time series, where t_i is a timestamp and v_i is a floating-point value
v, v'	Original floating-point value, erased floating-point value with long trailing zeros
$DF(v) = \pm(d_{h-1}d_{h-2}...d_0.d_{-1}d_{-2}...d_l)_{10}$	Decimal format of v, where $d_i \in \{1, 2, ..., 9\}$. "+" is usually omitted if $v > 0$
$BF(v) = \pm(b_{\bar{h}-1}b_{\bar{h}-2}...b_0.b_{-1}b_{-2}...b_{\bar{l}})_2$	Binary format of v, where $b_i \in \{1, 2\}$. "+" is usually omitted if $v > 0$
$DP(v)$, $DS(v)$, $SP(v)$	Decimal place count, decimal significand count, start decimal significand position of v
s, $\vec{e} = \langle e_1, e_2, ..., e_{11} \rangle$, $\vec{m} = \langle m_1, m_2, ..., m_{52} \rangle$	Sign bit, exponent bits, mantissa bits under IEEE 754 format, where $s, e_i, m_j \in \{0, 1\}$
e, α, β, β^*	Decimal value of $\vec{e}$, alias of $DP(v)$, alias of $DS(v)$, modified β

2 PRELIMINARIES

This section first gives some basic definitions, and then introduces the double-precision floating-point format of IEEE 754 Standard [26]. Table 1 lists the symbols used frequently throughout this paper.

2.1 Definitions

Definition 1. ***Floating-Point Time Series.*** *A floating-point time series $TS = \langle (t_1, v_1), (t_2, v_2), ... \rangle$ is a sequence of pairs ordered by the timestamps in an ascending order, where each pair (t_i, v_i) represents that the floating-point value v_i is recorded in timestamp t_i.*

To compress floating-point time series compactly, one of the best ways is to compress the timestamps and floating-point values separately [15, 34, 46]. For the timestamp compression, existing methods such as delta encoding and delta-of-delta encoding [46] can achieve rather good performance, but for the floating-point compression, there is still much room for improvement. To this end, this paper primarily focuses on the compression for floating-point values, particularly for double-precision floating-point values (abbr. **double values**) in time series. Single-precision floating-point compression can be easily extended from our proposed method.

Definition 2. ***Decimal Format and Binary Format.*** *The decimal format of v is $DF(v) = \pm(d_{h-1}d_{h-2}...d_0.d_{-1}d_{-2}...d_l)_{10}$, where $d_i \in \{0, 1, ..., 9\}$ for $l \le i \le h-1$, $d_{h-1} \neq 0$ unless $h = 1$, and $d_l \neq 0$ unless $l = -1$. That is, $DF(v)$ would not start with "0" except that $h = 1$, and would not end with "0" except that $l = -1$. Similarly, the binary format of v is $BF(v) = \pm(b_{\bar{h}-1}b_{\bar{h}-2}...b_0.b_{-1}b_{-2}...b_{\bar{l}})_2$, where $b_j \in \{0, 1\}$ for $\bar{l} \le j \le \bar{h}-1$. We have the following relation:*

$$v = \pm \sum_{i=l}^{h-1} d_i \times 10^i = \pm \sum_{j=\bar{l}}^{\bar{h}-1} b_j \times 2^j \tag{1}$$

Here, "±" (which means "+" or "−") is the sign of v. If $v \ge 0$, "+" is usually omitted. For example, $DF(0) = (0.0)_{10}$, $DF(5.20) = (5.2)_{10}$, and $BF(-3.125) = -(11.001)_2$.

Definition 3. ***Decimal Place Count, Decimal Significand Count and Start Decimal Significand Position.*** *Given v with its decimal format $DF(v) = \pm(d_{h-1}d_{h-2}...d_0.d_{-1}d_{-2}...d_l)_{10}$, $DP(v) = |l|$ is called its decimal place count. If for all $l < n \le i \le h-1$, $d_i = 0$ but $d_{n-1} \neq 0$ (i.e., d_{n-1} is the first digit that is not equal to 0), $SP(v) = n-1$ is called the start decimal significand position [1], and $DS(v) = n - l = SP(v) + 1 - l$ is called the decimal significand count. For the case of $v = 0$, we let $DS(v) = 0$ and $SP(v) = undefined$.*

[1]We have $SP(v) = \lfloor log_{10}|v| \rfloor$.

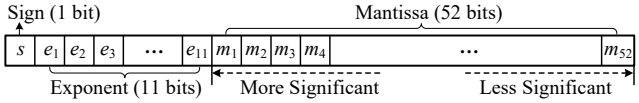


Figure 3: Double-Precision Floating-Point Format.

For example, $DP(3.14) = 2$, $DS(3.14) = 3$, and $SP(3.14) = 0$; $DP(-0.0314) = 4$, $DS(-0.0314) = 3$, and $SP(-0.0314) = -2$; $DP(314.0) = 1$, $DS(314.0) = 4$, and $SP(314.0) = 2$.

2.2 IEEE 754 Floating-Point Format

In accordance with IEEE 754 Standard [26], a double value v is stored with 64 binary bits , where 1 bit is for the sign s, 11 bits for the exponent $\vec{e} = \langle e_1, e_2, ..., e_{11} \rangle$, and 52 bits for the mantissa $\vec{m} = \langle m_1, m_2, ..., m_{52} \rangle$, as shown in Figure 3. When v is positive, $s = 0$, otherwise $s = 1$. According to the values of $\vec{e}$ and $\vec{m}$, a double value v can be categorized into two main types: **normal numbers** and **special numbers**. As normal numbers are the most cases of time series, this paper mainly describes the proposed algorithm for normal numbers. However, our proposed algorithm can be easily extended to special numbers, which will be discussed in Section 5.4. If v is a normal number (or a normal), its value satisfies:

$$\begin{aligned} v &= (-1)^s \times 2^{e-1023} \times (1.m_1m_2...m_{52})_2 \\ &= (-1)^s \times 2^{e-1023} \times (1 + \sum_{i=1}^{52} m_i \times 2^{-i}) \end{aligned} \tag{2}$$

where e is the decimal value of $\vec{e}$ [2], i.e., $e = \sum_{i=1}^{11} e_i \times 2^{11-i}$. If let $m_0 = 1$ and $BF(v) = (-1)^s(b_{\bar{h}-1}b_{\bar{h}-2}...b_0.b_{-1}b_{-2}...b_{\bar{l}})_2$, we have:

$$b_{-i} = m_{i+e-1023}, i > 0 \tag{3}$$

As shown in Figure 3, in the mantissa $\vec{m} = \langle m_1, m_2, ..., m_{52} \rangle$ of a double value v, m_i is more significant than m_j for $1 \le i < j \le 52$, since m_i contributes more to the value of v than m_j.

3 *ELF* ERASER AND RESTORER

In this section, we introduce *Elf* Eraser and Restorer since they are strongly correlated.

3.1 *Elf* Eraser

The main idea of *Elf* compression is to erase some less significant mantissa bits (i.e., set them to zeros) of a double value v. As a result, v itself and the XORed result of v with its previous value are expected to have many trailing zeros. Note that v and its opposite number

[2]We also have $e = \lfloor log_2|v| \rfloor + 1023$.

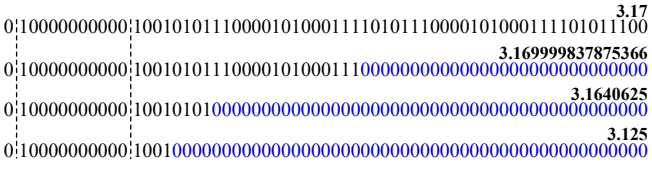


Figure 4: Examples of Mantissa Prefix Number.

$-v$ have the same double-precision floating-point formats except the different values of their signs. That is to say, the compression process for $-v$ can be converted into that for v if we reverse its sign bit only, and vice versa. To this end, in the rest of the paper, if not specified, we assume v to be **positive** for the convenience of description. Before introducing the details of *Elf* Eraser, we first give the definition of mantissa prefix number.

Definition 4. ***Mantissa Prefix Number.*** *Given a double value v with $\vec{m} = \langle m_1, m_2, ..., m_{52}\rangle$, the double value v' with $\vec{m'} = \langle m'_1, m'_2, ..., m'_{52}\rangle$ is called the mantissa prefix number of v if and only if there exists a number $n \in \{1, 2, ..., 51\}$ such that $m'_i = m_i$ for $1 \le i \le n$ and $m'_j = 0$ for $n+1 \le j \le 52$, denoted as $v' = MPN(v, n)$.*

For example, as shown in Figure 4, we give four mantissa prefix numbers of 3.17, i.e., $3.17 = MPN(3.17, 50)$, $3.169999837875366 = MPN(3.17, 23)$, $3.1640625 = MPN(3.17, 8)$ and $3.125 = MPN(3.17, 4)$.

3.1.1 Observation. Our proposed *Elf* compression algorithm is based on the following observation: given a double value v with its decimal format $DF(v) = (d_{h-1}d_{h-2}...d_0.d_{-1}d_{-2}...d_l)_{10}$, we can find one of its mantissa prefix numbers v' and a minor double value δ, $0 \le \delta < 10^l$, such that $v' = v - \delta$. If we retain the information of v' and δ, we can recover v without losing any precision.

On one hand, there could be many mantissa prefix numbers. Since we aim to maximize the number of trailing zeros of the XORed results, we should select the optimal mantissa prefix number that has the most trailing zeros. Considering the case of $v = 3.17$ shown in Figure 4, there are many satisfied pairs of (v', δ), e.g., $(3.17, 0)$, $(3.169999837875366, 0.000000162124634)$ and $(3.1640625, 0.0059375)$. As 3.1640625 has more trailing zeros than 3.169999837875366 and 3.17, the mantissa prefix number 3.1640625 is the most suitable v'.

On the other hand, we find it even unnecessary to figure out and store δ. If $\delta \neq 0$ (we will talk about the case when $\delta = 0$ in Section 3.1.4) and the decimal place count $DP(v)$ is known, we can easily recover v from v' losslessly. Suppose $\alpha = DP(v)$ and $DF(v') = (d_{h'-1}d_{h'-2}...d_0.d_{-1}d_{-2}...d_{l'})_{10}$, we have [3]:

$$v = LeaveOut(v', \alpha) + 10^{-\alpha} \tag{4}$$

where $LeaveOut(v', \alpha) = (d_{h'-1}d_{h'-2}...d_0.d_{-1}...d_{-\alpha}\cancel{d_{-(\alpha+1)}...d_{l'}})_{10}$ is the operation that leaves out the digits after $d_{-\alpha}$ in $DF(v')$. For example, given $\alpha = DP(3.17) = 2$ and $v' = 3.1640625$, we have $v = LeaveOut(v', \alpha) + 10^{-\alpha} = (3.16\cancel{40625})_{10} + 10^{-2} = 3.17$.

