# Kangaroo: Efficient Lossless Floating-Point Compression via Dynamic Reference Selection

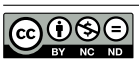
SHUO LI, Northeastern University, China
XIAOCHUN YANG*, Northeastern University, China
CHUNHUI SHEN, Alibaba Group, China
YUTONG HAN, Dalian Minzu University, China
XIANG WANG, Alibaba Group, China
LINGDU KONG, Northeastern University, China
BIN WANG, Northeastern University, China
FEIBO LI, Alibaba Group, China

To address the dual challenges of data transmission and storage in Internet of Things (IoT) systems, the development of efficient streaming lossless compression algorithms for floating-point data has become a critical research focus. Existing XOR-based floating-point compression algorithms adopt a fixed reference selection strategy by using the immediate predecessor value as the reference. Usually, this choice is not the best, as earlier values in the data stream often provide more similar and effective references. Based on this observation, we propose dynamically searching previous values to identify the reference value that yields the best compression results for the current value. To this end, we present **Kangaroo**, an efficient lossless compression algorithm for floating-point time series that implements a dynamic reference selection. Our approach begins by establishing optimal reference selection criteria aimed at minimizing the number of encoded bits, introducing a pruning-based encoding strategy to reduce encoding overhead. We further propose a dynamic search strategy that selects reference values from an extended historical window, skipping historical data that generates suboptimal XOR results. This strategy enables precise identification of the suitable reference value while improving search efficiency. Additionally, we design a bit-flip-based erasure strategy to maximize trailing zeros, thereby comprehensively enhancing compression performance. Extensive experiments on 26 datasets demonstrate that *Kangaroo* outperforms all baseline methods across all evaluation metrics. Most notably, compared to state-of-the-art streaming compression algorithms, our algorithm achieves a **23.2%** average improvement (peaking at **98.0%**) in compression ratio, with compression and decompression speeds averaging 2.13× and 2.21× (reaching up to 11.75× and 5.24×) of the baseline, respectively. This novel solution establishes comprehensive performance leadership in floating-point time-series compression, and has been applied to the Lindorm database on Alibaba Cloud since 2024, managing tens of petabytes of Internet of Vehicles (IoV) data.

CCS Concepts: • **Information systems → Data management systems**; **Data structures**; **Data layout**; **Data compression**;

*Corresponding author

Authors' Contact Information: Shuo Li, Northeastern University, Shenyang, Liaoning, China, 2410815@stu.neu.edu.cn; Xiaochun Yang, Northeastern University, Shenyang, Liaoning, China, yangxc@mail.neu.edu.cn; Chunhui Shen, Alibaba Group, Hangzhou, Zhejiang, China, tianwu.sch@alibaba-inc.com; Yutong Han, Dalian Minzu University, Dalian, Liaoning, China, hanyt@dlnu.edu.cn; Xiang Wang, Alibaba Group, Hangzhou, Zhejiang, China, wangxiang@alibaba-inc.com; Lingdu Kong, Northeastern University, Shenyang, Liaoning, China, 2210707@stu.neu.edu.cn; Bin Wang, Northeastern University, Shenyang, Liaoning, China, binwang@mail.neu.edu.cn; Feibo Li, Alibaba Group, Hangzhou, Zhejiang, China, lizi@alibaba-inc.com.




ACM 2836-6573/2026/6-ART199
https://doi.org/10.1145/3802076

Additional Key Words and Phrases: Lossless Compression; Floating Point Compression; Streaming Compression;Time Series

**ACM Reference Format:**
Shuo Li, Xiaochun Yang, Chunhui Shen, Yutong Han, Xiang Wang, Lingdu Kong, Bin Wang, and Feibo Li. 2026. Kangaroo: Efficient Lossless Floating-Point Compression via Dynamic Reference Selection. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 199 (June 2026), 26 pages. https://doi.org/10.1145/3802076

