Chimp: Efficient Lossless Floating Point Compression for Time Series Databases

Panagiotis Liakos
Athens University of Economics and Business
Athens, Greece
panagiotisliakos@aueb.gr

Katia Papakonstantinopoulou
Athens University of Economics and Business
Athens, Greece
katia@aueb.gr

Yannis Kotidis
Athens University of Economics and Business
Athens, Greece
kotidis@aueb.gr

ABSTRACT

Applications in diverse domains such as astronomy, economics and industrial monitoring, increasingly press the need for analyzing massive collections of *time series* data. The sheer size of the latter hinders our ability to efficiently store them and also yields significant storage costs. Applying general purpose compression algorithms would effectively reduce the size of the data, at the expense of introducing significant computational overhead. Time Series Management Systems that have emerged to address the challenge of handling this overwhelming amount of information, cannot suffer the ingestion rate restrictions that such compression algorithms would cause. Data points are usually encoded using faster, streaming compression approaches. However, the techniques that contemporary systems use do not fully utilize the compression potential of time series data, with implications in both storage requirements and access times. In this work, we propose a novel streaming compression algorithm, suitable for floating point time series data. We empirically establish properties exhibited by a diverse set of time series and harness these features in our proposed encodings. Our experimental evaluation demonstrates that our approach *readily outperforms* competing techniques, attaining compression ratios that are competitive with *slower* general purpose algorithms, and on average around 50% of the space required by state-of-the-art streaming approaches. Moreover, our algorithm outperforms all earlier techniques with regards to both *compression and access time*, offering a significantly *improved trade-off* between space and speed. The aforementioned benefits of our approach –in terms of *all* space requirements, compression time and read access– significantly improve the efficiency in which we can store and analyze time series data.

PVLDB Reference Format:
Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. Chimp: Efficient Lossless Floating Point Compression for Time Series Databases. PVLDB, 15(11): 3058 - 3070, 2022.
doi:10.14778/3551793.3551852

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/panagiotisl/chimp.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551852

1 INTRODUCTION

Applications found in multiple industries, such as healthcare, finance, environmental monitoring and manufacturing, are producing high quality information in the form of time-ordered data sequences at an unprecedented rate [11, 32, 36]. Consequently, data analysts are awash in petabytes of measurements that they need to harness to deliver valuable and accurate insights. However, the sheer volume of the generated *time series* data hardens the task of storing them for subsequent analysis, as the respective space required translates to excessive storage costs [25]. One way to cope with the increasing volume of data points is to compress them, to reduce the overall space requirements, while also providing data analysts with access to all historical data. Besides the obvious storage savings, the reduced requirements would also improve the performance of read queries, as fewer disk pages need to be read and more blocks can be cached in memory. Nonetheless, general purpose compression algorithms are quite expensive and their use would significantly impede the ingestion of voluminous data sequences.

Indeed, Time Series Management Systems (TSMSs) [17], that have recently emerged due to the growing need for systems that efficiently handle the velocity and volume of time series data, predominantly *avoid* using general purpose compression algorithms. Instead, to reduce the space requirements of storing both the measurements and the respective timestamps, TSMSs adopt alternative, *faster* compression approaches. When handling floating point time series data, an effective compression technique is to perform a bitwise XOR operation between the current value and the previous value. The resulting set of bits describing the difference is likely to contain a lot of zeros, as neighboring data points do not change significantly. This technique is the basis of many existing approaches [13, 16, 24, 34], including Gorilla [33], the state-of-the-art lossless compression algorithm currently employed in most widely-used TSMSs. All these XOR-based floating point compression algorithms support compression over a *stream of data* and are fast enough to handle the ingestion rate requirements of contemporary systems. However, despite their advantages and wide adoption, their respective space savings are very *modest* and, in any case, not on par with those of general purpose compression algorithms.

In this paper, we perform a rigorous study of compression algorithms suitable for floating point data to uncover the advantages and disadvantages of different approaches. We also study various real-world floating point time series and bring to light properties that provide high compression potential. Our investigation shows that when two consecutive floating point values are not identical, their respective XORed value is not likely to have a large number of trailing zeros. On the contrary, most resulting XORed values exhibit a considerable number of leading zeros. Based on our findings, we design and propose Chimp, a novel lossless streaming compression algorithm that preserves the compression and decompression speed of earlier streaming approaches, while also providing significant space savings that are competitive with slower, yet extremely effective general purpose compression schemes.

Chimp builds upon the distribution of the number of leading zeros to provide a very space efficient representation that is frequently reused by successive values. Moreover, we induce impressive compression rates by exploiting trailing zeros *only* when their number is large enough to provide savings. Finally, we propose a variant of our algorithm, termed Chimp$_{128}$, that examines additional earlier values –besides the immediately previous one– to compare with the current value. In this way, Chimp$_{128}$ identifies values that significantly boost our compression potential and help us greatly outperform the current state-of-the-art streaming approaches. Due to our space savings and the use of a sophisticated data structure that enables us to quickly retrieve the best candidate values, Chimp$_{128}$ offers impressive performance with regard to *compression speed* as well. These findings establish our approach as the undisputed preferable option for the compression of floating point time series, as we provide space-efficiency that is equivalent with up to 48x slower general purpose compression approaches while also being at least as fast as the state-of-the-art streaming approaches. Overall, our improvements with regards to *all* compression time, decompression time, and space requirements, directly translate to significant tangible benefits for applications that work with time series data.

We summarize here the key contributions of our proposed compression algorithm. In particular, we:

- perform a survey of compression algorithms suitable for floating point data and investigate their pros and cons.
- uncover and report important properties of floating point time series towards fully exploiting their compression potential, through a highly efficient streaming approach.
- significantly reduce the space requirements for compressing floating point data, compared to state-of-the-art streaming approaches. We show that our optimized representations offer improvements that reach up to 70% over earlier approaches, requiring on average about *half of the space*.
- greatly outperform general purpose algorithms that provide better compression ratio by offering up to 48 times faster compression. We also outperform streaming techniques in terms of speed as far as both compression and decompression of the values are concerned, offering an overall significantly improved trade-off between space and speed, which clearly stands out compared to earlier efforts, streaming or not.

2 PRELIMINARIES

We begin our discussion by providing definitions that will be useful throughout the paper in Section 2.1. Next, we detail the standard we use for the representation of floating point numbers in Section 2.2, and discuss the state-of-the-art Gorilla compression scheme for floating-point time series in Section 2.3.

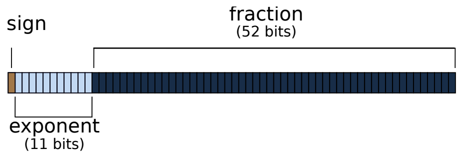

Figure 1: 64-bit IEEE 754 double precision floating point format. One bit is used for the sign, 11 bits for the exponent and 52 bits for the fraction.

2.1 Floating Point Time Series

We first provide necessary definitions in the context of our work:

Definition 2.1 (Time series). A time series TS is a sequence of data points, represented as timestamp and value pairs, ordered by time in increasing order: $TS = \langle (t_1, v_1), (t_2, v_2), \ldots \rangle$. For each pair (t_i, v_i) the timestamp t_i represents the time when the value v_i was recorded.

Definition 2.2 (Bounded Time series). A time series TS that comprises a fixed number of n data points $TS = \langle (t_1, v_1), \ldots, (t_n, v_n) \rangle$ is a *bounded time series.*

Definition 2.3 (Floating Point Time series). A time series $TS = \langle (t_1, v_1), (t_2, v_2), \ldots \rangle$ with $v_i \in \mathbb{R}$ for all $i \in \mathbb{N}$, is a *floating point time series.*

The following is an example of a floating point time series, with each pair representing the timestamp, in seconds, since the recording of measurements was initiated, and the respective values:

$$TS = \langle (0, 3.2), (5, 3.7), (10, 3.2), \ldots \rangle.$$

We can construct a bounded floating point time series by using a subset of data points from TS, for example conditioning on $0 \leq t_i \leq 100$.

In this paper our focus is on floating point time series, bounded or not, and their representation using the double precision floating point format that occupies 64 bits in memory.

2.2 IEEE 754 Double Precision Floating Point Format

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation and is currently the most common representation for real numbers on computers [5]. First established in 1985, the standard addressed the inconsistencies caused due to many diverse floating point implementations used earlier, as each computer manufacturer developed its own floating point system [31].

The standard specifies two basic formats, *single* and *double* which use a 32-bit single word and a 64-bit double word, respectively. The single word format is not adequate for many applications, due to the greater exponent range needed and -more frequently- the higher precision required [31]. Figure 1 shows the three basic components of the IEEE 754 double precision floating point data type:

- The **sign (S)** of the number: This one-bit field specifies either 0 for positive or 1 for negative.
- The **biased exponent (E)** of the number in base 2. This field contains 11 bits. The exponent is biased by $B = 1023$.

- The **fractional (F)** part of the number's significand in base 2. This field contains 52 bits.