With the observation above, in the process of compression, what we should do is to find the most appropriate mantissa prefix number v' of v and record $\alpha = DP(v)$. During the decompression process, we can recover v losslessly with the help of v' and α according to Equation (4). However, there are still two problems left to be addressed. **Problem I**: How to find the best mantissa prefix number v' of v with the minimum efforts? **Problem II**: How to store the decimal place count α with the minimum storage cost?

[3]Equation (4) can be implemented by $v = RoundUp(v', \alpha)$, where $RoundUp(v', \alpha)$ is the operation to round v' up to α decimal places.

3.1.2 Mantissa Prefix Number Search. To address Problem I, one intuitive idea is to iteratively check all the mantissa prefix numbers $v' = MPN(v, i)$ until $\delta = v - v'$ is greater than $10^{-\alpha}$, where i is sequentially from 52 to 1. However, this intuitive idea is rather time-consuming since we need to verify the mantissa prefix numbers at most 52 times in the worst case. Although we can enhance the efficiency through a binary search strategy [14], the computation complexity $O(log_2 52)$ is still high. To this end, we propose a novel mantissa prefix number search method which only takes $O(1)$.

Theorem 1. *Given a double value v with its decimal place count $DP(v) = \alpha$ and binary format $BF(v) = (b_{\bar{h}-1}b_{\bar{h}-2}...b_0.b_{-1}b_{-2}...b_{\bar{l}})_2$, $\delta = (0.0...0b_{-(f(\alpha)+1)}b_{-(f(\alpha)+2)}...b_{\bar{l}})_2$ is smaller than $10^{-\alpha}$, where $f(\alpha) = \lceil |log_2 10^{-\alpha}| \rceil = \lceil \alpha \times log_2 10 \rceil$.*

Proof.
$$\begin{aligned}\delta &= \sum_{i=f(\alpha)+1}^{|\bar{l}|} b_i \times 2^{-i} \le \sum_{i=f(\alpha)+1}^{|\bar{l}|} 2^{-i} < \sum_{i=f(\alpha)+1}^{+\infty} 2^{-i} \\ &= 2^{-f(\alpha)} = 2^{-\lceil \alpha \times log_2 10 \rceil} \le 2^{-\alpha \times log_2 10} \\ &= (2^{log_2 10})^{-\alpha} = 10^{-\alpha}\end{aligned}$$ □

Here, $f(\alpha) = \lceil |log_2 10^{-\alpha}| \rceil$ means that the decimal value $10^{-\alpha}$ requires exactly $\lceil |log_2 10^{-\alpha}| \rceil$ binary bits to represent. Suppose δ is obtained based on Theorem 1, $v - \delta$ can be regarded as erasing the bits after $b_{-f(\alpha)}$ in v's binary format. Recall that for any b_{-i} in $BF(v)$ where $i > 0$, we can find a corresponding $m_{i+e-1023}$ according to Equation (3). Consequently, $v - \delta$ can be further deemed as erasing the mantissa bits after $m_{g(\alpha)}$ in v's underlying floating-point format, in which $g(\alpha)$ is defined as:

$$g(\alpha) = f(\alpha) + e - 1023 = \lceil \alpha \times log_2 10 \rceil + e - 1023 \tag{5}$$

where $\alpha = DP(v)$ and $e = (e_1 e_2 ... e_{11})_2 = \sum_{i=1}^{11} e_i \times 2^{11-i}$.

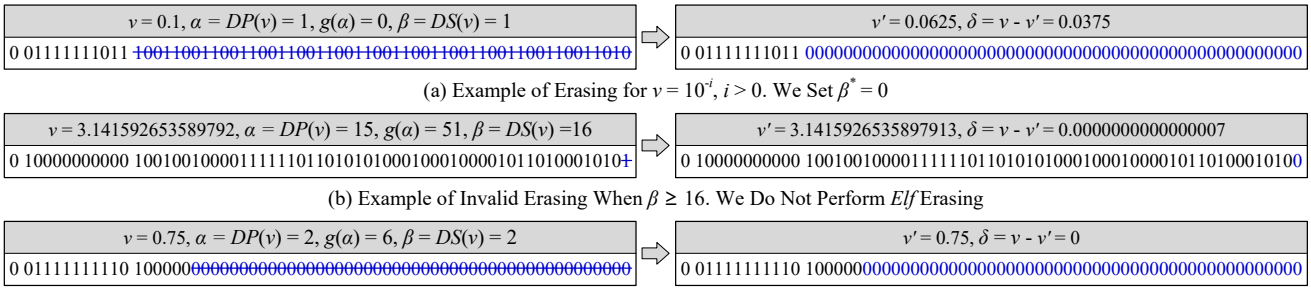
As a result, we can directly calculate the best mantissa prefix number v' by simply erasing the mantissa bits after $m_{g(\alpha)}$ of v, which takes only $O(1)$.

3.1.3 Decimal Place Count Calculation. To solve Problem II, the basic idea is to utilize $\lceil log_2 \alpha_{max} \rceil$ bits for α storage, where α_{max} is the possible maximum value of a decimal place count. According to [26], the minimum value of the double-precision floating-point number is about 4.9×10^{-324}, so $\alpha_{max} = 324$ and $\lceil log_2 \alpha_{max} \rceil = 9$, i.e., the basic method needs as many as 9 bits to store α during the compression process for each double value, which results in a large storage cost and low compression ratio.

Given a double value v with its decimal format $DF(v) = (d_{h-1}d_{h-2}...d_0.d_{-1}d_{-2}...d_l)_{10}$, we notice that its decimal place count $\alpha = DP(v)$ can be calculated by the decimal significand count $\beta = DS(v)$. Since the decimal significand count β of a double value would not be greater than 17 under the IEEE 754 Standard [26, 34], it requires much fewer bits to store β. According to Definition 3, we have $\alpha = DP(v) = |l| = -l$ and $\beta = DS(v) = SP(v) + 1 - l$, so we have:

$$\alpha = \beta - (SP(v) + 1) \tag{6}$$

Next, we discuss how to get $SP(v)$ without even knowing v.



(a) Example of Erasing for $v = 10^{-i}$, $i > 0$. We Set $\beta^* = 0$

(b) Example of Invalid Erasing When $\beta \geq 16$. We Do Not Perform *Elf* Erasing

(c) Example of Invalid Erasing When $\delta = 0$. We Do Not Perform *Elf* Erasing

Figure 5: Corner Cases of *Elf* Eraser.

$$
\begin{array}{ll}
 & DF(v):\ (d_{h\text{-}1}\ d_{h\text{-}2}\ \ldots\ d_0\ .\ d_{\text{-}1}\ d_{\text{-}2}\ \ldots\ d_{\text{-}(\alpha\text{-}1)}\ d_{\text{-}\alpha})_{10} \\
- & DF(\delta):\ (\qquad\qquad 0\ .\ 0\quad 0\quad \ldots\ 0\qquad 0\qquad d_{\text{-}(\alpha+1)}\ d_{\text{-}(\alpha+2)}\ldots)_{10} \\
\hline
 & DF(v'):\ (d'_{h\text{-}1}\ d'_{h\text{-}2}\ \ldots\ d'_0\ .\ d'_{\text{-}1}\ d'_{\text{-}2}\ \ldots\ d'_{\text{-}(\alpha\text{-}1)}\ (d_{\text{-}\alpha}\text{-}1)\ d'_{\text{-}(\alpha+1)}\ d'_{\text{-}(\alpha+2)}\ldots)_{10}
\end{array}
$$

Figure 6: Subtraction in Vertical Form.

Theorem 2. *Given a double value v and its best mantissa prefix number v', if $v \neq 10^{-i}$, $i > 0$, then $SP(v) = SP(v')$.*

Proof. Suppose $\alpha = DP(v)$ and $v' = v - \delta$, where $0 \leq \delta < 10^{-\alpha}$. If $\delta = 0$, i.e., $v = v'$, $DF(v)$ and $DF(v')$ undoubtedly have the same start decimal significand position.

If $\delta \neq 0$, we let $DF(v) = (d_{h-1}d_{h-2}...d_0.d_{-1}d_{-2}...d_{-\alpha})_{10}$, $DF(\delta) = (0.0...0d_{-(\alpha+1)}d_{-(\alpha+2)}...)_{10}$ and $DF(v') = (d'_{h-1}d'_{h-2}...d'_0.d'_{-1}d'_{-2}...d'_{-\alpha}...)_{10}$. Figure 6 shows the vertical form of the calculation for $v' = v - \delta$, from which we can clearly conclude that $d_i = d'_i$ for $-(\alpha-1) \leq i \leq h-1$, and that $d'_{-\alpha} = d_{-\alpha} - 1$. There are two cases: $SP(v) = -\alpha$ and $SP(v) \neq -\alpha$. For the former, we have $d_i = 0$ for $-(\alpha-1) \leq i \leq h-1$ and $d_{-\alpha} \neq 0$ according to the definition of the start decimal significand position. Since $v \neq 10^{-i}$, i.e., $d_{-\alpha} \neq 1$, we have $d'_{-\alpha} = d_{-\alpha} - 1 \neq 0$, i.e., $SP(v') = -\alpha = SP(v)$. For the latter, as $v \neq 0$ and $SP(v) \neq -\alpha$, there must exist $j \in \{h-1, h-2, ..., -(\alpha-1)\}$ such that $d_j \neq 0$. Suppose d_{j^*} is the first one for $d_j \neq 0$, i.e., $SP(v) = j^*$. Because $d'_i = d_i$ for $-(\alpha-1) \leq i \leq h-1$, d'_{j^*} is also the first one for $d'_j \neq 0$, i.e., $SP(v') = j^* = SP(v)$. □

When $v = 10^{-i}$, $i > 0$, Theorem 2 does not hold. Figure 5(a) gives an example of $v = 0.1$ with $SP(v) = -1$. If performing the erasing operation on v, we get $v' = 0.0625$ with $SP(v') = -2$.