## 1 Introduction

With the rapid development of IoV, IoT, edge computing, and sensor networks, massive high-frequency floating-point time series data are increasingly being generated. Didi, a prominent mobility-as-a-service (MaaS) platform in China, processes over 15 billion data points per day, with its daily raw data volume reaching 70 terabytes (TB) [1]. Alibaba Cloud's Lindorm multi-model database handles 80TB of data streams per day, manages petabytes of IoV data storage, and simultaneously supports tens of millions of query requests per second [2–4]. In such high-throughput applications, storage costs often exceed 50% of the total computing expenditure. General compression algorithms [5–8] are usually inefficient when processing floating-point formats like IEEE 754, and they cannot satisfy the stringent requirements of real-time compression and decompression. This highlights the critical need for high-performance compression methods designed specifically for floating-point values.

Existing floating-point compression algorithms can be primarily categorized into two types: lossy compression [9–14] and lossless compression [15–24]. Lossy compression improves compression ratio at the cost of data precision, making it unsuitable for precision-sensitive applications. Therefore, lossless compression is required to ensure data integrity for computation. Among lossless compression algorithms, XOR-based algorithms [15–19] have emerged as particularly efficient solutions. The core idea is to leverage the temporal correlation between values by computing and storing their differences via XOR operations, thereby achieving more efficient compression.

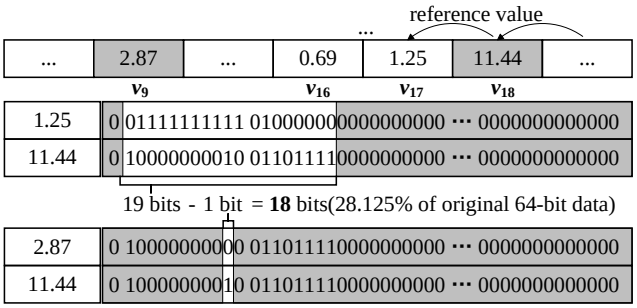



Fig. 1. Comparison of using the immediate predecessor versus a previous value as the reference, showing the IEEE 754 representations of 11.44, 1.25, and 2.87 after trailing zeros erasure.

Existing XOR-based algorithms select reference values under constrained conditions, primarily using the immediate predecessor of the current value $v_i$ for compression. Although such immediate-predecessor-based algorithms are computationally efficient, they still suffer from a fundamental limitation: they ignore the potential of previous values to serve as more effective references, thereby resulting in suboptimal compression performance. As shown in Fig. 1, the existing XOR strategy compresses the value $v_{18}(11.44)$ by referencing only its immediate predecessor $v_{17}(1.25)$. After the XOR operation, this strategy requires storing up to 19 bits of difference information. In contrast,

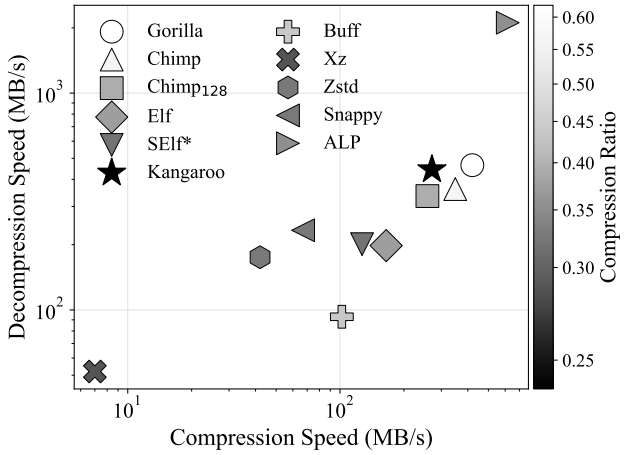



Fig. 2. Trade-offs among compression ratio, compression throughput, and decompression throughput. Darker colors represent lower compression ratios (better compression).

when using the previous value $v_9(2.87)$ as the reference, only 1 bit of difference information needs to be stored, achieving a 28.125% reduction in storage space per value.