A number x has the following form in the double-precision floating-point representation:

$$x = (-1)^S \times 2^{(E-B)} \times 1.F$$

An exponent value of 1023, i.e., equal to B, represents the actual zero. Exponents of -1023 (all 0s) and $+1024$ (all 1s) are reserved for special numbers. Thus, the biased exponent ranges from -1022 to $+1023$. The double precision floating point data type provides from 15 up to 17 significant decimal digits precision ($2^{-53} \approx 1.11 \times 10^{-16}$).

2.3 Gorilla Compression

Gorilla [33] introduces a streaming compression scheme that is suitable for double precision floating point numbers, as specified in the IEEE 754 standard. Gorilla was proposed by Facebook as part of an in-memory time series storage engine that was later open-sourced as Beringei[1] and is currently used as the default encoding for floating point measurements in most well known time series databases, including InfluxDB,[2] IotDB[3] [39], Prometheus,[4] TimeScaleDB[5] and M3.[6]

The compression scheme of Gorilla is a simplified version of the algorithm discussed in [34]. More specifically, Gorilla discards the expensive prediction phase featured in [34], and simply compares the value to be compressed with the previous value. This simplification is based on the empirical observation that real-world time series values do not change significantly when compared to their neighboring data points. Therefore, consecutive values will often have identical sign, exponent, as well as the first few bits of the significand. Gorilla leverages this empirical property by computing a simple XOR between the current and previous values, in an attempt to come up with many *successive* leading and trailing zeros. Then, Gorilla encodes these long *runs* of zeros storing only their count, followed by the raw *block of meaningful bits* in-between the leading and trailing zeros. More specifically, Gorilla uses the following variable length encoding scheme:

- The first value is stored with no compression.
- If XOR with the previous value is zero, i.e., the values are identical, store a single '0' bit.
- When XOR is non-zero, calculate the number of leading and trailing zeros in the XOR, store bit '1' followed by either:
  - Control bit '0': If the block of meaningful bits falls within the block of previous meaningful bits, i.e., there are at least as many leading zeros and as many trailing zeros as with the previous value, use that information for the block position and just store the meaningful XORed value.
  - Control bit '1': Store the length of the number of leading zeros in the next 5 bits, then store the length of the meaningful XORed value in the next 6 bits. Finally store the meaningful bits of the XORed value.

[1]https://github.com/facebookarchive/beringei
[2]https://github.com/influxdata/influxdb
[3]https://github.com/apache/iotdb
[4]https://github.com/prometheus/prometheus
[5]https://github.com/timescale/timescaledb
[6]https://github.com/m3db/m3

Algorithm 1: Gorilla

```
Function Gorilla (value, s)
    if $first = True$ then
        s.write_bits(value, 64);
        $s.first \leftarrow False$;
    else
        $xored\_value \leftarrow value \oplus s.pr\_value$;
        if $xored\_value = 0$ then
            s.write_bit(0);
        else
            s.write_bit(1);
            $lead \leftarrow count\_leading(xored\_value)$;
            $trail \leftarrow count\_trailing(xored\_value)$;
            if $lead \geq s.pr\_lead$ & $trail \geq s.pr\_trail$ then
                s.write_bit(0);
                $center\_bits \leftarrow 64 - s.pr\_lead - s.pr\_trail$;
                s.write_bits(value[s.pr_lead:-s.pr_trail], center_bits);
            else
                s.write_bit(1);
                s.write_bits(lead, 5);
                $center\_bits \leftarrow 64 - lead - trail$;
                s.write_bits(center_bits, 6);
                s.write_bits(xored_value»trail, center_bits);
                $s.pr\_lead \leftarrow lead$;
                $s.pr\_trail \leftarrow trail$;
    s.$pr\_value \leftarrow value$;
```

This operation is detailed in Algorithm 1. We note that Gorilla uses flags '0', '10', and '11' to signify three different cases. Thus, Gorilla clearly favors the case of consecutive identical values, as the respective flag, i.e., '0', uses a single bit. Moreover, Gorilla attempts to *jointly* exploit similarities in the number of leading and trailing zeros of consecutive XORed values. Naturally, such a case would work best when the expected number of *both* leading and trailing zeros is large and relatively stable. In what follows, we perform a rigorous study to investigate whether these design principles of Gorilla are aligned with properties exhibited by a diverse set of real-world and synthetic time series datasets.

3 PROPERTIES OF REAL-WORLD TIME SERIES

The original focus of Gorilla is on compressing time series values that could be *both* floating point or integer numbers. Yet, there exist much more efficient schemes for compressing integer time series, such as: (i) Delta encoding, (ii) Delta-of-delta encoding, (iii) Run-length encoding, and (iv) Simple-8b [7]. Thus, TSMSs opt to exploit the aforementioned algorithms when dealing with integer numbers and achieve greater savings than what the Gorilla scheme can offer [8]. In the context of TSMSs, Gorilla is *exclusively* used for compressing floating point data. Also, similarly to other time-aware schemes [22], timestamps are compressed *separately*.

Here, we study various diverse and predominantly real-world time series to investigate the properties that floating point values exhibit, and whether these properties are inline with the design principles of Gorilla. The dataset we use comprises the fourteen time series listed in Table 2 and is detailed in Section 5.1.

3.1 Trailing Zeros

Figure 2 shows the distribution of the number of trailing zeros that result when performing bitwise XOR between two consecutive

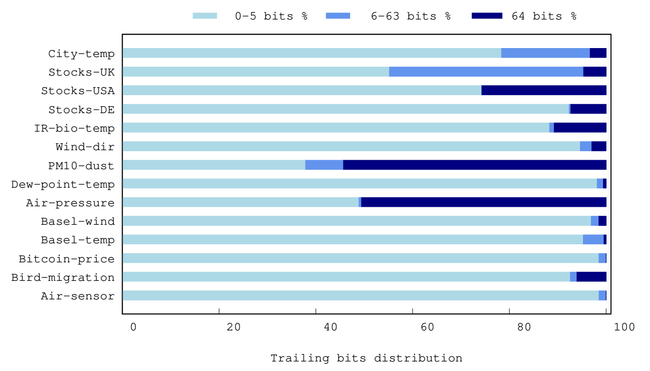

Figure 2: Average number of trailing zeros in the XOR with the previous value for various time series datasets.

0.2: 00111111 11001001 10011001 10011001 10011001 10011001 10011001 10011010
0.4: 00111111 11011001 10011001 10011001 10011001 10011001 10011001 10011010
0.8: 00111111 11101001 10011001 10011001 10011001 10011001 10011001 10011010
2.2: 01000000 00000001 10011001 10011001 10011001 10011001 10011001 10011010

(a) Common trailing bits with 0.2

0.002: 00111111 01100000 01100010 01001101 11010010 11110001 10101001 11111100
0.004: 00111111 01110000 01100010 01001101 11010010 11110001 10101001 11111100

(b) Common trailing bits with 0.002

Figure 3: Floating point numbers for which XOR would result in many trailing zeros.

values for all the time series of our dataset. We observe that with the exception of two time series, namely PM10-dust and Air-pressure, it is rather unlikely for two consecutive values to be identical. For most of the time series examined there are limited cases of resulting XORed values with 64 trailing zeros. Moreover, we observe that with very high probability the resulting values have less than six trailing zeros. Therefore, the design choice of Gorilla to reserve six bits to denote the number of trailing zeros,[7] actually *increases* the space requirements of simply storing the actual bits. Finally, there are only three datasets for which there is a significant number of cases of trailing zeros in the range $6 - 63$, i.e., City-temp, Stocks-UK, and PM10-dust. The first two use a single decimal digit and, as we can see in Figure 3a, there are many cases in which the XOR of different values with this precision causes a considerable number of trailing zeros in that range. PM10-dust uses 3 decimal digits, but certain fractional parts, such as the ones of Figure 3b, are much more frequent than others in this dataset. In most of the time series of Figure 2, we can clearly see that there are very few cases of resulting values with trailing zeros in the range $6 - 63$.

3.2 Leading Zeros

The first twelve digits of a double floating point number represent the sign and exponent, as we can see in Figure 1. Therefore, we expect measurements whose values do not change significantly to have many common leading bits. Figure 4 depicts the distribution

[7]Gorilla implicitly denotes the number of trailing bits by storing the number of meaningful bits (Line 21 of Algorithm 1).

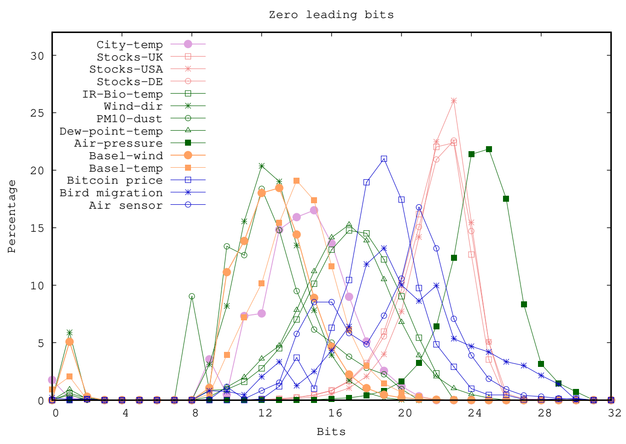

Figure 4: Distribution of the number of leading zeros in the XOR with the previous value for various time series datasets. The resulting values exhibit a considerable amount of leading zeros, with very limited cases in the range 1-7.

of the number of leading zeros that result when applying bitwise XOR between two consecutive values for all the time series of our dataset. As we can see, for most time series the result usually has at least 12 leading zeros, meaning that both the sign and the exponent of the XORed values are identical. This is not the case for City-temp, PM10-dust, and Wind-dir that often exhibit smaller runs of leading zeros, in the range $8 - 12$. This means that the exponent is not identical, but quite similar, as a few of its less significant bits are different. We also notice in Figure 4 that Air-pressure is the only dataset whose successive values produce a large number of leading zeros when XORed. This is due to the large integer part the values of this dataset have, which causes similarity between consecutive values with regards to *all* sign, exponent, and the first bits of the significand. However, even for the Air-pressure dataset, XORed values rarely have more than 30 leading zeros.