Theorem 3. *Given a double value $v = 10^{-i}$, $i > 0$, and its best mantissa prefix number v', we have $SP(v) = SP(v') + 1$.*

Proof. Suppose $\alpha = DP(v)$, we have $\alpha > 0$ and $v = 10^{-\alpha}$. The exponent value of the v's underlying storage is $e = \lfloor log_2|v| \rfloor + 1023 = \lfloor -\alpha \times log_2 10 \rfloor + 1023$. Based on Equation (5), we have $g(\alpha) = \lceil \alpha \times log_2 10 \rceil + \lfloor -\alpha \times log_2 10 \rfloor = 0$. That is, we will erase all of the mantissa bits, so $v' = (-1)^s \times 2^{\lfloor log_2|v| \rfloor} = 2^{\lfloor log_2 10^{-\alpha} \rfloor}$. Let $v \div v' = 10^{-\alpha} \div 2^{\lfloor log_2 10^{-\alpha} \rfloor} = 2^{log_2 10^{-\alpha}} \div 2^{\lfloor log_2 10^{-\alpha} \rfloor} = 2^{log_2 10^{-\alpha} - \lfloor log_2 10^{-\alpha} \rfloor}$. Since $log_2 10^{-\alpha} - \lfloor log_2 10^{-\alpha} \rfloor \in (0, 1)$, we have $v \div v' \in (1, 2)$. Further $v' \in (0.5 \times 10^{-\alpha}, 10^{-\alpha})$. Consequently, $SP(v) = SP(v') + 1$. □

According to Theorem 2 and Theorem 3, Equation (6) can be rewritten as:

$$\alpha = \begin{cases} \beta - (SP(v') + 1) & v \neq 10^{-i}, i > 0 \\ \beta - (SP(v') + 2) & v = 10^{-i}, i > 0 \end{cases} \tag{7}$$

For any normal number v, its decimal significand count β will not be zero. Besides, if we know $v = 10^{-SP(v)}$, we can easily get v from v' by the following equation:

$$v = 10^{-(SP(v')+1)} \tag{8}$$

To this end, we can record a modified decimal significand count β^* for the calculation of α.

$$\beta^* = DS^*(v) = \begin{cases} 0 & v = 10^{-i}, i > 0 \\ \beta & others \end{cases} \tag{9}$$

Although there are 18 possible different values of β^*, i.e., $\beta^* \in \{0, 1, 2, ..., 17\}$, we do not consider the situations when $\beta^* = 16$ or 17, because for these two situations, we can only erase a small number of bits but need more bits to record β^*, which leads to a negative gain (more details will be discussed in Section 5.1). For example, as shown in Figure 5(b), given $v = 3.141592653589792$ with $\beta = 16$, we can erase one bit only. In our implementation, we leverage 4 bits to record β^* for $0 \leq \beta^* \leq 15$. To ensure a positive gain, when $52 - g(\alpha) \leq 4$, we do not perform the erasing operation.

3.1.4 When δ is Zero. As shown in Figure 5(c), given $v = 0.75$, we get $v' = v$ and $\delta = 0$. In this situation, we cannot recover v from v' according to Equation (4). In fact, $\delta = 0$ indicates that v itself has long trailing zeros. Therefore, once $\delta = 0$, we will keep v as it is.

3.1.5 Summary of Elf *Eraser.* We utilize one bit as a flag to indicate whether we have erased v or not. As shown in Algorithm 1, *Elf* Eraser takes as input a double value v and an output stream *out*.

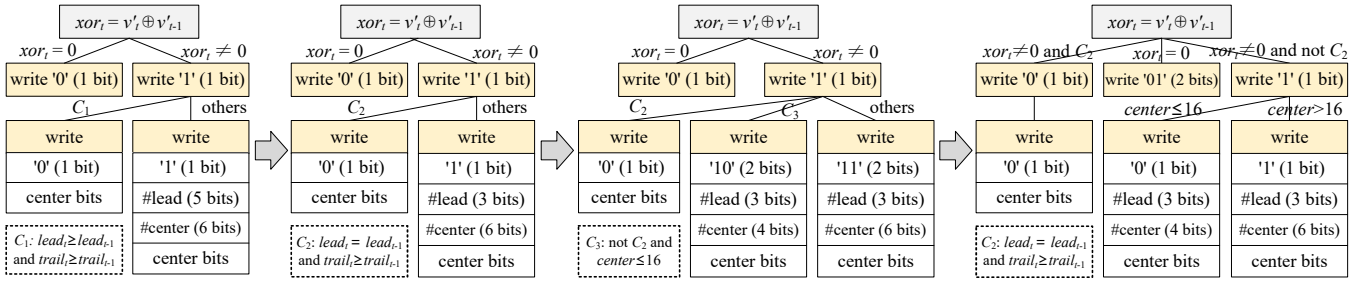
We first calculate the decimal place count α and modified decimal significand count β^* based on Equation (9), and get δ by extracting the least $52 - g(\alpha)$ significant mantissa bits of v (Lines 1-2).

If the three conditions (i.e., $\beta^* < 16$, $\delta \neq 0$ and $52 - g(\alpha) > 4$) hold simultaneously, the output stream *out* writes one bit of "1" to indicate that v should be transformed, followed by 4 bits of β^* for the recovery of v. We get v' by erasing the least $52 - g(\alpha)$ significant mantissa bits of v (Lines 4-5). Otherwise, the output stream *out* writes one bit of "0", and v' is assigned v without any modification (Line 7).

Finally, the obtained v' is passed to an XOR-based compressor together with *out* for further compression (Line 8).

3.2 *Elf* Restorer

Elf Restorer is an inverse process of *Elf* Eraser. Algorithm 2 depicts the pseudo-code of *Elf* Restorer, which takes in an input stream *in*.



(a) Gorilla Compressor (b) Optimizing Leading Code (c) Optimizing Center Code (d) Reassigning Flag Code

Figure 7: Evolutionary Process of *Elf* $\mathbf{XOR}_{cmp}$ for v'_t ($t \neq 1$).

Algorithm 1: $ElfEraser(v, out)$

```
1  α ← DP(v), β* ← DS*(v); // Equation (9)
2  δ ← ~(0xffffffffffffffL << (52 − g(α))) & v;
3  if β* <16 and δ ≠ 0 and 52 − g(α) > 4 then // perform erasing
4      out.writeBit("1"); out.write(β*, 4);
5      v' ← (0xffffffffffffffL << (52 − g(α))) & v;
6  else // do not perform erasing
7      out.writeBit("0"); v' ← v;
8  XOR_cmp(v', out);
```

Algorithm 2: $ElfRestorer(in)$

```
1  flag ← in.read(1);
2  if flag = 0 then // no restoration required
3      v ← XOR_dcmp(in);
4  else // perform restoring
5      β* ← in.read(4); v' ← XOR_dcmp(in);
6      if β* = 0 then
7          v ← 10^−(SP(v')+1); // Equation (8)
8      else
9          α ← β* − (SP(v') + 1); // Equation (7)
10         v ← LeaveOut(v', α) + 10^−α; // Equation (4)
11 return v;
```

First, we read one bit from the input stream *in* to get the modification flag $flag$ (Line 1), which has two cases:

(1) If $flag$ equals to 0, it means that we have not modified the original value, so we get a value from an XOR-based decompressor and assign it to v directly (Line 3).

(2) Otherwise, we read 4 bits from the input stream to get the modified decimal significand count β^*, and then get a value v' from an XOR-based decompressor. If β^* equals to 0, v has a format of 10^{-i}, where $i = SP(v') + 1$ (Line 7). If $\beta^* \neq 0$, we can recover v from β^* and v' based on Equation (7) and Equation (4) (Lines 9-10).

Finally, the recovered v is returned (Line 11).

4 *ELF* $\mathbf{XOR}_{cmp}$ AND $\mathbf{XOR}_{dcmp}$

Theoretically, any existing XOR-based compressor such as Gorilla [46] and Chimp [34] can be utilized in *Elf*. Since the erased value v' tends to contain long trailing zeros, to compress the time series compactly, in this section, we propose a novel XOR-based compressor and the corresponding decompressor.

4.1 *Elf* $\mathbf{XOR}_{cmp}$

4.1.1 First Value Compression. Existing XOR-based compressors store the first value v'_1 of a time series using 64 bits. However, after being erased some insignificant mantissa bits, v'_1 tends to have a large number of trailing zeros. As a result, we leverage $\lceil log_2 65 \rceil = 7$ bits to record the number of trailing zeros $trail$ of v'_1 (note that $trail$ can be assigned a total of 65 values from 0 to 64), and store v'_1's non-trailing bits with $64 - trail$ bits. In all, we utilize $71 - trail$ bits to record the first value, which is usually less than 64 bits.

4.1.2 Other Values Compression. For each value v'_t that $t > 1$, we store $xor_t = v'_t \oplus v'_{t-1}$ as most existing XOR-based compressors did. Our proposed XOR-based compressor is extended from Gorilla [46] and at the same time borrows some ideas from Chimp [34].

Gorilla Compressor. As shown in Figure 7(a), Gorilla compressor checks whether xor_t is equal to 0 or not. If $xor_t = 0$ (i.e., $v'_t = v'_{t-1}$), Gorilla writes one bit of "0", and thus it can save many bits without actually storing v'_t. If $xor_t \neq 0$, Gorilla writes one bit of "1" and further checks whether the condition C_1 is satisfied. Here C_1 is "$lead_t \geq lead_{t-1}$ and $trail_t \geq trail_{t-1}$", meaning that the leading zeros count and trailing zeros count of xor_t are greater than or equal to those of xor_{t-1}, respectively. If C_1 does not hold, after writing a bit of "1", Gorilla stores the leading zeros count and center bits count with 5 bits and 6 bits respectively, followed by the actual center bits. Otherwise, xor_t shares the information of leading zeros count and center bits count with xor_{t-1}, which is expected to save some bits.

Leading Code Optimization. Observing that the leading zeros count of an XORed value is rarely more than 30 or less than 8, Chimp [34] proposes to use only $log_2 8 = 3$ bits to represent up to 24 leading zeros. In particular, Chimp leverages 8 exponentially decaying steps (i.e., 0, 8, 12, 16, 18, 20, 22, 24) to approximately represent the leading zeros count. If the actual leading zeros count is between 0 and 7, Chimp approximates it to be 0; if it is between 8 and 11, Chimp regards it as 8; and so on. The condition of C_1 is therefore converted into C_2, i.e., "$lead_t = lead_{t-1}$ and $trail_t \geq trail_{t-1}$". By applying this optimization to the Gorilla compressor, we can get a compressor shown in Figure 7(b).