In the field of floating-point compression, the idea of utilizing previous values has been preliminarily explored. For example, *Chimp*$_{128}$ [16] maintains a ring buffer of the most recent 128 values and uses a hash table to match the lower 14 bits for accelerated retrieval. However, this approach merely extends the window and does not fundamentally improve the criterion for reference value selection. It suffers from three main limitations: (i) it lacks a quantifiable optimality criterion for compression; (ii) its hash-based retrieval only guarantees candidate existence but not optimality; and (iii) its design is tightly coupled with value representation, limiting the integration of preprocessing techniques such as erasure strategies [17–19, 25], and consequently restricting potential performance gains.

Existing compression algorithms often face a trade-off between compression ratio, compression speed, and decompression speed. As illustrated in Figure 2, batch-based algorithms [6–8, 25] incur high computational overhead, resulting in suboptimal performance in streaming scenarios. In contrast, immediate-predecessor-based streaming algorithms [15–19], can achieve higher [de]compression speed but at the expense of compression ratios. *Chimp*$_{128}$ [16], in order to preserve compression speed, can only select a previous value that is temporally closest and has similar lower bits, rather than searching all previous values to find the globally most similar one.

To address these challenges, we propose **Kangaroo**, a novel lossless compression algorithm for floating-point time series that achieves a favorable trade-off between compression ratio and speed. *Kangaroo* first establishes an optimal reference selection criterion aimed at minimizing the number of encoded bits. Based on an analysis of redundant bit usage, we design a multi-branch encoding strategy that improves compression efficiency by selecting high-frequency branches. Furthermore, we develop a dynamic search algorithm that leverages stored historical computation information to skip suboptimal candidates during reverse traversal, enabling efficient and precise identification of the optimal reference. Finally, prior to encoding, *Kangaroo* integrates run-length encoding to optimize repeated values and introduces a bit-flip-based erasure mechanism, reducing both storage and computational overhead compared to conventional methods.

The main contributions of this work are summarized as follows:

- We propose **Kangaroo**, a novel lossless compression algorithm for floating-point time series data. *Kangaroo* establishes a quantifiable reference selection criterion that identifies the optimal reference value from previous values. Based on a detailed analysis of redundant bit overhead in encoding, we further design a pruning-based encoding strategy to achieve high compression efficiency.
- We also introduce a dynamic reference search strategy that rapidly skips suboptimal references, enabling accurate localization of the optimal reference value while improving compression speed. In addition, prior to encoding, we integrate run-length encoding and propose a bit-flip–based erasure strategy to further enhance overall compression performance.
- Extensive experiments on 26 real-world datasets show *Kangaroo* outperforms 11 baselines: it improves compression ratio by 23.2% at 2.13× (2.21×) compression (decompression) speedup over *SElf*[*]*[19]*, and improves 14.0% compression ratio with 38.71× (8.54×) speedups over *Xz[8]*.

In the rest of this paper, we first introduce the necessary preliminaries in Section 2. We then elaborate on the *Kangaroo* compression algorithm in Section 3. Section 4 presents a dynamic reference search strategy for efficient localization of the optimal reference value. Further implementation optimizations are discussed in Section 5. Experimental results and corresponding analysis are demonstrated in Section 6. An overview of related work is provided in Section 7, and we finally conclude the paper and suggest future research directions in Section 8.

## 2 Preliminaries

In Section 2.1, we introduce the IEEE 754 double-precision floating-point format. Section 2.2 provides the related definitions, and Table 1 summarizes the key notations.