3.3 Revisiting Gorilla Compression

The investigation we performed using various diverse time series uncovered several properties that allow for effective compression. Here, we discuss whether the design principles of Gorilla exploit these properties sufficiently.

3.3.1 Flag Bits. Figure 5 shows the three different cases of XORed values that Gorilla considers, along with their respective flag bit sequences, i.e., ‘0’, ‘10’, and ‘11’. The use of a single ‘0’ bit represents the case of a zero XORed value, that results when two consecutive measurements are identical. However, the results shown in Figure 2 indicate that this is usually not the most frequent case. In fact, we see that there are only two datasets in which the frequency of zero XORed values is larger than 30%. Therefore, a different assignment of flags that would use the smallest bit sequence for a more frequent case would be beneficial towards obtaining an improved compression ratio.

3.3.2 Length of Meaningful XORed Value (Center Bits). In the rightmost case of Figure 5, applied to non-zero XORed values whose

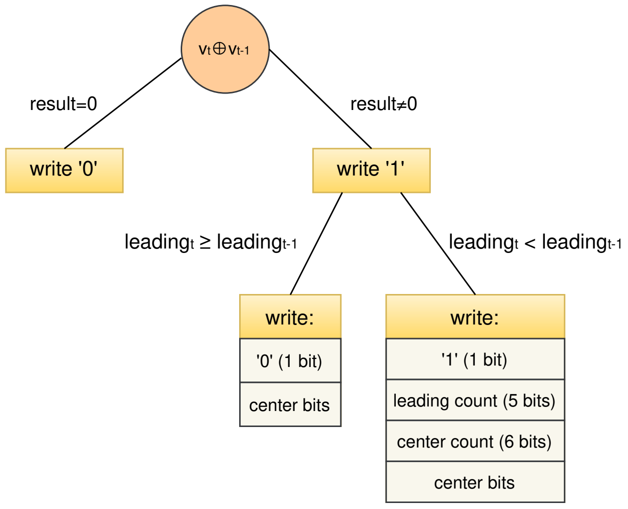

Figure 5: Gorilla compression algorithm.

block of meaningful bits does not fall within the block of previous meaningful bits, Gorilla stores the length of the meaningful XORed value using 6 bits. This allows for omitting all trailing zero bits, whose number can be derived from the length of leading zero bits and the length of the meaningful XORed value. However, Figure 2 shows that when two consecutive values are not identical, the number of trailing zeros in the XORed values is usually less than 6. Therefore, this strategy of Gorilla clearly leads to spending more bits than what the respective projected savings actually are. If we explicitly stored the trailing zeros when their number is small, we would be able to save these 6 bits, as we could derive the length of the meaningful XORed value directly from the number of leading zeros.

3.3.3 Previous Block Position. In the bottom-left case of Figure 5, used when the block of meaningful bits of a non-zero XORed value falls within the block of previous meaningful bits, Gorilla uses the existing information for the block position and just stores the meaningful XORed value. More often than not, however, the value to be stored has more leading or trailing zeros than the previous block position specifies.

Table 1 reports for every time series of our dataset the average total leading and trailing zero bits, along with averages of zero bits that this strategy fails to utilize. We see that a considerable percentage of both leading and trailing zero bits is not exploited. With regard to leading zero bits, the loss due to the grouping of meaningful bits that Gorilla applies, is at least as equal to the space required to specify the actual number of leading zeros, i.e., 5 bits, for 11 out of 14 datasets. Moreover, in the last row of Table 1 we see that this strategy leads Gorilla to explicitly store on average 6.49 leading zero bits, a figure that is larger than the prospective gain.

Table 1 also shows that for 13 out of 14 time series *almost all* trailing zero bits are not utilized. This is particularly important for City-temp and PM10-dust that exhibit a large average number of trailing zero bits and thus, the respective unexploited potential is substantial. The Stocks-UK time series stands out in Table 1, as Gorilla generally manages to take advantage of its trailing bits.

Table 1: Average leading and trailing bits, and the respective bit that remain unexploited due to the grouping of meaningful bits that Gorilla uses.

Dataset	Av. leading bits		Av. trailing bits	
	total	**unexploited**	**total**	**unexploited**
City-temp	14.18	8.85	9.20	9.19
Stocks-UK	21.71	5.38	17.26	2.44
Stocks-USA	22.14	4.62	0.99	0.96
Stocks-DE	21.86	4.84	1.1	1.09
IR-bio-temp	16.84	7.45	1.28	1.27
Wind-dir	11.95	7.27	2.04	2.03
PM10-dust	12.52	4.36	7.69	7.36
Dew-point-temp	16.59	5.54	1.19	1.19
Air-pressure	24.62	6.3	1.11	1.09
Basel-wind	12.19	10.71	1.07	1.06
Basel-temp	13.69	8.01	2.31	2.31
Bitcoin-price	18.9	5.25	1.07	1.07
Bird-migration	20.05	7.23	1.02	1.01
Air-sensor	19.27	5.0	0.99	0.99
Average	17.61	6.49	3.45	2.36

However, we must note that there are large parts of Stocks-UK that have only 0.5 precision, which causes many resulting XORed values with 40 or more trailing zero bits. For the rest of Stocks-UK values that have 0.1 precision, the trailing zero bits that are not utilized are almost equal to the total trailing zero bits, similarly to all other datasets. Therefore, the grouping of meaningful bits that Gorilla uses is not a suitable strategy for trailing zero bits either. Overall, the results of Table 1 clearly demonstrate that Gorilla's attempt to *jointly* exploit the number of both leading and trailing zeros is not particularly successful. Instead, this design choice very often leads to spending *additional space.*

4 OVERVIEW

In this section, we first provide the details of our novel compression algorithm. Then, we investigate how we can further exploit the previously encountered values of a time series and propose a variant of our algorithm, that further improves its efficiency.

4.1 Our Chimp Algorithm

We discuss here our Chimp[8] algorithm for compressing floating point time series. Similar to Gorilla, Chimp is a lossless streaming compression algorithm, suitable for settings in which high throughput rates are essential, as in the case of TSMSs. The latter have widely adopted Gorilla as the de facto standard for compressing floating point time series, even though Gorilla's original scope was both integer and floating point data. In the following we provide the details of Chimp and discuss its novel key aspects that better exploit the properties of floating point time series.

4.1.1 Possible Flag Sequences. Our empirical investigation showed that identical consecutive measurements are not very frequent in

[8]Chimps are smaller in size compared to Gorillas, and they are known to have bigger brains, which makes them more intelligent.

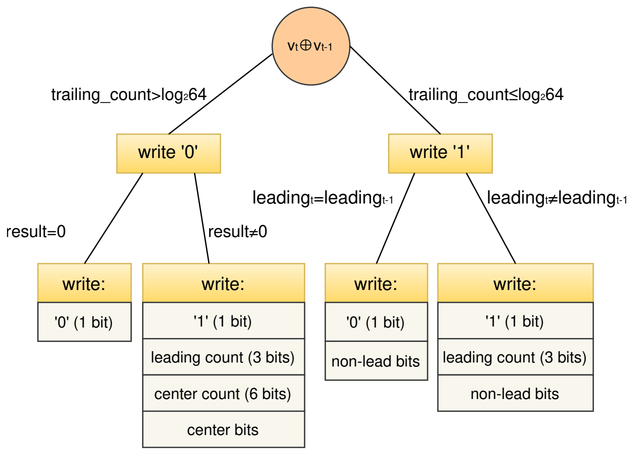

Figure 6: Chimp compression algorithm.

floating point time series. Thus, the design choice of Gorilla to use a smaller flag sequence to signify this case is not particularly effective when focusing exclusively on floating point data. We considered assigning a smaller flag sequence to a more frequent case. However, differentiating between 4 cases instead of 3, opens up further opportunities for significant savings. Therefore, we designed Chimp to use a total of four 2-bit-long flag sequences. These four different cases are illustrated in Figure 6.