Center Code Optimization. Both v'_t and v'_{t-1} are supposed to have many trailing zeros, which results in an XORed value with long trailing zeros. Besides, v'_t would not differentiate much from v'_{t-1} in most cases, contributing to long leading zeros in the XORed value. That is, the XORed value tends to have a small number of

Algorithm 3: $ElfXOR_{cmp}(v'_t, out)$

```
1  if v'_t is the first value then // compress the first value
2      lead_t ← ∞; trail_t ← numOfTrailingZeros(v'_t);
3      out.write(trail_t, 7);
4      out.write(nonTrailingBits(v'_t), 64 − trail_t);
5  else // compress other values
6      xor ← v'_t ⊕ v'_{t−1};
7      if xor = 0 then // case 01 in Figure 7(d)
8          out.writeBit("01");
9          lead_t ← lead_{t−1}; trail_t ← trail_{t−1};
10     else
11         lead_t ← binNumOfLeadingZeros(xor);
12         trail_t ← numOfTrailingZeros(xor);
13         center ← 64 − lead_t − trail_t;
14         if lead_t = lead_{t−1} and trail_t ≥ trail_{t−1} then
15             out.writeBit("00"); // case 00 in Figure 7(d)
16         else if center ≤ 16 then // case 10 in Figure 7(d)
17             out.writeBit("10");
18             out.write(lead_t, 3); out.write(center, 4);
19         else // case 11 in Figure 7(d)
20             out.writeBit("11");
21             out.write(lead_t, 3); out.write(center, 6);
22         out.write(centerBits(v'_t), center);
```

center bits (usually not more than 16). To this end, if the center bits count is less than or equal to 16, we use only $log_2 16 = 4$ bits to encode it. Although we need one more flag bit, we can usually save one bit in comparison with the original solution. After optimizing the center code, we get a compressor shown in Figure 7(c).

Flag Code Reassignment. Figure 7(c) shows that we use only 1 flag bit for the case of $xor_t = 0$, but 2 or 3 flag bits for the cases of $xor_t \neq 0$. As pointed out by Chimp [34], identical consecutive values are not very frequent in floating-point time series. Thus, using only 1 bit to indicate the case of $xor_t = 0$ is not particularly effective. To this end, we reassign the flag codes to the four eases. Therefore, each case uses only 2 bits of flag, as illustrated in Figure 7(d).

4.1.3 Summary of Elf *XOR$_{cmp}$.* Algorithm 3 depicts the pseudo-code of *Elf* XOR$_{cmp}$, which is self-explanatory. In Lines 1-4, we deal with the first value of a time series, and in Lines 6-22, we handle the four cases shown in Figure 7(d) respectively. Note that the function $binNumOfLeadingZeros(xor)$ in Line 11 calculates the approximate leading zeros count of xor, as discussed above.

4.2 *Elf* XOR$_{dcmp}$

The decompressor takes opposite actions of the compressor. As shown in Algorithm 4, the proposed decompressor *Elf* XOR$_{dcmp}$ takes an input stream *in* as input. We decompress the first value in Lines 1-3, and cope with the four cases respectively in Lines 5-16. For case 01, the algorithm sets the current value v'_t as the previous one v'_{t-1}. For case 00, case 10 and case 11, we first update the leading zeros count $lead_t$ and trailing zeros count $trail_t$, with which we can calculate the center bits count *center* (Line 15) and get the current value v'_t (Line 16). At last, v'_t is returned to *Elf* Restorer (Line 17).

Algorithm 4: $ElfXOR_{dcmp}(in)$

```
1  if it is the first value then // decompress the first value
2      lead_t ← ∞; trail_t ← in.read(7);
3      v'_t ← in.read(64 − trail_t) << trail_t;
4  else // decompress other values
5      flag ← in.read(2);
6      if flag = "01" then // case 01 in Figure 7(d)
7          lead_t ← lead_{t−1}; trail_t ← trail_{t−1}; v'_t ← v'_{t−1};
8      else
9          if flag = "00" then // case 00 in Figure 7(d)
10             lead_t ← lead_{t−1}; trail_t ← trail_{t−1};
11         else if flag = "10" then // case 10 in Figure 7(d)
12             lead_t ← in.read(3); trail_t ← in.read(4);
13         else // case 11 in Figure 7(d)
14             lead_t ← in.read(3); trail_t ← in.read(6);
15         center ← 64 − lead_t − trail_t;
16         v'_t ← (in.read(center) << trail_t) ⊕ v'_{t−1};
17 return v'_t;
```

5 DISCUSSION

In this section, we first analyze the effectiveness and complexity of *Elf* compression algorithm, and then investigate a possible variant. Finally, we extend *Elf* to the special numbers of double values.

5.1 Effectiveness Analysis

Elf Eraser transforms a floating-point value to another one with more trailing zeros under a guaranteed bound (see Theorem 4), so it can potentially improve the compression ratio of most XOR-based compression methods tremendously.

Theorem 4. *Given a double value v with its decimal significand count $\beta = DS(v)$, we can erase x bits in its mantissa, where* $51 - \beta log_2 10 < x < 53 - (\beta - 1) log_2 10$.

Proof. Suppose $\alpha = DP(v)$, we have:

$$DF(v) = \begin{cases} (d_{\beta-\alpha-1} d_{\beta-\alpha-2} ... d_0 . d_{-1} d_{-2} ... d_{-\alpha})_{10} & \text{if} v \geq 1 \\ (0.00 ... d_{\beta-\alpha-1} d_{\beta-\alpha-2} ... d_{-\alpha})_{10} & \text{if} v < 1 \end{cases}$$

$\Longrightarrow 10^{\beta-\alpha-1} \leq v < 10^{\beta-\alpha} \Longrightarrow log_2 10^{\beta-\alpha-1} \leq log_2 v < log_2 10^{\beta-\alpha}$
$\Longrightarrow \lfloor (\beta - \alpha - 1) log_2 10 \rfloor \leq \lfloor log_2 v \rfloor \leq \lfloor (\beta - \alpha) log_2 10 \rfloor$
$\Longrightarrow \lceil \alpha log_2 10 \rceil + \lfloor (\beta - \alpha - 1) log_2 10 \rfloor \leq \lceil \alpha log_2 10 \rceil + \lfloor log_2 v \rfloor = g(\alpha)$
$\leq \lceil \alpha log_2 10 \rceil + \lfloor (\beta - \alpha) log_2 10 \rfloor$
$\Longrightarrow \alpha log_2 10 + (\beta - \alpha - 1) log_2 10 - 1 < g(\alpha) < \alpha log_2 10 + 1 + (\beta - \alpha) log_2 10 \Longrightarrow (\beta - 1) log_2 10 - 1 < g(\alpha) < \beta log_2 10 + 1$
$\Longrightarrow 51 - \beta log_2 10 < 52 - g(\alpha) = x < 53 - (\beta - 1) log_2 10.$ □

According to Theorem 4, the number of erased bits is dependent merely on the decimal significand count β of the double value. A bigger β usually means fewer bits erased. If $\beta \leq 14$, we can erase at least $\lceil 51 - 14 \times log_2 10 \rceil = 5$ bits, which always guarantees a positive gain. But if $\beta \geq 16$, we can only erase at most $\lfloor 53 - (16 - 1) \times log_2 10 \rfloor = 3$ bits, leading to a negative gain as it requires at least 4 bits to record β^*. As a consequence, *Elf* compression algorithm keeps v as it is when $DS(v) \geq 16$.

5.2 Complexity Analysis

5.2.1 Time Complexity. For each value, *Elf* Eraser (i.e., Algorithm 1) can directly determine the erased bits in $O(1)$ and perform the erasing operation by efficient bitwise manipulations. In *Elf* XOR_{cmp} (i.e., Algorithm 3), all operations can be performed in $O(1)$. For *Elf* Decompressor, Restorer (i.e., Algorithm 2) and XOR_{dcmp} (i.e., Algorithm 4) sequentially read data from an input stream and perform all operations in $O(1)$. Overall, the time complexity of *Elf* is $O(N)$, where N is the length of a time series.

Our proposed *Elf* compression algorithm performs an extra erasing step before actually compressing the data. It is reasonable that the overall computation complexity of *Elf* compression algorithm is a little bit higher than that of other XOR-based compression methods, e.g., Gorilla and Chimp. However, thanks to the erasing action, our method can achieve a much better compression ratio.

5.2.2 Space Complexity. Neither Eraser nor Restorer stores any data, while both XOR_{cmp} and XOR_{dcmp} only store and utilize the previous leading zeros count $lead_{t-1}$, trailing zeros count $trail_{t-1}$ and value v'_{t-1}. To this end, the space complexity of *Elf* is $O(1)$.

5.3 A Possible Variant Discussion

In the *Elf* erasing process, we let $v' = v - \delta$ where $0 \le \delta < 10^{-\alpha}$. Can we let $0 \le \delta < k \times 10^{-\alpha}$, $k \in \{1, 2, ..., 9\}$, which is supposed to make v' have more trailing zeros?

The decimal value $k \times 10^{-\alpha}$ can be represented by $f_k(\alpha) = \lceil |log_2(k \times 10^{-\alpha})| \rceil = \lceil |log_2 k - \alpha log_2 10| \rceil$ binary bits. Since $k < 10$ and $\alpha \ge 1$, $f_k(\alpha) = \lceil \alpha log_2 10 - log_2 k \rceil$. Back to Theorem 1, $\delta = \sum_{i=f_k(\alpha)+1}^{|\bar{l}|} b_i \times 2^{-i} \le \sum_{i=f_k(\alpha)+1}^{|\bar{l}|} 2^{-i} < \sum_{i=f_k(\alpha)+1}^{+\infty} 2^{-i} = 2^{-f_k(\alpha)} = 2^{-\lceil \alpha log_2 10 - log_2 k \rceil} \le 2^{-(\alpha log_2 10 - log_2 k)} = 2^{log_2(k \times 10^{-\alpha})} = k \times 10^{-\alpha}$. That is to say, if we erase the bits after $b_{-f_k(\alpha)}$ in $BF(v)$, we can still recover v by $LeaveOut(v', \alpha) + k' \times 10^{-\alpha}$, where $LeaveOut$ has the same meaning with that in Equation (4), and $k' \in \{1, 2, ..., k\}$. But it requires $\lceil log_2 k \rceil$ bits to store k'. We call this method Elf_k.