Table 1. Summary of Notations

| Notation | Description |
|---|---|
| $S, E, M$ | Sign bit, exponent field, and mantissa field of a floating-point value. |
| $v_i, v_{i-1}, v_{\text{opt}}$ | Current value, immediate predecessor, and optimal reference value, respectively. |
| $xor, lead, trail, center$ | XOR result of $v_i$ and reference; the number of leading zeros, trailing zeros, and center bits. |
| $leadBits, trailBits, centerBits$ | Bits allocated for $lead$, $trail$, and $center$, respectively. |
| $index, indexBits$ | Reference value index of $v_i$, and bits allocated for storing the index. |
| $\mathcal{P}_i, \mathcal{H}_i, \mathcal{L}_i$ | Previous values, candidate references, and $lead$ of neighboring candidate references for $v_i$. |
| $h_j, \ell_j$ | Shorthand notations for $\mathcal{H}_i[j]$ and $\mathcal{L}_i[j]$, respectively. |
| $result_{en}, cost_{en}$ | Encoded representation of a value, and total number of bits required for storage, respectively. |

### 2.1 IEEE 754 Floating-Point Format

The IEEE 754 standard represents a double-precision floating-point value $v$ using 64 bits, as illustrated in Figure 3. It allocates 1 bit for the sign, 11 bits for the exponent, and 52 bits for the mantissa. For a normalized floating-point number, the value is expressed as:

$$v = (-1)^S \times 2^E \times (1 + M). \tag{1}$$

$E$ and $M$ represent the decimal form of the exponent and mantissa, respectively. For a double-precision floating-point number, the exponent is calculated as: $E = \sum_{i=1}^{11} e_i \times 2^{11-i} - 1023$, and the mantissa as: $M = \sum_{i=1}^{52} m_i \times 2^{-i}$.

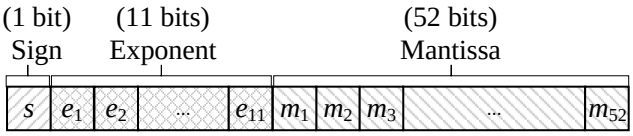



Fig. 3. Illustration of the IEEE 754 double-precision floating-point format.

## 2.2 Definitions

**Definition** 1. ***Floating-Point Time Series.*** *A floating-point time series $TS = \{(t_1, v_1), (t_2, v_2), \dots\}$ is a sequence of ordered pairs, where each $v_i \in \mathbb{R}$ is a floating-point value observed at timestamp $t_i$.*

As delta encoding and delta-of-delta encoding have demonstrated excellent performance in timestamp compression, this work focuses primarily on the compression of floating-point values. For notational simplicity, we hereafter refer to the floating-point values simply as "values".

**Definition** 2. ***Decimal Places ($\alpha$).*** *The number of digits after the decimal point in a base-10 representation of a floating-point number.*

**Definition** 3. ***Terminal Longest Strictly Increasing Subsequence (T-LSIS).*** *For a sequence $\mathcal{S} = \langle s_1, \dots, s_n \rangle$, the* T-LSIS*, denoted as T-LSIS($\mathcal{S}$), is the subsequence $\langle s_{i_1}, \dots, s_{i_k} \rangle$ ending at $s_n$ that satisfies $s_{i_j} < s_{i_{j+1}}$ for all $1 \le j < k$ and attains the maximum possible length $k$.*

**Definition** 4. ***Terminal Longest Non-decreasing Subsequence (T-LNDS).*** *Given a sequence $\mathcal{S} = \langle s_1, \dots, s_n \rangle$, the* T-LNDS*, denoted as T-LNDS($\mathcal{S}$), is the subsequence $\langle s_{i_1}, \dots, s_{i_k} \rangle$ terminating at $s_n$ where $s_{i_j} \le s_{i_{j+1}}$ holds for all $1 \le j < k$ and $k$ is maximized.*

# 3 Kangaroo

We provide the details of *Kangaroo*. Section 3.1 presents the overall workflow of the algorithm. Section 3.2 introduces the encoding cost-based criterion for optimal reference selection, followed by the proposed multi-branch encoding strategy in Section 3.3.