4.1.2 Representing the Length of Leading Zero Bits. Similarly to earlier approaches [13, 16, 24, 33, 34], Chimp uses a bitwise XOR operation to exploit similarities between consecutive measurements. More specifically, the intent of this operation is to generate a resulting value with long runs of leading and trailing zeros. Our investigation over a diverse set of predominantly real-world floating point time series, showed that the resulting XORed values rarely have more than 30 leading zeros. Thus, the design choice of Gorilla to use 5 bits for the representation of up to 31 leading zeros is sufficient for most cases. However, we can come up with an improved representation of the number of leading zeros using 4 bits, instead of 5, at 2-bit granularity. In particular, we use mapping $f : A \longrightarrow B$, where $A = \{0, 1, \ldots, 2^n - 1\}$, $B = \{0, 1, \ldots, 2^{n-1} - 1\}$, and $n \in \mathbb{N}$, given by

$$f(x) = \begin{cases} \frac{x}{2}, & \text{when x is even, and} \\ \frac{x-1}{2}, & \text{when x is odd} \end{cases} \quad (1)$$

to encode the run of leading zeros using one fewer bit. In addition, we use mapping $f' : B \longrightarrow C$, where $C = \{0, 2, \ldots, 2^n - 2\}$, given by $f'(x) = 2x$, to decode the run of leading zeros. It is evident that the above mappings are lossy in the case of odd numbers, with an error that is always equal to 1. Therefore, to preserve all information for odd numbers, we have to encode a zero bit along with the remaining XORed value. Despite this fact, this representation is always at least as compact as that of Gorilla, as shown below.

Theorem 1. Given a 64-bit sequence s to be represented as a run of its leading zeros followed by its raw remaining bits, encoding the run using $n - 1$ bits at 2-bit granularity is at least as space-efficient as encoding it using n bits at 1-bit granularity.

Proof. Let's assume that s is even. Using Eq.(1) we can represent without error any even whole number smaller than 2^n using $n - 1$ bits. Thus, by encoding the run of leading zeros with $n - 1$ bits at 2-bit granularity instead of n bits at 1-bit granularity, we save exactly one bit without increasing the length of the remaining bits. Now, let's assume that s is odd. Using Eq.(1), we can encode any odd natural number smaller than 2^n using just $n - 1$ bits, with error ϵ that is always equal to 1. To obtain a lossless representation, we store the extra zero together with the remaining bits. This will increase the length of the remaining bits by exactly one bit. However, encoding the run using $n - 1$ bits instead of n bits retains the overall space requirements unchanged. □

We can further improve the space efficiency of the leading zeros representation by exploiting their distribution, shown in Figure 4. We observe that small values of runs are rarely encountered. Thus, our final design for Chimp uses only 3 bits to represent up to 24 leading zeros with an *exponentially decaying step* between the mapped values. The actual steps used are 0, 8, 12, 16, 18, 20, 22, 24 and provide Chimp with improved compression gains compared to the 4-bit representation with 2-bit granularity for *all* time series of our dataset. More specifically, the 4-bit representation with 2-bit granularity offers an average compression gain of 0.51 bits per value, whereas the 3-bit representation that uses steps with exponential decay provides an average gain of 0.95 bits per value.

4.1.3 Specifying the Length of the Meaningful XORed Value. Our rigorous study of consecutive XORed values revealed that long runs of trailing zeros are *rather rare*. Given the above observation, Chimp differentiates between resulting XORed values, depending on whether their number of trailing zeros exceeds a certain threshold, as we see in Figure 6. We set this threshold to 6, i.e., the bit length necessary to specify the maximum number of meaningful bits in any XORed value. Therefore, when the number of trailing zeros is less than or equal to this threshold, we write the entire XORed value, except its leading zero bits. As the number of trailing zeros is at most 6 in these cases, it is always cheaper to include the zeros in the meaningful part of the XORed value and avoid wasting 6 bits to explicitly specify their length.

4.1.4 Using the Previous Number of Leading Zero Bits. Table 1 shows that the grouping of consecutive XORed values with respect to the common part of their meaningful bits, that Gorilla applies to avoid specifying its position, is not very effective. Chimp limits the encoded representation of the number of leading zeros to just 3 bits. Moreover, we specify the number of meaningful bits only when it exceeds the bits we need to spend to do so. Still, there exist cases in which we can capitalize on similarities with regard to the number of leading zeros in consecutive XORed values to induce further savings.

Chimp considers two cases of XORed values whose number of trailing zeros is less than a certain threshold. When a value's number of leading zeros is *exactly equal* to the previous value's number of leading zeros, we can simply write a single '0' control bit, instead of spending 3 bits to specify this number again. Otherwise, we write control bit '1' as well as the number of leading zeros. We note here that using our 3-bit representation of leading zero bits, different numbers are mapped to the same value, e.g., 16 and 17

Algorithm 2: Chimp

```
1  Function Chimp(value, s)
2      if $s.first = True$ then
3          s.write_bits(value, 64);
4          $s.first \leftarrow False$;
5      else
6          $xored\_value \leftarrow value \oplus s.pr\_value$;
7          $lead \leftarrow bin\_count\_leading(xored\_value)$;
8          $trail \leftarrow count\_trailing(xored\_value)$;
9          if $trail > 6$ then
10             s.write_bit(0);
11             if $xored\_value = 0$ then
12                 s.write_bit(0);
13             else
14                 s.write_bit(1);
15                 s.write_bits(lead, 3);
16                 $center\_bits \leftarrow 64 - lead - trail$;
17                 s.write_bits(center_bits, 6);
18                 s.write_bits(xored_value»trail, center_bits);
19         else
20             s.write_bit(1);
21             if $lead = s.pr\_lead$ then
22                 s.write_bit(0);
23                 s.write_bits(xored_value, 64-lead);
24             else
25                 s.write_bit(1);
26                 s.write_bits(lead, 3);
27                 s.write_bits(xored_value, 64-lead);
28         $s.pr\_lead \leftarrow lead$;
29     $s.pr\_value \leftarrow value$;
```

are both mapped to 100 and are thus, considered equal. This favors the first case of our strategy, and therefore, our overall approach becomes very effective in terms of exploiting leading zero bits.

4.1.5 Detailed Chimp Compression. Chimp performs a bitwise XOR between the current and previous value and encodes the result with the following variable length encoding scheme.

- The first value is stored with no compression.
- When XOR with the previous value has more than 6 trailing zeros, store a single '0' bit followed by either:
  - Control bit '0': If the result is zero, i.e., the values are identical.
  - Control bit '1': If the result is not zero, store the length of the number of leading zeros in the next 3 bits, then store the length of the meaningful XORed value in the next 6 bits. Finally store the meaningful bits of the XORed value.
- When XOR has 6 or less trailing zeros, store a single '1' bit followed by either:
  - Control bit '0': If the number of leading zeros is exactly equal to the previous leading zeros, use that information and just store the meaningful XORed value.
  - Control bit '1': Store the length of the number of leading zeros in the next 3 bits, then store the length of the meaningful XORed value in the next 6 bits. Finally store the meaningful bits of the XORed value.

This operation is portrayed with the diagram of Figure 6 and detailed in Algorithm 2. To come up with a single precision variation of Chimp we simply substitute 64 with 32 in Lines 3, 16, 23, and 27, as well as 6 with 5 in Lines 9 and 17.

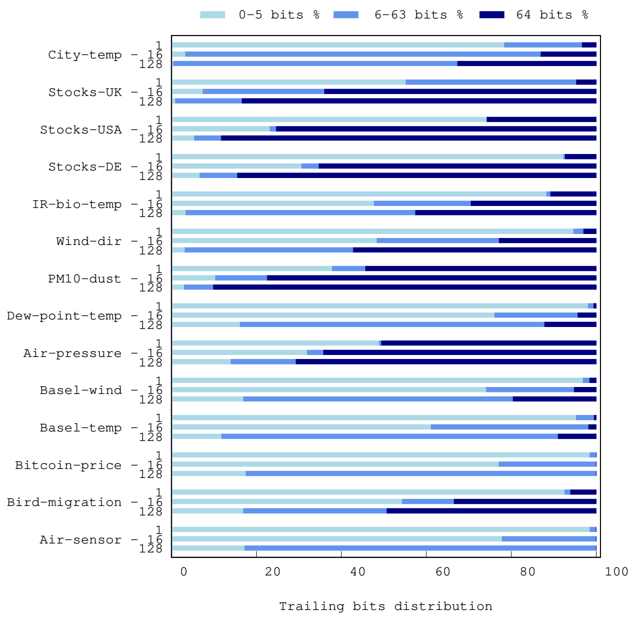

Figure 7: Average number of trailing zeros in the XOR with the best of 1, 16, and 128 previous values, respectively, for various time series datasets.

4.2 Exploiting Earlier Values with Chimp$_{128}$

An important aspect that impacts the effectiveness of Chimp is the similarity of consecutive values. In the case of identical values, Chimp spends only two flag bits to represent the second instance. Equivalently, when the similarity between the values causes a XORed result with many leading and trailing zeros, Chimp is able to reduce the overall space requirements significantly. However, our investigation and the results of Figure 2 in particular, indicate that such cases are not very common, especially when working with measurements of high precision. Even the slightest changes generate very different binary representations.