Theorem 5. *Elf_k will not achieve a better gain than Elf.*

Proof. Suppose y is the additional number of bits that Elf_k can erase over Elf (i.e., Elf_1), then $y - \lceil log_2 k \rceil$ is the gain of Elf_k over Elf. We have: $y = (52 - g_k(\alpha)) - (52 - g_1(\alpha)) = \lceil \alpha log_2 10 \rceil - \lceil \alpha log_2 10 - log_2 k \rceil \Longrightarrow \alpha log_2 10 - (\alpha log_2 10 - log_2 k + 1) < y < (\alpha log_2 10 + 1) - (\alpha log_2 10 - log_2 k) \Longrightarrow log_2 k - 1 < y < log_2 k + 1 \Longrightarrow log_2 k - 1 - \lceil log_2 k \rceil < y - \lceil log_2 k \rceil < log_2 k + 1 - \lceil log_2 k \rceil \Longrightarrow -2 < y - \lceil log_2 k \rceil < 1$. It means that Elf_k would consume the same bits with or one more bit than Elf. □

5.4 *Elf* for Special Numbers

As shown in Figure 3, according to the values of $\vec{e}$ and $\vec{m}$, there are four types of special numbers:

(1) **Zero**. If $\forall i \in \{1, 2, ..., 11\}$, $e_i = 0$ and $\forall j \in \{1, 2, ..., 52\}$, $m_j = 0$, then v represents a zero.

(2) **Infinity**. If $\forall i \in \{1, 2, ..., 11\}$, $e_i = 1$ and $\forall j \in \{1, 2, ..., 52\}$, $m_j = 0$, then v stands for an infinity.

(3) **Not a Number**. If $\forall i \in \{1, 2, ..., 11\}$, $e_i = 1$ and $\exists j \in \{1, 2, ..., 52\}$, $m_j = 1$, then v is not a number (i.e., $v = NaN$).

(4) **Subnormal Number**. If $\forall i \in \{1, 2, ..., 11\}$, $e_i = 0$ and $\exists j \in \{1, 2, ..., 52\}$, $m_j = 1$, then v is a subnormal number (or a subnormal).

In this case, we have the following equation:

$$\begin{aligned} v &= (-1)^s \times 2^{-1022} \times (0.m_1 m_2 ... m_{52})_2 \\ &= (-1)^s \times 2^{-1022} \times \sum_{i=1}^{52} m_i \times 2^{-i} \end{aligned} \tag{10}$$

For these four special numbers, their restorers, compressors and decompressors are the same with that of normal numbers, but their erasers need to be tailored carefully.

Zero and Infinity Eraser. If v is a zero or infinity, we do not perform *Elf* erasing because all its mantissa bits are already 0s.

NaN Eraser. If v is NaN, in order to make its trailing zeros as many as possible, we perform the NaN_{norm} operation on it, which sets $m_1 = 1$ and $m_i = 0$ for $i \in \{2, 3, ..., 52\}$, i.e.,

$$v' = NaN_{norm}(v) = \text{0xfff8000000000000L} \ \& \ v \tag{11}$$

Subnormal Number Eraser. According to Equation (2) and Equation (10), subnormal numbers can be regarded as the special cases of normal numbers by setting $e = 1$ and $m_0 = 0$. As a result, we can compress subnormal numbers in the same way of normal numbers using *Elf* Eraser.

6 EXPERIMENTS

6.1 Datasets and Experimental Settings

6.1.1 Datasets. To verify the performance of *Elf* compression algorithm, we adopt 22 datasets including 14 time series and 8 non time series, which are further divided into three categories respectively according to their average decimal significand counts (as described in Table 2). Apart from the datasets used by Chimp [34], we also add three datasets (i.e., Vehicle-charge, City-lat and City-lon) to enrich the non time series with small and medium decimal significand counts. Each time series is ordered by the timestamps, while each non time series is in a random order given by its data publisher.

City-temp [2], collected by the University of Dayton to record the temperature of major cities around the world.

IR-bio-temp [43], which exhibits the changes in the temperature of infrared organisms.

Wind-speed [40], which describes the wind speed.

PM10-dust [42], which records near real-time measurements of PM10 in the atmosphere.

Stocks-UK, Stocks-USA and Stocks-DE [5], which contain the stock exchange prices of UK, USA and German respectively.

Dewpoint-temp [44], which records relative dew point temperature observed by sensors floating on rivers and lakes.

Air-pressure [41], which shows Barometric pressure corrected to sea level and surface level.

Basel-wind and Basel-temp [7], which respectively record the historical wind speed and temperature of Basel, Switzerland.

Bitcoin-price [8], which includes the price of Bitcoin in dollar exchange rate.

Bird-migration [8], an online dataset of animal tracking data that records the position of birds and the vegetation.

Air-sensor [8], a synthetic dataset recording air sensor data with random noise.

Food-price [6], global food prices data from the World Food Programme.

Table 2: Details of Datasets

Dataset			#Records	β	Time Span
Time Series	**Small β**	City-temp (CT)	2,905,887	3	25 years
		IR-bio-temp (IR)	380,817,839	3	7 years
		Wind-speed (WS)	199,570,396	2	6 years
		PM10-dust (PM10)	222,911	3	5 years
	Medium β	Stocks-UK (SUK)	115,146,731	5	1 year
		Stocks-USA (SUSA)	374,428,996	4	1 year
		Stocks-DE (SDE)	45,403,710	6	1 year
		Dewpoint-temp (DT)	5,413,914	4	3 years
		Air-pressure (AP)	137,721,453	7	6 years
		Basel-wind (BW)	124,079	8	14 years
		Basel-temp (BT)	124,079	9	14 years
		Bitcoin-price (BP)	2,741	9	1 month
		Bird-migration (BM)	17,964	7	1 year
	Large β	Air-sensor (AS)	8,664	17	1 hour
Non Time Series	**Small β**	Food-price (FP)	2,050,638	3	-
		Vehicle-charge (VC)	3,395	3	-
	Medium β	Blockchain-tr (BTR)	231,031	5	-
		SD-bench (SB)	8,927	4	-
		City-lat (CLat)	41,001	6	-
		City-lon (CLon)	41,001	7	-
	Large β	POI-lat (PLat)	424,205	16	-
		POI-lon (PLon)	424,205	16	-

Vehicle-charge [3], which records the total energy use and charge time of a collection of electric vehicles.

Blockchain-tr [1], which records the transaction value of Bitcoin for a single day.

SD-bench [10], which describes the performance of multiple storage drives through a standardized series of tests.

City-lat, City-lon [11], which records the latitude and longitude of the cities and towns all over the world.

POI-lat, POI-lon [9], the coordinates in radian of Position-of-Interests (POI) extracted from Wikipedia.

6.1.2 Baselines. We compare *Elf* compression algorithm with **four** state-of-the-art lossless floating-point compression methods (i.e., Gorilla [46], Chimp [34], Chimp_{128} [34] and FPC [16]) and **five** widely-used general compression methods (i.e., Xz [12], Brotli [13], LZ4 [19], Zstd [18] and Snappy [21]). By regarding *Elf* Eraser as a preprocessing step, we also compare **three** variants of Gorilla, Chimp and Chimp_{128} (denoted as Gorilla+Eraser, Chimp+Eraser and Chimp_{128}+Eraser respectively) to verify the effectiveness of the erasing and XOR_{cmp} strategies. Most implementations of these competitors are extended from [34]. To make a fair comparison, we optimize the stream implementation of Gorilla as the same as Chimp [34], which improves the efficiency of Gorilla tremendously. All source codes and datasets are publicly available [4].

6.1.3 Metrics. We verify the performance of various methods in terms of three metrics: compression ratio, compression time and decompression time. Note that the compression ratio is defined as the ratio of the compressed data size to the original one.

6.1.4 Settings. As Chimp [34] did, we regard 1,000 records of each dataset as a block. Each compression method is executed on up to 100 blocks per dataset, and the average metrics of one block are finally reported. All experiments are conducted on a personal computer equipped with Windows 11, 11th Gen Intel(R) Core(TM) i5-11400 @ 2.60GHz CPU and 16GB memory. The JDK (Java Development Kit) version is 1.8.

6.2 Overall Comparison with Baselines

Table 3 shows the performance of different compression algorithms on all datasets. We group the datasets into two categories (i.e., Time Series and Non Time Series), and investigate the performance of floating-point compression algorithms and general compression algorithms on each group of datasets, respectively.

6.2.1 Compression Ratio. With regard to the compression ratio, we have the following observations from Table 3.

(1) ***Elf* VS floating-point compression algorithms**. Among all the floating-point compression algorithms, *Elf* has the best compression ratio on almost all datasets. In particular, for the time series datasets, compared with Gorilla and FPC, *Elf* has an average relative improvement of $(0.76 - 0.37)/0.76 \approx 51\%$. Chimp has optimized the coding of Gorilla, and its upgraded version Chimp_{128} resorts to a hash table (up to 33KB memory occupation) for fast searching an appropriate value in previous 128 data records. Therefore, they can achieve a significant improvement over Gorilla. However, thanks to the erasing technique and elaborate XOR_{cmp}, *Elf* can still achieve relative improvement of 47% and 12% over Chimp and Chimp_{128} respectively on the time series datasets. Note that *Elf* has a lower memory footprint (i.e., $O(1)$) in comparison with Chimp_{128}. For the non time series datasets, *Elf* is also relatively $(0.63 - 0.55)/0.63 \approx 12.7\%$ better than the best competitor Chimp_{128}. We notice that there are few datasets that Chimp_{128} is slightly better than *Elf* in terms of compression ratio. For the datasets of WS, SUSA and BT, we find that there are many duplicate values within 128 consecutive records. In this case, Chimp_{128} can use only 9 bits to represent the same value. For the datasets of AS, PLat and PLon, since they have large decimal significand counts, *Elf* does not perform erasing but still consumes some flag bits. As pointed out by [34], real-world floating point measurements often have a decimal place count of one or two, which usually results in small or medium β. To this end, *Elf* can achieve good performance in most real-world scenarios.

(2) ***Elf* VS general compression algorithms**. Most of the general compression algorithms have a good compression ratio. However, upon most occasions, *Elf* is still better than LZ4, Zstd and Snappy (with average relative improvement of 30.2%, 7.5% and 27.5% respectively for the time series datasets, and 18%, 3.5% and 16.7% respectively for the non time series datasets), and shows a similar performance to Xz and Brotli in terms of compression ratio. Moreover, in comparison with non time series datasets, *Elf* can achieve more improvement over general compression algorithms for time series datasets (e.g., 30.2% v.s. 18% for LZ4). It is because non time series datasets do not have a time-based ordering, which reduces the usefulness of exploiting previous values.

(3) **Different decimal significand counts**. As shown in Table 3, with a larger β, both general and floating-point compression algorithms suffer from a lower compression ratio, since a larger β

Table 3: Overall comparison with baselines (the best values in each group are marked in bold). The compression ratio, compression time and decompression time are the average measurements on one block (i.e., 1,000 values).