## 3.1 Overview

As illustrated in Figure 4, *Kangaroo* first applies Variable-Length Run-Length Encoding (VL-RLE) to the input floating-point values, converting consecutive identical values into compact <*value*, *len*> pairs. Each new run pair triggers a compression operation. For each value $v_i$ in a run pair, *Kangaroo* performs three steps. First, it applies an erasure operation that removes insignificant fractional bits to increase trailing zeros, producing an intermediate value $v_i'$. Next, it selects the optimal reference value $v_{i,\text{opt}}'$ from previous values to minimize the bitwise difference with $v_i'$. Finally, the bitwise XOR between $v_i'$ and $v_{i,\text{opt}}'$ is encoded to obtain $xor_i'$, which serves as the final compressed output. The encoded results are then stored or transmitted. Since decompression is the inverse of compression, we focus on the design and optimization of the compression process.

## 3.2 Cost Model of Reference

When selecting the optimal reference value from the previous values for $v_i$, the goal is to minimize the storage cost of the encoded result. The encoded result includes not only the number of leading

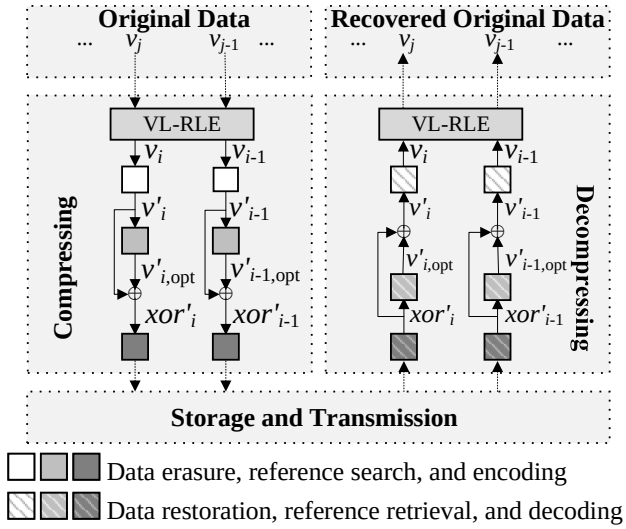



Fig. 4. Overall workflow of the Kangaroo.

zeros (*lead*), trailing zeros (*trail*), and differing center bits (*center*) between the current value $v_i$ and its reference, but also the reference's position information (*index*). Formally, the encoded result can be expressed as

$$result_{en} = index \,||\, lead \,||\, trail \,||\, center. \tag{2}$$

" || " denotes the bit-level concatenation of encoded bitstreams. Accordingly, we define a cost model for the encoded result as

$$cost_{\text{en}} = indexBits + leadBits + trailBits + centerBits. \tag{3}$$

*indexBits* denotes the number of bits required to encode the reference position, *leadBits* and *trailBits* represent the bits used to store the number of leading and trailing zeros, and *centerBits* corresponds to the bits representing the difference between $v_i$ and its reference value. This cost model quantifies the total number of bits needed to encode $v_i$ and is used to guide the selection of the optimal reference value.

Based on this cost model, the optimal reference value $v_{\text{opt}}$ is selected by minimizing the total encoding cost over the previous values set $\mathcal{P}_i$:

$$v_{\text{opt}} = \underset{v_j \in \mathcal{P}_i}{\arg\min}(cost_{\text{en}}). \tag{4}$$

This selects the reference value from the previous values that minimizes $cost_{\text{en}}$. Performing a global search over all previous values to find the optimal reference is computationally infeasible, especially in low-latency streaming scenarios. Leveraging the inherent temporal locality of floating-point values, we restrict the search to a fixed-size window $W$, under the assumption that more recent values are more likely to serve as effective references than older ones. In this way, the problem of finding the global optimal reference is transformed into finding the optimal reference within a given window, which we simply refer to as the "optimal reference" hereafter. However, beyond a certain window size, the benefits of increasing $W$ diminish: the marginal reduction in encoded