In Figure 7, we perform the same investigation computing the bitwise XOR operation with the last 16 and 128 values, respectively, instead of using just the immediately previous value. We then pick the value that produces the most trailing zeros. We remind the reader that Gorilla uses 6 bits to denote the number of trailing zeros, so we report the percentage of values below and above this threshold, together with identical values (64 bits). It is evident from the figure that there is great potential with regard to finding similar measurements when exploiting more than one previous values. Naturally, as the number of earlier values examined grows, so does the probability of finding good matches for the current measurement. The improvement is evident in most time series when using just 16 previous values. Setting this number to 128, results in very few cases of XORed values with less than 6 trailing zeros for all the time series of our dataset. On the contrary, we very often find an identical value, or –at the very least– a large number of trailing zeros.

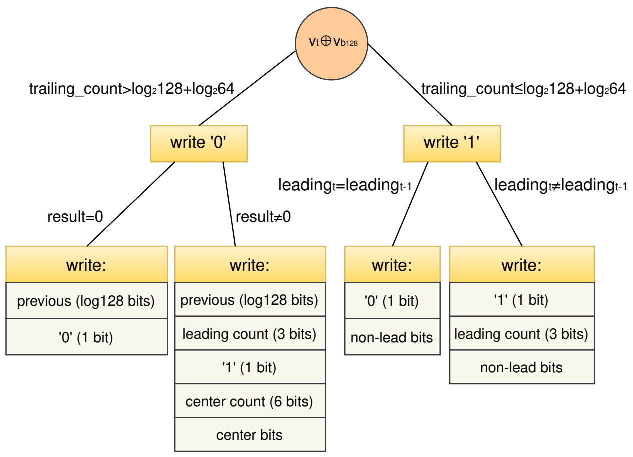

Figure 8: Chimp$_{128}$ compression algorithm.

The diagram of Figure 8 illustrates the encoding scheme of Chimp$_{128}$, that make use of 128 previous values, instead of a single one. The bitwise XOR operation in Chimp$_{128}$ is performed between the current value v_t and the best of 128 previous values in terms of most trailing zeros, $v_{b_{128}}$. If the resulting number of trailing zeros *surpasses* the number of bits needed to denote the previous value used ($log_2 128$ bits) plus the number of bits required to specify the number of meaningful bits ($log_2 64$ bits), then we make use of and actually store the previous value used (two bottom-left cases of Figure 8). Otherwise, the use of $v_{b_{128}}$ is not particularly useful and, due to the flexibility of Chimp, we can use the immediately previous value v_{t-1} instead, and avoid wasting additional bits to denote the previous value used (two bottom-right cases of Figure 8).

Despite the impressive compression potential of using previous values, performing a considerable number of XOR operations to find the best match is costly. We are interested, however, in providing compression that is fast enough to cope with the ingestion rates that contemporary time series databases need to handle. To this end, Chimp$_{128}$ uses a circular (ring) buffer of size 128 to hold the most recent values and an array of size 2^{14} ($2^{log_2 64 + log_2 128 + 1}$) to quickly come up with a suitable previous value. More specifically, we place every value v_i we encounter in the $i\%128$ position of the ring buffer. We also place i in the v_i & $(2^{14}-1)$ position of the array. That is, we use the less significant bits of v_i to come up with the position in the array. In this way, while compressing a new value v_j, we can retrieve in constant time the most recent value already encountered with at least 14 identical trailing bits, by looking in the v_j & $(2^{14}-1)$ position of the array. If this value is within the 128 previous data points, i.e., if $j - i \leq 128$, we can use it to compress our new value. Even though this approach evicts some of the previous values examined, we will show in our experimental evaluation that our respective compression ratio loss is negligible. On the contrary, the compression time speed-up gains are significant. Moreover, the respective $33KB$ of memory requirements are very modest.

5 EXPERIMENTAL EVALUATION

We implemented our compression algorithm using Java and tested its performance against 10 approaches suitable for lossless or lossy floating point data compression. Our implementation as well as reproducible tests are publicly available.[9] This repository also features reproducible tests for most of the alternative approaches, namely Xz [1], Brotli [6], LZ4 [2], Zstd [4], Snappy [3], FPC [9], Gorilla [33], PMC-MR [21] and Swing [12]. Additionally, we have extended the widely used InfluxDB time series database that originally uses the Gorilla compression scheme to also feature Chimp.[10] In this section, we first present the dataset and technical details on our experiments. Then, we proceed with the evaluation of our algorithm by answering the following questions:

- What are the space requirements of Chimp compared to earlier approaches?
- Is Chimp as fast as other streaming compression algorithms with regards to compression and access time?
- How does the trade-off of Chimp between compression ratio and compression time compare to other approaches?
- Does the single precision variation of Chimp provide compression savings?
- How does Chimp compare with lossy approaches?

Table 2: Details of our 14 time series (first) and 5 non time series datasets (last).

	Dataset	Size	Timespan	Decimal Digits	Source
Time series	City-temp	2,905,887	25 years	1	UDayton
	Stocks-UK	115,146,731	1 year	1	INFORE
	Stocks-USA	374,428,996	1 year	2	INFORE
	Stocks-DE	45,403,710	1 year	3	INFORE
	IR-bio-temp	380,817,839	7 years	2	NEON
	Wind-dir	199,570,396	6 years	2	NEON
	PM10-dust	222,911	5 years	3	NEON
	Dew-point-temp	5,413,914	3 years	3	NEON
	Air-pressure	137,721,453	6 years	5	NEON
	Basel-wind	124,079	14 years	7	meteoblue
	Basel-temp	124,079	14 years	9	meteoblue
	Bitcoin-price	2,741	1 month	4	InfluxDB
	Bird-migration	17,964	1 year	5	InfluxDB
	Air-sensor	8,664	1 hour	17	InfluxDB
General	Food-prices	2,050,638	-	4	WFP
	POI-lat	424,205	-	5	Kaggle
	POI-lon	424,205	-	5	Kaggle
	Blockchain-tr	231,031	-	4	Blockchair
	SD-bench	8,927	-	1	Kaggle

5.1 Experimental Setting

We ran our experiments on a computer running Xubuntu 20.04 with an Intel® Core^{TM} i5-4590, with a CPU frequency of 3.30GHz and a 6MB L3 cache, and a total of 16GB DDR3 1600MHz RAM. In all experiments we set the block size to 1,000 data points. For the general purpose compression algorithms, we use the default settings of the open source HBase implementations.[11]

[9]https://github.com/panagiotisl/chimp
[10]https://github.com/panagiotisl/influxdb
[11]https://github.com/apache/hbase/tree/master/hbase-compression

Our dataset consists of 14 time series and 5 non time series datasets from diverse sources. Even though real-world floating point measurements often have precision of one or two decimal digits, we include in our experiments time series with higher precision to showcase the wide applicability of our approach. The properties of the time series used in our experiments are listed in Table 2 and are thoroughly discussed below:

- **City-temp**: Dataset created by the University of Dayton with the temperature of major cities of the world.[12]
- **Stock Exchange Datasets**: Time series containing stock exchange price data used in INFORE project.[13] We extracted a total of three time series:
  - **Stocks-UK**: UK stocks with a single decimal digit.
  - **Stocks-USA**: USA stocks with two decimal digits.
  - **Stocks-DE**: German stocks with three decimal digits.
- **NEON datasets**: Below we list several datasets made publicly available by The National Science Foundation's National Ecological Observatory Network (NEON).
  - **IR-bio-temp**: Infrared biological temperature, i.e., surface temperature[29].
  - **PM10-dust**: Near real-time measurements of PM10 in the atmosphere using a optical sensor[28].
  - **Air-pressure**: Barometric pressure corrected to sea level and surface level[27].
  - **Dew-point-temp:** Relative dew point temperature. Observations are made by sensors located on the buoy in lakes and rivers[30].
  - **Wind-dir:** Two-dimensional wind direction[26].
- **meteoblue datasets**: Basel, Switzerland historical weather data, that are available without any restrictions.[14]
  - **Basel-wind**: Wind-speed data points for Basel.
  - **Basel-temp**: Temperature measurements for Basel.
- **InfluxDB datasets**: Datasets made available by InfuxDB, that uses Gorilla.[15]
  - **Bitcoin-price**: Bitcoin-US dollar exchange rate data from the CoinDesk API.
  - **Bird-migration**: Sample data adapted from the Movebank: Animal Tracking dataset,[16] representing animal migratory movements throughout 2019.
  - **Air-sensor**: Synthetic dataset with random noise, containing sample air sensor data.
- **Non time series datasets**:
  - **Food-prices**: Global Food Prices data from the World Food Programme for December 2020.[17]
  - **POI-lat / POI-lon**: Points of interest derived by parsing Wikipedia.[18]
  - **Blockchain-tr**: Bitcoin transactions' values for a single day (2022-03-26).[19]
  - **SD-bench**: SSD & HDD benchmark scores.[20]

[12]https://www.kaggle.com/sudalairajkumar/daily-temperature-of-major-cities
[13]https://zenodo.org/record/3886895#.YbM_6ZuxVqu
[14]https://www.meteoblue.com/en/weather/archive/export/basel_switzerland
[15]https://github.com/influxdata/influxdb2-sample-data
[16]https://www.kaggle.com/pulkit8595/movebank-animal-tracking
[17]https://data.humdata.org/dataset/wfp-food-prices
[18]https://www.kaggle.com/datasets/ehallmar/points-of-interest-poi-database
[19]https://gz.blockchair.com/bitcoin/transactions/
[20]https://www.kaggle.com/datasets/alanjo/ssd-and-hdd-benchmarks

5.2 Compression Ratio

We start our evaluation by measuring the space required to compress each of the 19 datasets. We observe in Table 3 that general purpose compression algorithms perform much better than the two earlier streaming approaches examined here, namely FPC and Gorilla. Our Chimp is always better than these two approaches, with an average relative improvement of 8.3% and 9.6%, respectively. These reduced space requirements are a result of several techniques we employ to induce savings. More specifically, Chimp does not group values with respect to the number of *both* their leading and trailing zeros, and thus, exploits *long* runs of zeros much more effectively. In addition, Chimp does not waste bits to represent trailing zeros of insignificant size, as this would increase the overall space requirements, as is the case with Gorilla. Finally, Chimp uses an improved representation of the leading zeros that uses up to two bits less than the respective representation of Gorilla.