		Dataset	Time Series															Non Time Series								
			Small β				Medium β									Large β	Avg.	Small β		Medium β				Large β		Avg.
			CT	IR	WS	PM10	SUK	SUSA	SDE	DT	AP	BW	BT	BP	BM	AS		FP	VC	BTR	SB	CLat	CLon	PLat	PLon	
Compression Ratio	Floating	Gorilla	0.85	0.64	0.83	0.48	0.58	0.68	0.72	0.83	0.73	0.99	0.94	0.84	0.79	0.82	0.76	0.58	1.00	0.74	0.63	1.03	1.03	1.03	1.03	0.88
		Chimp	0.64	0.59	0.81	0.46	0.52	0.64	0.67	0.77	0.65	0.88	0.85	0.77	0.72	**0.77**	0.70	0.47	0.86	0.67	0.55	0.92	0.98	**0.90**	**0.99**	0.79
		$Chimp_{128}$	0.32	0.24	**0.23**	0.21	0.29	**0.23**	0.27	0.35	0.54	0.71	**0.47**	0.72	0.50	**0.77**	0.42	0.34	0.36	0.55	**0.27**	0.78	0.85	**0.90**	**0.99**	0.63
		FPC	0.75	0.61	0.85	0.50	0.74	0.70	0.73	0.82	0.67	0.92	0.90	0.81	0.75	0.82	0.75	0.62	0.91	0.69	0.59	0.96	1.00	0.95	1.00	0.84
		Elf	**0.25**	**0.21**	0.25	**0.16**	**0.22**	0.24	**0.26**	**0.31**	**0.31**	**0.59**	0.58	**0.56**	**0.42**	0.85	**0.37**	**0.23**	**0.34**	**0.36**	**0.27**	**0.56**	**0.63**	0.96	1.06	**0.55**
	General	Xz	**0.18**	**0.16**	**0.15**	**0.11**	**0.16**	**0.17**	**0.19**	**0.27**	**0.47**	**0.57**	**0.35**	**0.63**	**0.43**	**0.79**	**0.33**	**0.23**	**0.23**	**0.40**	**0.13**	**0.60**	**0.63**	**0.93**	**0.96**	**0.51**
		Brotli	0.20	0.18	0.17	0.12	0.19	0.20	0.22	0.32	0.51	0.61	0.39	0.71	0.47	0.85	0.37	0.26	0.28	0.43	0.14	0.65	0.68	0.94	**0.96**	0.54
		LZ4	0.36	0.36	0.37	0.27	0.39	0.39	0.41	0.52	0.69	0.69	0.54	0.87	0.61	1.01	0.53	0.41	0.47	0.53	0.30	0.79	0.82	1.00	1.00	0.67
		Zstd	0.22	0.24	0.19	0.14	0.22	0.24	0.26	0.38	0.58	0.61	0.41	0.75	0.51	0.91	0.40	0.30	0.34	0.45	0.17	0.68	0.71	0.94	**0.96**	0.57
		Snappy	0.29	0.30	0.27	0.21	0.32	0.32	0.35	0.51	0.73	0.75	0.54	0.99	0.61	1.00	0.51	0.39	0.42	0.54	0.25	0.83	0.87	1.00	1.00	0.66
Compression Time (μs)	Floating	Gorilla	**18**	**21**	**17**	**15**	**17**	**17**	**17**	**18**	**20**	**21**	**20**	**19**	**18**	**20**	**18**	**16**	**19**	**18**	**16**	**19**	**19**	**19**	**19**	**18**
		Chimp	23	**21**	22	18	23	22	23	24	**20**	26	25	24	25	27	23	21	24	22	20	26	26	23	26	23
		$Chimp_{128}$	23	23	22	20	24	22	25	26	38	47	35	48	38	50	32	27	27	39	23	48	48	45	46	38
		FPC	34	40	40	40	28	28	28	31	40	42	47	27	30	38	35	39	43	43	41	42	48	40	48	43
		Elf	51	53	59	50	54	56	58	57	51	73	69	63	65	87	60	52	55	62	48	64	70	71	72	62
	General	Xz	948	1106	810	1056	877	836	900	1045	1959	1527	1100	1531	1444	2146	1235	898	1636	1036	1040	1252	1516	1476	1351	1276
		Brotli	1639	1685	1557	1449	1584	1611	1693	1702	2074	1792	1715	1729	1827	1798	1704	1741	1674	1755	1522	1692	1712	1628	1633	1669
		LZ4	1082	1106	963	984	966	976	952	1091	1285	1013	1010	1001	1000	1026	1032	985	974	1060	976	988	986	966	957	987
		Zstd	209	**212**	112	**208**	177	112	**117**	218	317	259	291	271	**256**	277	217	211	**227**	251	202	236	245	206	**113**	211
		Snappy	**195**	236	**52**	214	**169**	**56**	172	**195**	**179**	**189**	**200**	**169**	261	**158**	**175**	**188**	250	**190**	**200**	**207**	**238**	**178**	149	**200**
Decompression Time (μs)	Floating	Gorilla	**16**	18	17	21	**16**	17	**17**	**17**	**18**	**23**	**18**	**16**	**17**	**20**	**18**	**16**	**18**	**17**	**16**	**17**	**17**	**17**	**17**	**17**
		Chimp	24	22	24	19	22	24	24	54	19	30	26	27	25	25	26	21	26	24	21	26	26	24	26	24
		$Chimp_{128}$	17	**16**	**16**	**15**	18	**16**	18	18	22	28	21	26	22	25	20	18	19	22	17	26	26	23	24	22
		FPC	28	28	26	29	25	24	25	25	32	27	31	24	26	34	28	28	29	29	29	30	36	28	35	31
		Elf	38	44	46	43	37	45	44	45	41	58	53	48	48	29	44	33	44	49	39	52	57	31	33	42
	General	Xz	161	147	114	125	156	133	148	226	435	427	284	479	345	629	272	196	194	312	126	434	461	664	663	381
		Brotli	61	58	36	53	41	43	69	70	109	97	79	93	87	100	71	103	70	86	58	243	85	86	77	101
		LZ4	40	**35**	**18**	**37**	**19**	**19**	**18**	42	56	42	**38**	**40**	**38**	**44**	**35**	**36**	**37**	**39**	37	**38**	**37**	35	**19**	**35**
		Zstd	46	48	30	42	31	31	50	45	99	66	113	72	62	68	57	45	47	60	44	47	48	43	32	46
		Snappy	**38**	54	20	38	**19**	21	20	**39**	**49**	**40**	42	41	46	48	37	40	39	**39**	**36**	42	**37**	**32**	43	38

means a more complex data layout. To this end, the poor compression ratio on datasets with a large β is not just a problem for *Elf*. It is a common and interesting problem worthy of further exploration.

6.2.2 Compression Time and Decompression Time. As shown in the lower parts of Table 3, we have the following observations.

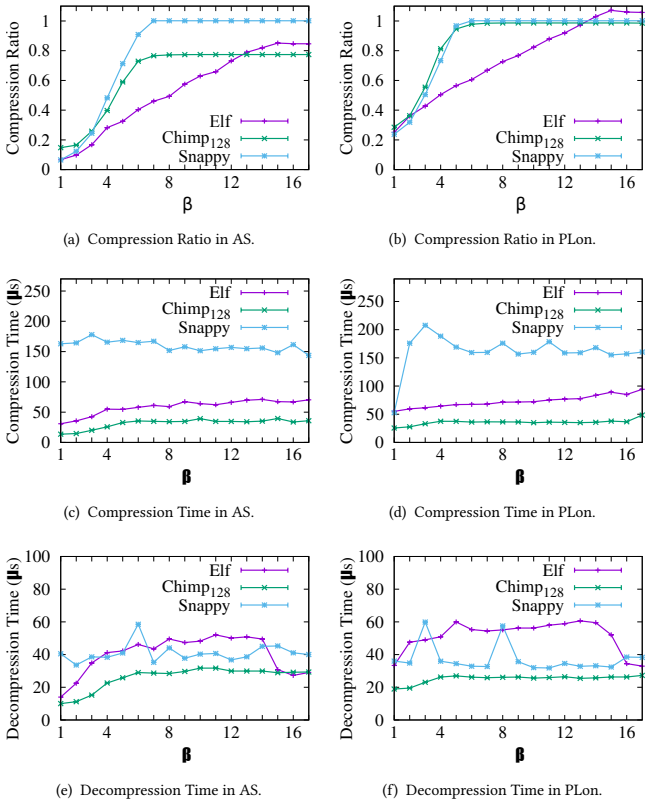
(1) The general compression algorithms take one or two orders of magnitude of more compression time than floating-point compression algorithms on average. For example, although Xz can achieve a slightly better compression ratio than *Elf*, it takes as much as 200 times longer than *Elf*. Even for the fastest general compression algorithms Zstd and Snappy, they still take about 3 times longer than *Elf*, which prevents them from being applied to real-time scenarios.

(2) *Elf* takes a little more time than other floating-point compression algorithms during both compression and decompression processes. Compared with other floating-point compression algorithms, *Elf* adds an erasing step and a restoring step, which inevitably takes more time. However, the difference is not obvious, since they are all on the same order of magnitude. Gorilla has the least compression time and decompression time, because it considers fewer cases (see Figure 7(a)) compared with Chimp and $Chimp_{128}$.

(3) In comparison with compression time, the distinction of decompression time among different algorithms (except for Xz) is insignificant, since most algorithms sequentially read the decompression stream directly. As a result, most algorithms focus more on the trade-off between compression ratio and compression time.

6.2.3 Summary. In summary, *Elf* can usually achieve remarkable compression ratio improvement for both time series datasets and non time series datasets, with the affordable cost of more time.