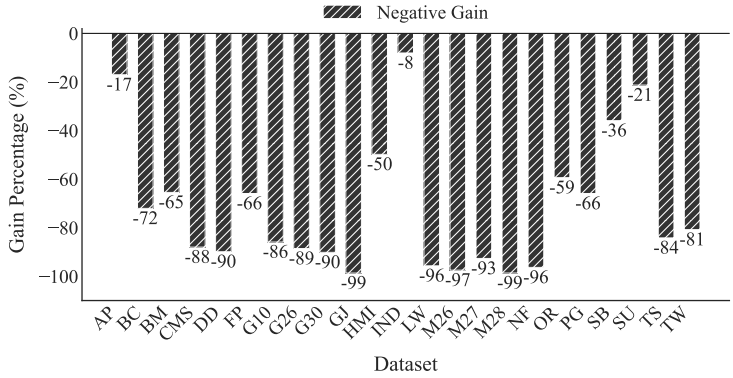



Fig. 5. Comparison of storage efficiency gains achieved by applying XOR-based trailing zeros versus the original trailing zeros. A value closer to -100% indicates a higher proportion of negative gains.

bits is often offset by the overhead of storing larger indices. Therefore, choosing an appropriate $W$ represents a practical trade-off between compression efficiency and computational cost.

### 3.3 Pruning-Based Encoding

*3.3.1 Trailing zeros benefit analysis.* Existing XOR-based methods store the number of trailing zeros in the XOR results directly without assessing their actual contribution to compression efficiency. However, we observe that XOR operations do not always yield positive trailing-zero gains. In some cases, they may even produce **negative benefits**. We define a function $f$ that characterizes the relationship between the number of trailing zeros in the XOR result and those in the original value, formally defined as:

$$f(v_i \oplus v_j) = \begin{cases} \min\big(f(v_i), f(v_j)\big), & \text{if } f(v_i) \neq f(v_j), \\ f(v_i) + \delta, & \text{if } f(v_i) = f(v_j), \end{cases} \tag{5}$$

where $\delta \in [1, n - f(v_i)]$ denotes the incremental value, and $n$ is the bit width (e.g., 64 bits).

We find that negative benefits occur only when the two XOR operands have different numbers of trailing zeros. To quantify the gain of storing trailing zeros from the XOR result compared to the original values, we define the *gain* $g_i$ as the difference between the number of trailing zeros in the XOR result and that in the original value:

$$g_i = trail_{xor} - trail_v, \tag{6}$$

where $trail_{xor}$ denotes the number of trailing zeros in the XOR result between $v_i$ and its reference, and $trail_v$ denotes the number of trailing zeros in $v_i$ itself.

We then define the positive and negative gains across all values as

$$\text{gain}_{\text{pos}} = \sum_{i:g_i>0} g_i, \quad \text{gain}_{\text{neg}} = \sum_{i:g_i<0} g_i, \tag{7}$$

and compute the overall gain metric $\mathcal{G}$ as

$$\mathcal{G} = \frac{\text{gain}_{\text{pos}} + \text{gain}_{\text{neg}}}{\text{gain}_{\text{pos}} + |\text{gain}_{\text{neg}}|}. \tag{8}$$

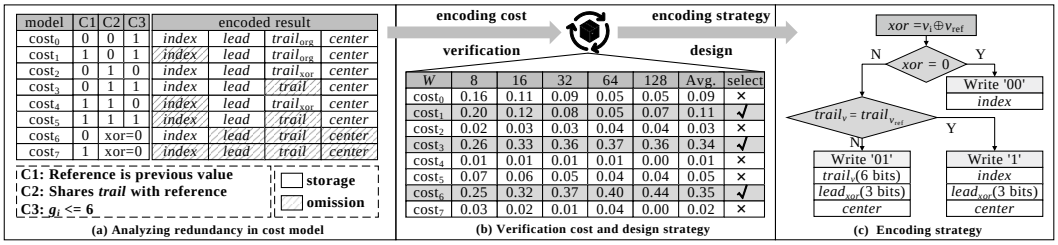



Fig. 6. Evolution of pruning-based encoding.