Table 3 also shows the performance of our overall approach that also utilizes more than one previous values. In particular, we include results for Chimp_{128}, that examines 128 previous values, and selects the one that produces the most trailing zeros when performing bitwise XOR with the current value. Chimp_{128} provides impressive compression savings that are on average 39% and 40% better than FPC and Gorilla, respectively. This percentage raises up to 44% for both approaches, if we consider only the 14 time series. The slightly smaller improvement obtained for the non time series datasets is due to the absence of a time-based ordering, that would increase the usefulness of exploiting previous values. We also see that Chimp_{128} is better than two of the general purpose compression schemes, and very competitive against the rest, especially when focusing on the time series of our dataset.

We also see in Table 3 that the smallest compression gains we obtain for time series datasets are for Air-Sensor, which is a synthetic dataset. This is expected as the values are random and due to the large number of decimal digits used it is quite rare to come up with a lot of similar values in a single block. Equivalently, the 1BTC to USD exchange rate values reported in the Bitcoin-price time series, exhibit large fluctuations, which limit the compression gains we can achieve through compressing runs of trailing zeros. In fact, for both these time series the compression gains of all streaming approaches steam mostly from exploiting their leading zero bits distribution.

5.3 Compression Time

Time series databases usually write incoming data firstly in *uncompressed* format into a write-ahead log. Values are grouped into blocks and are stored in a persistent file format using a *compressed* representation at a later stage. Still, we aim for fast compression algorithms that do not cause CPU overheads, that would have an impact on the ingestion rate that a time series database would provide.

Table 4 shows the average time required for the compression of a default sized block of 1,000 values for the all time series and non time series datasets of our experimental setting. The results are averages of multiple executions. We observe that our approaches are at least twice as fast as the most efficient general purpose compression algorithm, i.e., Snappy, which is however not as effective

Table 3: Compression size results in bits/value. Chimp is always better than earlier streaming approaches. Chimp$_{128}$ exploits more previous values and provides impressive savings over streaming approaches, offering compression ratios that are competitive against general purpose encoding algorithms.

	Dataset	**General Purpose Compression**					**Streaming Compression**			
		Xz	**Brotli**	**LZ4**	**Zstd**	**Snappy**	**FPC**	**Gorilla**	**Chimp**	**Chimp$_{128}$**
Time series	City-temp	14,04	15,31	27,64	17,90	24,30	55,16	58,72	46,21	22,92
	Stocks-UK	7,61	8,54	19,84	10,32	15,80	46,15	33,45	31,27	16,70
	Stocks-USA	7,19	8,11	18,16	9,92	14,68	36,02	36,43	34,67	12,06
	Stocks-DE	8,80	9,96	20,63	12,06	16,83	44,54	45,63	42,88	13,46
	IR-bio-temp	13,82	16,05	29,13	20,19	25,58	48,52	50,33	46,39	18,94
	Wind-dir	12,66	14,98	26,95	17,88	22,02	58,12	59,62	54,31	19,80
	PM10-dust	6,55	7,21	15,03	8,50	12,52	27,79	26,91	24,40	13,64
	Dew-point-temp	20,92	25,16	38,34	29,65	38,60	53,63	54,42	51,57	32,49
	Air-pressure	14,35	14,96	21,66	17,23	21,61	24,07	23,96	22,92	19,23
	Basel-wind	36,77	38,93	44,20	38,96	47,57	58,75	63,63	56,09	45,65
	Basel-temp	22,22	25,10	34,67	26,06	34,51	57,58	60,19	54,10	30,12
	Bitcoin-price	40,29	46,46	55,20	47,64	63,19	52,22	52,50	49,68	47,17
	Bird-migration	24,97	27,11	35,50	29,12	34,00	48,14	50,24	45,92	28,37
	Air-sensor	50,16	54,22	64,32	58,53	64,10	52,56	52,98	49,54	49,56
	Time series average	20,03	22,29	32,23	24,57	31,09	47,38	47,79	43,57	26,44
Non time series	Food-prices	16,32	17,87	27,65	19,96	26,28	43,53	37,94	27,92	24,59
	POI-lat	39,30	41,94	50,19	43,08	52,81	60,65	65,95	57,80	47,71
	POI-lon	43,97	46,46	54,16	47,67	57,24	63,77	66,07	62,71	54,55
	Blockchain-tr	45,00	47,82	54,90	48,50	59,11	60,10	62,83	58,25	53,16
	SD-bench	8,12	8,98	19,05	10,65	15,70	37,74	40,25	35,10	17,00
	Non time series average	30,54	32,61	41,19	33,97	42,23	53,16	54,61	48,36	39,40

Table 4: Compression and decompression time per block (1,000 values) in μsec for the time series of our experimental setting as well as our non time series datasets.

Algorithm	**Compression (μs)**		**Decompression (μs)**	
	Time series	**Non time series**	**Time series**	**Non time series**
Xz	1,679.69	1,641.85	298.67	405.59
Brotli	1,409.54	1,430.53	59.67	69.65
LZ4	1,199.04	1,188.53	25.29	25.63
Zstd	163.27	180.99	56.64	60.29
Snappy	89.56	93.43	34.17	35.12
FPC	56.05	61.73	38.03	46.50
Gorilla	39.28	43.86	31.79	34.86
Chimp	31.18	31.02	30.68	31.10
Chimp$_{128}$	35.81	42.68	28.47	34.38

in its compression ratio compared to Chimp$_{128}$. General purpose algorithms that provide better compression ratio are between 4.5 and 47.9 times slower than Chimp$_{128}$. We also see that the design choice of Gorilla to drop the expensive predictors of FPC, does speed-up its compression time. However, due to its simplicity and reduced space requirements, our Chimp is the fastest approach. The calculations it performs to produce a compressed representation have a cost similar to that of Gorilla. Yet, the resulting bit sequences of Chimp are smaller, due to its improved compression ratio. Each bit sequence is written into a bit-stream while processing the values of each block. Bit level precision writes are costly, and thus, the overall compression time of Chimp is smaller, as it writes fewer bits to this stream.

Chimp$_{128}$ selects the best of 128 previous values to use for the bitwise XOR operation with the current value. We first retrieve the most recent previous value with similar trailing bits, using the structure discussed in Section 4.2. If this value is within the last 128 previous values, we use it to perform an XOR operation with the current value. Otherwise, we use the immediately previous value. These additional actions cause Chimp$_{128}$ to spend more time than Chimp or Gorilla to calculate the compressed bit sequence of each block. However, the compression time speed-up due to the creation of smaller bit-streams is also evident in the case of Chimp$_{128}$, on account of its impressive space savings. Therefore, as we see in Table 4, Chimp$_{128}$ is faster than Gorilla with regard to compression time. This effect is less evident for the non time series datasets, as the respective space savings are smaller.

We note here, that the speed-up reported in Table 4 could not be realized without using the structure discussed in Section 4.2. Indeed, examining the use of *all* 128 previous values would lead to an average compression time per block of 125.82 μsec. At the same time, the additional compression savings would be negligible, as we would achieve on average 47% improvement over earlier streaming approaches, instead of 45%, as reported in Table 3.

5.4 Decompression Time

We continue our experimental evaluation by measuring the time required to retrieve the actual values from the compressed blocks. Table 4 reports the average time required for decompressing a

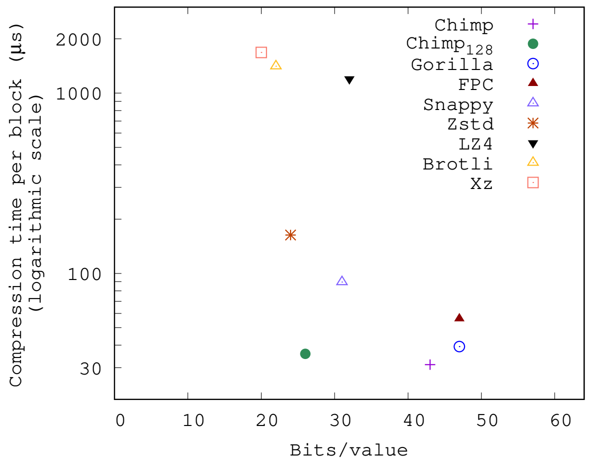

Figure 9: Trade-off between compression time (*μs* in log scale) and space (bits/value) for various algorithms. The evident superiority of Chimp$_{128}$ establishes our approach as the undisputed preferable option for compressing floating point values in the context of time series data.

default sized block of 1,000 values for all time series and non time series datasets of our experimental setting. Again, reported results are averages of multiple executions.