One interesting question is how much efficiency gain can we benefit from *Elf* over the best competitor $Chimp_{128}$? Consider a scenario of data transmission. Suppose the raw data size is D, the compression ratio is η, and the rates of compression, decompression and transmission are r_{cmp}, r_{dcmp} and r_{tr}, respectively. The latency of the whole data from sending to receiving is: $t = D/r_{cmp} + D/r_{dcmp} + D \times \eta / r_{tr}$. According to Table 3, in terms of the average metrics for time series, we have $r_{cmp}^{Elf} = 1000 \times 64/(60 \times 10^{-6}) \approx 1.07 \times 10^9$ bits/s, $r_{dcmp}^{Elf} = 1000 \times 64/(44 \times 10^{-6}) \approx 1.45 \times 10^9$ bits/s, and $\eta^{Elf} = 0.37$. Similarly, $r_{cmp}^{Chimp_{128}} = 2 \times 10^9$ bits/s, $r_{dcmp}^{Chimp_{128}} = 3.2 \times 10^9$ bits/s,

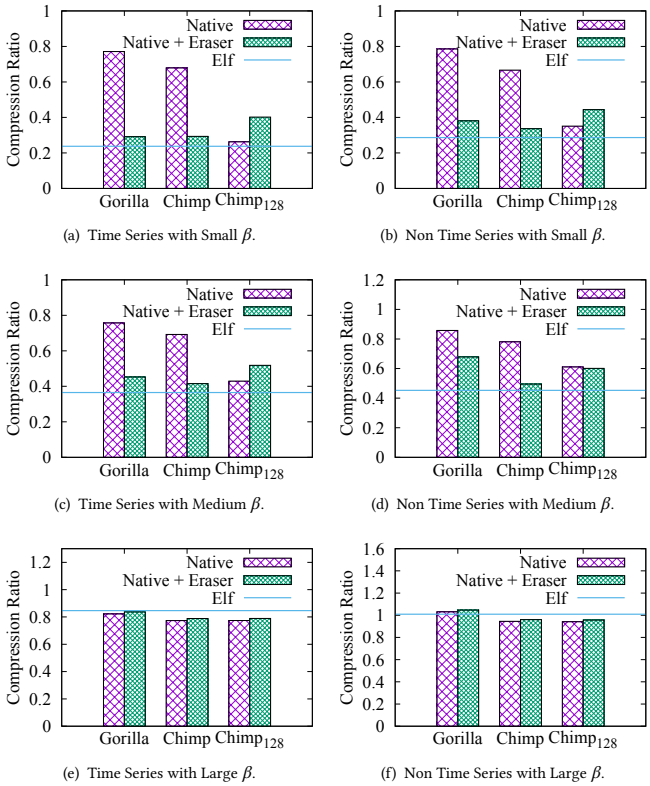


(a) Compression Ratio in AS. (b) Compression Ratio in PLon.

(c) Compression Time in AS. (d) Compression Time in PLon.

(e) Decompression Time in AS. (f) Decompression Time in PLon.

Figure 8: Performance with Different β.



(a) Time Series with Small β. (b) Non Time Series with Small β.

(c) Time Series with Medium β. (d) Non Time Series with Medium β.

(e) Time Series with Large β. (f) Non Time Series with Large β.

Figure 9: Compression Ratio Improvement of Erasing and XOR$_{cmp}$ Strategies.

and $\eta^{Chimp_{128}} = 0.42$. Therefore, $t^{Elf}/t^{Chimp_{128}} \approx (1.62 + 0.37 \times 10^9/r_{tr})/(0.81 + 0.42 \times 10^9/r_{tr})$, where $r_{tr}^{Elf} = r_{tr}^{Chimp_{128}} = r_{tr}$. Let $t^{Elf}/t^{Chimp_{128}} < 1$, we have $r_{tr} < 6.17 \times 10^7$ bits/s. That is, when the transmission rate is smaller than 6.17×10^7 bits/s, the overall performance of *Elf* is supposed to be better than that of Chimp$_{128}$.

We want to emphasize two points here. First, in a typical client-server architecture, the bandwidth and memory in the server are rather precious resources, and the bandwidth for a connection rarely exceeds 6.17×10^7 bits/s. Moreover, for each connection, Chimp$_{128}$ would allocate 33KB memory, which is unaffordable for high concurrency scenarios. Second, we find that the most time-consuming part of *Elf* is to calculate β of a floating-point value. If we could calculate it faster, the efficiency would be further enhanced tremendously. Maybe in the future we can design a special hardware or a special computer instruction to achieve this.

6.3 Performance with Different β

To further investigate the effect of β, we conduct a set of experiments by gradually reducing the decimal significand counts of a time series dataset AS and a non time series dataset PLon. We select Chimp$_{128}$ and Snappy as baselines, since they achieve the best trade-off between the compression ratio and compression time among the floating-point competitors and general competitors respectively.

As shown in Figure 8(a) and Figure 8(b), with an increasing β from 1 to 15, the compression ratio of *Elf* increases linearly, which is consistent with Theorem 4. When β is greater than 15, the compression ratio of *Elf* keeps stable, because *Elf* does not perform the erasing step if $\beta > 15$. For Chimp$_{128}$ and Snappy, with the increase of β, their compression ratios first increase steeply and then keep stable when $\beta > 6$. On both AS and PLon, *Elf* always has the best compression ratio if β is between 3 and 13. When $\beta = 6$, the compression ratio gain of *Elf* over Chimp$_{128}$ and Snappy achieves the highest (33% and 55% relative improvement in AS, and 40.2% and 41.6% relative improvement in PLon, respectively). For the time series dataset AS, *Elf* always performs better than Snappy, because *Elf* can capture the time ordering characteristic.

Figures 8(c-f) present the compression time and decompression time of the three algorithms on the two datasets, respectively. With a larger β that $\beta < 15$, the compression time and decompression time of both *Elf* and Chimp$_{128}$ get larger, because they need to write or read more streams. Things have changed for Snappy because it contains a complex dictionary building step. When $\beta \geq 15$, the decompression time of *Elf* drops sharply, because it skips the restoring step. On both datasets, *Elf* takes slightly more compression time than Chimp$_{128}$, but much less than Snappy. Besides, although *Elf* takes about double decompression time of Chimp$_{128}$, it is still less than 60μs for all values of β.

6.4 Validation of Erasing and XOR$_{cmp}$ Strategies

To verify the effectiveness of the erasing strategy, we regard *Elf* Eraser as a preprocessing operation on Gorilla, Chimp and Chimp$_{128}$.

Figures 9(a-f) present the average compression ratio improvement over the native methods in three groups of β. It is observed that:

(1) For both time series datasets and non time series datasets with small or medium β, our proposed erasing strategy can improve the compression ratio of Gorilla and Chimp dramatically. In particular, if β is small, with the equipment of *Elf* Eraser, Gorilla can obtain a relative improvement of 62.2% and 51.6% on the time series datasets and non-time series datasets, respectively, while Chimp can also enjoy a relative improvement of 56.8% and 49.5%, respectively.

(2) Chimp_{128} can be hardly enhanced by *Elf* Eraser. This is because Chimp_{128} leverages the least 14 significant mantissa bits as its hash key. After erasing the mantissa, it is hard for Chimp_{128} to find an appropriate previous value, which might result in an XORed value with a small number of leading zeros. Besides, keeping track of the positions of the chosen values consumes additional bits. As a result, unlike Chimp_{128}, *Elf* considers only the neighboring values.

(3) For datasets with large β, *Elf* Eraser cannot enhance the XOR-based compressors, because for large β, *Elf* Eraser gives up erasing to avoid a negative gain.

(4) If β is not large, *Elf* compression algorithm is even 8.7%∼33.3% better than the Eraser-enhanced Gorilla and Chimp, which verifies the effectiveness of the optimization for XOR_{cmp}.

7 RELATED WORKS

7.1 General Compression

There are a wide range of impressive compression methods for general purposes, such as Xz [12], Brotli [13], LZ4 [19], Zstd [18] and Snappy [21]. Zstd combines a dictionary-matching stage with a fast entropy-coding stage. The dictionary is trainable and can be generated from a set of samples. Snappy also refers to a dictionary and stores the shift from the current position back to uncompressed stream. Both Zstd and Snappy can achieve a good trade-off between compression ratio and efficiency. Most general compression methods are lossless and can achieve a good compression ratio, but they do not leverage the characteristics of floating-point values and cannot be applied directly to streaming scenarios [28] either.

7.2 Lossy Floating-Point Compression

Since floating-point data is stored in a complex format, it is challenging to compress floating-point data without losing any precision. To this end, many lossy floating-point compression methods are proposed [27, 35–38, 54, 55]. For example, the representative method ZFP [36] compresses regularly gridded data with a certain loss guarantee. MDZ [55] is an adaptive error-bounded lossy compression framework that optimizes the compression for two execution models of molecular dynamics. However, these lossy compression methods are usually application specific. Moreover, many scenarios, especially in the fields of scientific calculation and databases [29, 50, 52], do not tolerate any loss of precision.

7.3 Lossless Floating-Point Compression

Most lossless floating-point compression algorithms are based on prediction. The distinction among them lies in two aspects: 1) How does the predictor work? 2) How to handle the difference between the predicted value and the real one?

Based on the former aspect, lossless floating-point compression algorithms can be further divided into model-based methods [15–17, 24, 25, 47, 51] and previous-value methods [34, 46]. DFCM [47] maps floating-point values to unsigned integers and predicts the values by a DFCM (differential finite context method) predictor. However, DFCM only works well for smoothly changing data. FPC [16, 17] sequentially predicts each value in a streaming fashion using two context-based predictors, i.e., FCM predictor [48] and DFCM predictor (which is quite different from that in DFCM [47]). Among the predicted values obtained by the two predictors, FPC chooses the closer one, and thus it can achieve a better prediction performance. Some other model-based methods [24, 25, 51] capture the characteristics of different series using machine learning models, and eventually choose the best compression approach. Due to the high cost of prediction, Gorilla [46] and Chimp [34] directly regard the previous one value as the predicted one, based on the observation that two consecutive values do not change much. Chimp_{128} is an upgraded version of Chimp, which exploits 128 earlier values to find the best matched value. To expedite the computation efficiency, Chimp_{128} maintains a hash table with size of 33KB, which might be not applicable in edge computing scenarios [39, 49].

Based on the latter aspect, a small number of methods [20] first map the differences between the predicted values and actual values to integers, and then compress the integers using integer-oriented compression techniques such as Delta encoding [46]. On the contrary, a majority of methods [17, 34, 46] encode their XORed values instead of the differences. Gorilla [46] assumes that the XORed values would contain both long leading zeros and long trailing zeros with high probability, so it uses 5 bits to record the number of leading zeros and 6 bits to store the number of trailing zeros. Chimp [34] points out the fact that the XORed values rarely have long trailing zeros, so it is ineffective for Gorilla to take up to 6 bits to record the number of trailing zeros. Therefore, Chimp optimizes the encoding strategy for the XORed values and can use fewer bits.

As a lossless compression solution, *Elf* belongs to a previous-value method and encodes the XORed values. However, different from Gorilla and Chimp, *Elf* performs an erasing operation on the floating-point values before XORing them, which makes the XORed values contain many trailing zeros. Besides, *Elf* designs a novel encoding strategy for the XORed values with many trailing zeros, which achieves a notable compression ratio.

8 CONCLUSION AND FUTURE WORK

This paper proposes a novel, compact and efficient erasing-based lossless floating-point compression algorithm *Elf*. Extensive experiments using 22 datasets verify the powerful performance of *Elf*. In particular, *Elf* achieves average relative compression ratio improvement of 12.4% and 43.9% over Chimp_{128} and Gorilla, respectively. Besides, *Elf* has a similar compression ratio to the best general compression algorithm with much less time. In our future work, we plan to optimize *Elf* for specific data types, such as trajectories.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (62202070, 61976168, 62172066, 62076191) and China Postdoctoral Science Foundation (2022M720567).