We observe that the LZ4 general purpose compression algorithm is the fastest approach in terms of decompression time. However, given its poor performance in terms of compression time, LZ4 is not suitable for time series compression. Our Chimp and Chimp$_{128}$ algorithms provide equivalent performance in terms of decompression time, while offering much faster compression than LZ4. Our two approaches are faster than all other algorithms listed in Table 4, even though most of them provide a satisfactory level of performance. The speed-up against streaming efforts is again attributed to the improved compression ratio of our approach. The original values of each block are reconstructed by reading the respective compressed bit sequence. Reading the sequence at bit level precision takes up a significant part of the overall decompression time. The smaller the compressed bit sequence, the faster the decompression operation.

5.5 Compression Space and Time Trade-off

Figure 9 illustrates for all algorithms of our experimental setting the trade-off between compression time and ratio achieved for the time series of our dataset. We see that Chimp surpasses the speed of streaming approaches while also providing reduced space requirements. Chimp$_{128}$ is equivalently fast and offers space savings that are competitive with significantly slower general purpose compression algorithms. The evident superiority of Chimp$_{128}$ as depicted in Figure 9 establishes our approach as the undisputed preferable option for compressing floating point values in the context of time series data.

Table 5: Compression size results for 32-bit versions of algorithms in bits/value.

Algorithm (32-bit)	Time series	Non time series
Xz	13.60	22.18
Brotli	15.36	22.26
LZ4	23.81	27.18
VictoriaMetrics	23.36	23.48
Zstd	16.76	22.85
Snappy	24.12	28.51
Gorilla	22.53	28.03
Chimp	19.60	26.80
Chimp$_{64}$	17.04	26.31

5.6 Single Precision and Lossy Compression

We conclude our experimental results by investigating the performance of our approaches using 32-bit single precision floating point numbers, and a comparison with lossy compression techniques.

Table 5 shows the space requirements of different general purpose and streaming approaches when compressing the time series and non time series datasets of our setting using single precision. As we are dealing with 32-bit instead of 64-bit numbers, we report results for Chimp$_{64}$ instead of Chimp$_{128}$, that exploits only 64 previous values. Again, we see that both Chimp and Chimp$_{64}$ outperform Gorilla, as well as three general purpose approaches. Therefore, our techniques are fully applicable on single precision floating point numbers and provide significant improvement over the state-of-the-art. Xz, Brotli and Zstd offer lower compression ratios at the expense of *prohibitively* larger CPU overhead.

Table 6 shows results for two lossy compression algorithms, namely PMC-MR and Swing. Using binary search we find and report the error ϵ required for each of the two techniques to yield lower compression ratios than what our Chimp$_{128}$ achieves. We also report the $\frac{\epsilon}{\sigma}$ percentage, where σ is the standard deviation of the values of each dataset. PMC-MR performs better than Swing. However, Table 6 clearly shows that the ϵ that both these techniques need to surpass the performance of Chimp$_{128}$ is *prohibitive*. For instance, a ±4 margin of error in temperature values, or ±31.94 in bitcoin prices would severely limit any analysis performed in the respective datasets. The performance of both techniques regarding the non time series datasets are even poorer. As an example, the required ±4 margin of error for coordinates means we can roughly tell what large state or country we are in. Therefore, such techniques are more suitable in applications such as continuous “always-on” monitoring, in which we are not interested in the actual values and instead, we simply need to guarantee a given error bound.

6 RELATED WORK

We review here existing approaches that provide lossless streaming compression of 64-bit IEEE double-precision floating-point values, appropriate for the high ingestion speed that is necessary for TSMSs. We omit discussions on earlier efforts that are not applicable in a streaming I/O setting [10, 14], and lossy approximation techniques that do not retain exact values [12, 20, 21, 23].

Table 6: Compression size results for lossy approaches in bits/value. We set the error ϵ to a value large enough to provide a better compression ratio than Chimp$_{128}$. We also report ϵ/σ for each dataset.

Dataset	PMC-MR			Swing		
	Bits/value	ϵ	ϵ/σ	Bits/value	ϵ	ϵ/σ
City-temp	13.68	4	19%	17.29	4	37%
Stocks-UK	15.66	0.25	0%	18.48	0.25	1%
Stocks-USA	7.2	0.03	1%	11.42	0.03	3%
Stocks-DE	12.43	0.015	6%	15.75	0.015	25%
Basel-temp	27.00	0.5	11%	25.56	0.25	21%
Basel-wind	40.67	0.5	7%	36.21	0.5	26%
Air-sensor	38.33	0.01	0%	35.40	0.008	0%
Bird-migration	23.11	0.016	0%	20.16	0.03	1%
Bitcoin-price	41.57	31.97	2%	42.32	31.94	8%
Air-pressure	10.18	0.004	3%	4.97	0.008	5%
Dew-point-temp	31.92	0.07	4%	34.94	0.06	18%
PM10-dust	7.84	0.004	0%	6.89	0.008	1%
IR-bio-temp	12.46	0.13	3%	10.61	0.13	7%
Wind-dir	12.46	0.25	43%	15.56	0.25	87%
Food-prices	20.85	16	0%	19.60	32	1%
POI-lat	42.87	4	19%	47.56	4	74%
POI-lon	51.78	8	11%	52.73	15.99	172%
Blockchain-tr	50.58	128	0%	51.75	128	0%
SSD-bench	9.46	253.4	47%	14.40	255.91	88%

Engelson et al. [13] predict values using polynomial extrapolation of values in previous time steps. By treating the actual and predicted floating point values as integers, they perform two's complement integer subtraction which results in a compressible run of leading zeros or ones. This run is compactly stored and is followed by the remaining raw trailing bits.

The work of Ratanaworabhan et al. [34] employs a slightly modified DFCM [15] value predictor and performs bitwise XOR between the next value to be compressed and the predicted value. The result is then encoded using fixed-width leading-zero count that is followed by the raw transmission of all trailing bits. Four-bits are used to represent the leading zero count at half-byte granularity, allowing for up to 64 leading zeros. The offered throughput rates are significantly faster than those of general purpose compressors.

The FPC [9] lossless compression algorithm, employs *two* value predictors, i.e., FPC [35] and DFCM, to come up with the best prediction of the two. As in [34], a bitwise XOR between the next value to be compressed and the predicted value is performed and the result is effectively compressed by exploiting the leading zero count. FPC additionally spends one bit for each encoded value to specify which predictor was used.

Gorilla [33] improves the efficiency of earlier floating point compression algorithms [9, 24, 34] by discarding their expensive predictions schemes. Instead of predicting the next value, Gorilla opts to simply compare the actual value with the previous one by performing a bitwise XOR operation. In the context of time series datasets, the resulting values usually exhibit long runs of leading and trailing zeros. The simplicity, throughput and compression ratio of Gorilla have helped towards its wide adoption, making it the de facto standard for compressing floating point data in TSMSs. Our approach is in turn inspired by Gorilla. However, our focus is exclusively on floating point time series, which allows for applying novel design principles and helps provide significant improvements to both compression ratio and throughput rate.

VictoriaMetrics [38] converts floating-point values to integer values by applying a 10^x multiplier and employs general purpose compression algorithms on top of the encoded data to increase its compression ratio. However, converting to integer values results in precision loss, whereas general purpose compression algorithms are slower than approaches currently used in the context of time series databases [37]. Instead, our focus is on supporting *lossless* compression over a *stream* of data.

In another line of work [18], Jensen et al. offer a multi-model compression approach that uses multiple model types to adapt as each time series' structure changes over time by choosing the most appropriate compression scheme. An extension of this work [19] additionally exploits correlation between different time series through a multi-model group compression approach that groups time series while also compressing them using multiple models. Both these approaches consider Gorilla as their algorithm of choice for lossless compression of floating point time series. Therefore, the contribution of our Chimp algorithm is orthogonal to any such approaches.

7 CONCLUSIONS AND FUTURE WORK

In this paper we introduce the Chimp lossless streaming compression algorithm for floating point time series. Similarly to earlier approaches, Chimp is based on a bitwise XOR operation between the current and previous values, as neighboring data points do not change significantly. However, here we uncover and exploit properties that floating point time series exhibit to offer a significantly improved encoding in terms of both space-efficiency and access to the compressed information. In addition, we argue that using *only* the immediately previous value severely limits the possibility of obtaining a highly compressible set of difference bits. Thus, we propose Chimp$_{128}$, a variant of our algorithm that exploits additional earlier values, and helps us achieve compression rates. The experimental evaluation of our algorithms shows that we attain a significantly lower, often *less than half*, compression ratio than earlier streaming approaches. At the same time, general purpose encodings that provide better compression ratio are between 4.5 and 47.9 times slower than our approach. In this regard, our algorithms empower the execution of algorithms on massive collections of time series data, in settings where the use of existing methods is prohibitive due to space requirements or simply too inefficient.

We plan to extend our work by investigating lossy streaming compression techniques that allow for recreating the time series within a very small error bound and further reduce the space requirements and access times.

ACKNOWLEDGMENTS

This work has received funding from the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Technology (GSRT), under grant agreement No 779.

REFERENCES

[1] 2009. The .xz File Format. (2009). Retrieved May 21, 2022 from https://tukaani.org/xz/xz-file-format.txt

[2] 2011. *LZ4*. Retrieved May 21, 2022 from https://lz4.github.io/lz4/

[3] 2011. *snappy*. Retrieved May 21, 2022 from http://google.github.io/snappy/

[4] 2015. *Zstd*. Retrieved May 21, 2022 from https://facebook.github.io/zstd/

[5] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

[6] Jyrki Alakuijala, Andrea Farruggia, Paolo Ferragina, Eugene Kliuchnikov, Robert Obryk, Zoltan Szabadka, and Lode Vandevenne. 2018. Brotli: A General-Purpose Data Compressor. *ACM Trans. Inf. Syst.* 37, 1 (dec 2018), 30. https://doi.org/10.1145/3231935

[7] Vo Ngoc Anh and Alistair Moffat. 2010. Index compression using 64-bit words. *Softw. Pract. Exp.* 40, 2 (2010), 131–147. https://doi.org/10.1002/spe.948

[8] Timescale Blog. 2020. Time-series compression algorithms, explained. https://blog.timescale.com/blog/time-series-compression-algorithms-explained/. Accessed: 2022-02-03.

[9] Martin Burtscher and Paruj Ratanaworabhan. 2007. High Throughput Compression of Double-Precision Floating-Point Data. In *2007 Data Compression Conference (DCC 2007), 27-29 March 2007, Snowbird, UT, USA*. IEEE Computer Society, 293–302. https://doi.org/10.1109/DCC.2007.44

[10] Huamin Chen, Jian Li, and Prasant Mohapatra. 2004. RACE: time series compression with rate adaptivity and error bound for sensor networks. In *2004 IEEE International Conference on Mobile Ad-hoc and Sensor Systems, Fort Lauderdale, FL, USA, October 25-27, 2004*. 124–133. https://doi.org/10.1109/MAHSS.2004.1392089

[11] Antonios Deligiannakis, Yannis Kotidis, and Nick Roussopoulos. 2007. Dissemination of compressed historical information in sensor networks. *VLDB J.* 16, 4 (2007), 439–461. https://doi.org/10.1007/s00778-005-0173-5

[12] Hazem Elmeleegy, Ahmed K. Elmagarmid, Emmanuel Cecchet, Walid G. Aref, and Willy Zwaenepoel. 2009. Online Piece-wise Linear Approximation of Numerical Streams with Precision Guarantees. *Proc. VLDB Endow.* 2, 1 (2009), 145–156. https://doi.org/10.14778/1687627.1687645

[13] Vadim Engelson, Dag Fritzson, and Peter Fritzson. 2000. Lossless Compression of High-Volume Numerical Data from Simulations. In *Data Compression Conference, DCC 2000, Snowbird, Utah, USA, March 28-30, 2000*. 574. https://doi.org/10.1109/DCC.2000.838221

[14] Florin Ghido. 2004. An Efficient Algorithm for Lossless Compression of IEEE Float Audio. In *2004 Data Compression Conference (DCC 2004), 23-25 March 2004, Snowbird, UT, USA*. IEEE Computer Society, 429–438. https://doi.org/10.1109/DCC.2004.1281488

[15] B. Goeman, H. Vandierendonck, and K. de Bosschere. 2001. Differential FCM: increasing value prediction accuracy by improving table usage efficiency. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. 207–216. https://doi.org/10.1109/HPCA.2001.903264

[16] Lawrence Ibarria, Peter Lindstrom, Jarek Rossignac, and Andrzej Szymczak. 2003. Out-of-core Compression and Decompression of Large n-dimensional Scalar Fields. *Comput. Graph. Forum* 22, 3 (2003), 343–348. https://doi.org/10.1111/1467-8659.00681

[17] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2017. Time Series Management Systems: A Survey. *IEEE Trans. Knowl. Data Eng.* 29, 11 (2017), 2581–2600. https://doi.org/10.1109/TKDE.2017.2740932

[18] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2018. ModelarDB: Modular Model-Based Time Series Management with Spark and Cassandra. *Proc. VLDB Endow.* 11, 11 (2018), 1688–1701. https://doi.org/10.14778/3236187.3236215

[19] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2021. Scalable Model-Based Management of Correlated Dimensional Time Series in ModelarDB+. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 1380–1391. https://doi.org/10.1109/ICDE51399.2021.00123

[20] Eamonn J. Keogh, Stefano Lonardi, and Chotirat (Ann) Ratanamahatana. 2004. Towards parameter-free data mining. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Seattle, Washington, USA, August 22-25, 2004*. 206–215. https://doi.org/10.1145/1014052.1014077

[21] Iosif Lazaridis and Sharad Mehrotra. 2003. Capturing Sensor-Generated Time Series with Quality Guarantees. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman (Eds.). 429–440. https://doi.org/10.1109/ICDE.2003.1260811

[22] Panagiotis Liakos, Katia Papakonstantinopoulou, Theodore Stefou, and Alex Delis. 2022. On Compressing Temporal Graphs. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 1301–1313. https://doi.org/10.1109/ICDE53745.2022.00102

[23] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. 2003. A Symbolic Representation of Time Series, with Implications for Streaming Algorithms *(DMKD '03)*. 2–11. https://doi.org/10.1145/882082.882086

[24] Peter Lindstrom and Martin Isenburg. 2006. Fast and Efficient Compression of Floating-Point Data. *IEEE Transactions on Visualization and Computer Graphics* 12, 5 (sep 2006), 1245–1250. https://doi.org/10.1109/TVCG.2006.143

[25] Yaser Mansouri, Adel Nadjaran Toosi, and Rajkumar Buyya. 2017. Data Storage Management in Cloud Environments: Taxonomy, Survey, and Future Directions. *ACM Comput. Surv.* 50, 6, Article 91 (dec 2017), 51 pages. https://doi.org/10.1145/3136623

[26] National Ecological Observatory Network (NEON). 2021. 2D wind speed and direction (DP1.00001.001). https://doi.org/10.48443/S9YA-ZC81

[27] National Ecological Observatory Network (NEON). 2021. Barometric pressure (DP1.00004.001). https://doi.org/10.48443/RXR7-PP32

[28] National Ecological Observatory Network (NEON). 2021. Dust and particulate size distribution (DP1.00017.001). https://doi.org/10.48443/4E6X-V373

[29] National Ecological Observatory Network (NEON). 2021. IR biological temperature (DP1.00005.001). https://doi.org/10.48443/JNWY-B177

[30] National Ecological Observatory Network (NEON). 2021. Relative humidity above water on-buoy (DP1.20271.001). https://doi.org/10.48443/Z99V-0502

[31] Michael L Overton. 2001. *Numerical computing with IEEE floating point arithmetic*. SIAM.

[32] Themis Palpanas. 2016. Big Sequence Management: A glimpse of the Past, the Present, and the Future. In *SOFSEM 2016: Theory and Practice of Computer Science - 42nd International Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 23-28, 2016, Proceedings*. 63–80. https://doi.org/10.1007/978-3-662-49192-8_6

[33] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *Proc. VLDB Endow.* 8, 12 (2015), 1816–1827. https://doi.org/10.14778/2824032.2824078

[34] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. 2006. Fast Lossless Compression of Scientific Floating-Point Data. In *2006 Data Compression Conference (DCC 2006), 28-30 March 2006, Snowbird, UT, USA*. IEEE Computer Society, 133–142. https://doi.org/10.1109/DCC.2006.35

[35] Y. Sazeides and J.E. Smith. 1997. The predictability of data values. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. 248–258. https://doi.org/10.1109/MICRO.1997.645815

[36] Abhishek B. Sharma, Franjo Ivancic, Alexandru Niculescu-Mizil, Haifeng Chen, and Guofei Jiang. 2014. Modeling and analytics for cyber-physical systems in the age of big data. *SIGMETRICS Perform. Evaluation Rev.* 41, 4 (2014), 74–77. https://doi.org/10.1145/2627534.2627558

[37] Aliaksandr Valialkin. 2019. VictoriaMetrics: achieving better compression than Gorilla for time series data. https://faun.pub/victoriametrics-achieving-better-compression-for-time-series-data-than-gorilla-317bc1f95932. Accessed: 2022-05-31.

[38] VictoriaMetrics. 2022. VictoriaMetrics: fast, cost-effective and scalable monitoring solution and time series database. https://github.com/VictoriaMetrics/VictoriaMetrics. Accessed: 2022-02-03.

[39] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin A. McGrail, Peng Wang, Diaohan Luo, Jun Yuan, Jianmin Wang, and Jiaguang Sun. 2020. Apache IoTDB: Time-Series Database for Internet of Things. *Proc. VLDB Endow.* 13, 12 (2020), 2901–2904. https://doi.org/10.14778/3415478.3415504