REFERENCES

[1] 2023. Blockchair Database Dumps. Retrieved March 19, 2023 from https://gz.blockchair.com/bitcoin/transactions/.

[2] 2023. Daily Temperature of Major Cities. Retrieved March 19, 2023 from https://www.kaggle.com/sudalairajkumar/daily-temperature-of-major-cities.

[3] 2023. Electric Vehicle Charging Dataset. Retrieved March 19, 2023 from https://www.kaggle.com/datasets/michaelbryantds/electric-vehicle-charging-dataset.

[4] 2023. Elf Floating-Point Compression. Retrieved March 19, 2023 from https://github.com/Spatio-Temporal-Lab/elf.

[5] 2023. Financial data set used in INFORE project. Retrieved March 19, 2023 from https://zenodo.org/record/3886895#.Y4DdzHZByM_.

[6] 2023. Global Food Prices Database (WFP). Retrieved March 19, 2023 from https://data.humdata.org/dataset/wfp-food-prices.

[7] 2023. Historical Weather Data Download. Retrieved March 19, 2023 from https://www.meteoblue.com/en/weather/archive/export/basel_switzerland.

[8] 2023. InfluxDB 2.0 Sample Data. Retrieved March 19, 2023 from https://github.com/influxdata/influxdb2-sample-data.

[9] 2023. Points of Interest POI Database. Retrieved March 19, 2023 from https://www.kaggle.com/datasets/ehallmar/points-of-interest-poi-database.

[10] 2023. SSD and HDD Benchmarks. Retrieved March 19, 2023 from https://www.kaggle.com/datasets/alanjo/ssd-and-hdd-benchmarks.

[11] 2023. World City. Retrieved March 19, 2023 from https://www.kaggle.com/datasets/kuntalmaity/world-city.

[12] 2023. The .xz File Format. Retrieved March 19, 2023 from https://tukaani.org/xz/xz-file-format.txt.

[13] Jyrki Alakuijala, Andrea Farruggia, Paolo Ferragina, Eugene Kliuchnikov, Robert Obryk, Zoltan Szabadka, and Lode Vandevenne. 2018. Brotli: A general-purpose data compressor. *ACM Transactions on Information Systems (TOIS)* 37, 1 (2018), 1–30.

[14] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[15] Davis Blalock, Samuel Madden, and John Guttag. 2018. Sprintz: Time series compression for the internet of things. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 3 (2018), 1–23.

[16] Martin Burtscher and Paruj Ratanaworabhan. 2007. High throughput compression of double-precision floating-point data. In *2007 Data Compression Conference (DCC'07)*. IEEE, 293–302.

[17] Martin Burtscher and Paruj Ratanaworabhan. 2008. FPC: A high-speed compressor for double-precision floating-point data. *IEEE Trans. Comput.* 58, 1 (2008), 18–31.

[18] Y Collet. 2016. Zstd github repository from facebook. Retrieved March 19, 2023 from https://github.com/facebook/zstd.

[19] Yann Collet et al. 2013. Lz4: Extremely fast compression algorithm. *code. google. com* (2013). Retrieved March 19, 2023 from https://github.com/lz4/lz4.

[20] Vadim Engelson, Dag Fritzson, and Peter Fritzson. 2000. Lossless compression of high-volume numerical data from simulations. In *Proc. Data Compression Conference*.

[21] Google. 2023. Snappy | A fast compressor/decompressor. Retrieved March 19, 2023 from https://github.com/google/snappy.

[22] Huajun He, Ruiyuan Li, Sijie Ruan, Tianfu He, Jie Bao, Tianrui Li, and Yu Zheng. 2022. TraSS: Efficient Trajectory Similarity Search Based on Key-Value Data Stores. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2306–2318.

[23] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2017. Time series management systems: A survey. *IEEE Transactions on Knowledge and Data Engineering* 29, 11 (2017), 2581–2600.

[24] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2018. Modelardb: Modular model-based time series management with spark and cassandra. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1688–1701.

[25] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2021. Scalable Model-Based Management of Correlated Dimensional Time Series in ModelarDB+. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1380–1391.

[26] William Kahan. 1996. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE* 754, 94720-1776 (1996), 11.

[27] Iosif Lazaridis and Sharad Mehrotra. 2003. Capturing sensor-generated time series with quality guarantees. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*. IEEE, 429–440.

[28] Ruiyuan Li, Jie Bao, Huajun He, Sijie Ruan, Tianfu He, Liang Hong, Zhongyuan Jiang, and Yu Zheng. 2020. Discovering real-time reachable area using trajectory connections. In *International Conference on Database Systems for Advanced Applications*. Springer, 36–53.

[29] Ruiyuan Li, Huajun He, Rubin Wang, Yuchuan Huang, Junwen Liu, Sijie Ruan, Tianfu He, Jie Bao, and Yu Zheng. 2020. Just: Jd urban spatio-temporal data engine. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1558–1569.

[30] Ruiyuan Li, Huajun He, Rubin Wang, Sijie Ruan, Tianfu He, Jie Bao, Junbo Zhang, Liang Hong, and Yu Zheng. 2021. Trajmesa: A distributed nosql-based trajectory data management system. *IEEE Transactions on Knowledge and Data Engineering* 35, 1 (2021), 1013–1027.

[31] Ruiyuan Li, Huajun He, Rubin Wang, Sijie Ruan, Yuan Sui, Jie Bao, and Yu Zheng. 2020. Trajmesa: A distributed nosql storage engine for big trajectory data. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2002–2005.

[32] Ruiyuan Li, Liang Zhang, Juan Pan, Junwen Liu, Peng Wang, Nianjun Sun, Shanmin Wang, Chao Chen, Fuqiang Gu, and Songtao Guo. 2022. Apache ShardingSphere: A Holistic and Pluggable Platform for Data Sharding. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2468–2480.

[33] Shancang Li, Li Da Xu, and Shanshan Zhao. 2015. The internet of things: a survey. *Information systems frontiers* 17, 2 (2015), 243–259.

[34] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: efficient lossless floating point compression for time series databases. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3058–3070.

[35] Xin Liang, Kai Zhao, Sheng Di, Sihuan Li, Robert Underwood, Ali M Gok, Jiannan Tian, Junjing Deng, Jon C Calhoun, Dingwen Tao, et al. 2022. SZ3: A modular framework for composing prediction-based error-bounded lossy compressors. *IEEE Transactions on Big Data* 9, 2 (2022), 485–498.

[36] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2674–2683.

[37] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J Elmore. 2021. Decomposed bounded floats for fast compression and queries. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2586–2598.

[38] Tong Liu, Jinzhen Wang, Qing Liu, Shakeel Alibhai, Tao Lu, and Xubin He. 2021. High-ratio lossy compression: Exploring the autoencoder to compress scientific data. *IEEE Transactions on Big Data* 9, 1 (2021), 22–36.

[39] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. 2017. A survey on mobile edge computing: The communication perspective. *IEEE communications surveys & tutorials* 19, 4 (2017), 2322–2358.

[40] National Ecological Observatory Network (NEON). 2022. 2D wind speed and direction (DP1.00001.001). https://doi.org/10.48443/77N6-EH42 Retrieved March 19, 2023 from https://data.neonscience.org/data-products/DP1.00001.001/RELEASE-2022.

[41] National Ecological Observatory Network (NEON). 2022. Barometric pressure (DP1.00004.001). https://doi.org/10.48443/ZR37-0238 Retrieved March 19, 2023 from https://data.neonscience.org/data-products/DP1.00004.001/RELEASE-2022.

[42] National Ecological Observatory Network (NEON). 2022. Dust and particulate size distribution (DP1.00017.001). https://doi.org/10.48443/RDZ9-XR84 Retrieved March 19, 2023 from https://data.neonscience.org/data-products/DP1.00017.001/RELEASE-2022.

[43] National Ecological Observatory Network (NEON). 2022. IR biological temperature (DP1.00005.001). https://doi.org/10.48443/7RS6-FF56 Retrieved March 19, 2023 from https://data.neonscience.org/data-products/DP1.00005.001/RELEASE-2022.

[44] National Ecological Observatory Network (NEON). 2022. Relative humidity above water on-buoy (DP1.20271.001). https://doi.org/10.48443/1W06-WM51 Retrieved March 19, 2023 from https://data.neonscience.org/data-products/DP1.20271.001/RELEASE-2022.

[45] Dinh C Nguyen, Ming Ding, Pubudu N Pathirana, Aruna Seneviratne, Jun Li, Dusit Niyato, Octavia Dobre, and H Vincent Poor. 2021. 6G Internet of Things: A comprehensive survey. *IEEE Internet of Things Journal* 9, 1 (2021), 359–383.

[46] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1816–1827.

[47] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. 2006. Fast lossless compression of scientific floating-point data. In *Data Compression Conference (DCC'06)*. IEEE, 133–142.

[48] Yiannakis Sazeides and James E Smith. 1997. The predictability of data values. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 248–258.

[49] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE internet of things journal* 3, 5 (2016), 637–646.

[50] Jinzhao Xiao, Yuxiang Huang, Changyu Hu, Shaoxu Song, Xiangdong Huang, and Jianmin Wang. 2022. Time series data encoding for efficient storage: a comparative analysis in Apache IoTDB. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2148–2160.

[51] Xinyang Yu, Yanqing Peng, Feifei Li, Sheng Wang, Xiaowei Shen, Huijun Mai, and Yue Xie. 2020. Two-level data compression using machine learning in time series database. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1333–1344.

[52] Zisheng Yu, Ruiyuan Li, Yang Guo, Zhongyuan Jiang, Jie Bao, and Yu Zheng. 2021. Distributed Time Series Similarity Search Method Based on Key-value Data Stores. *Journal of Software* 33, 3 (2021), 950–967.

[53] Xianyuan Zhan, Haoran Xu, Yue Zhang, Xiangyu Zhu, Honglei Yin, and Yu Zheng. 2022. Deepthermal: Combustion optimization for thermal power generating

units using offline reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 4680–4688.

[54] Kai Zhao, Sheng Di, Maxim Dmitriev, Thierry-Laurent D Tonellot, Zizhong Chen, and Franck Cappello. 2021. Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1643–1654.

[55] Kai Zhao, Sheng Di, Danny Perez, Xin Liang, Zizhong Chen, and Franck Cappello. 2022. Mdz: An efficient error-bounded lossy compressor for molecular dynamics. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 27–40.