We analyzed the overall gain of immediate-predecessor-based compression algorithms, as shown in Figure 5. The results indicate that negative gain occurs across all datasets, with over 30% of the datasets exhibiting a negative gain proportion exceeding 90%. This compelling evidence demonstrates that retaining trailing zeros after XOR operations substantially degrades compression efficiency. Therefore, when the operands have unequal trailing zeros, we adopt the strategy of keeping only the trailing zeros of the original values, thereby completely avoiding the performance penalty induced by XOR operations.

*3.3.2 Analyzing redundancy in cost model.* The $cost_{en}$ introduced in Section 3.2 is employed to evaluate the storage overhead between $v_i$ and candidate reference values, thereby determining the selection of the optimal reference value. As shown in Figure 6(a), our analysis identifies optimization opportunities in specific scenarios: (C1) when the immediate predecessor is used as the reference, storing the index variable is unnecessary; (C2) when $v_i$ and the reference have different trailing zeros, according to the method in Section 3.3.1, we store the trailing zeros of $v_i$ directly; (C3) when $v_i$ and the reference share identical trailing zeros, we further examine the gain $g_i$: if $g_i > 6$ (corresponding to the 6-bit exact representation of $trail$), the trailing zeros of the XOR result are stored; otherwise, trailing zeros information is not stored, as it can be implicitly obtained from the reference value. When the XOR result equals zero, $lead$, $trail$, and $center$ can be entirely omitted. For $lead$, we adopt the 3-bit approximate representation proposed by *Chimp*. Based on the above optimization strategies, we further define eight sub-cost models to accommodate different optimization scenarios.

*3.3.3 Verification and Pruning of Cost Models.* As shown in Figure 6(b), we analyze the distribution of optimal reference selections under different cost models across varying window sizes $W$ on real-world datasets, revealing how frequently each model is applied in practice. Although a more fine-grained set of cost models can theoretically improve compression ratio, it also introduces non-negligible encoding and control-flow overhead. Specifically, an 8-branch design results in an average label length of 2.24 bits, whereas the optimized 3-branch scheme requires only 1.54 bits, reducing metadata overhead by 31.3%. Moreover, the complex branching structure significantly reduces branch-prediction effectiveness, leading to additional runtime penalties. To balance compression efficiency and computational cost, we retain the two most frequently used cost models, $cost_3$ and $cost_6$, and prune the rarely used models whose marginal compression benefits are outweighed by branch management overhead. Among the remaining candidates, we select a single fallback model to handle cases where neither $cost_3$ nor $cost_6$ applies. Since the usage frequency of $cost_0$ and $cost_1$ is comparable, we adopt the simpler $cost_1$, which relies solely on the immediate predecessor. In practice, when a reference value with matching trailing-zero patterns is found in the history window, $cost_3$ or $cost_6$ is applied; otherwise, $cost_1$ is used as a fallback. By partitioning reference candidates based on trailing-zero characteristics, this combined strategy accelerates the reference search

process and improves throughput while preserving most of the compression gains of fine-grained models.

*3.3.4 Encoding strategy based on pruned cost models.* Based on the three pruned cost models, we propose a pruning-based encoding strategy, as illustrated in Figure 6(c). Variable-length markers "`1`", "`00`", and "`01`" are assigned to these branches according to their usage frequency. Notably, both branches storing *center* employ precise trailing zero preservation for $v_i$. This implementation reveals a key optimization opportunity: when the center bits length exceeds zero, the last bit must be '1' due to the trailing zero representation. We can therefore safely omit this deterministic bit, achieving an additional 1-bit saving for all qualifying values without information loss. Results show this optimization applies to over 70% of datasets.

*3.3.5 VL-RLE optimization for repeated values.* Existing XOR-based compression algorithms process each repeated value independently during encoding, requiring a complete compression cycle for every occurrence, which leads to low computational efficiency. *SElf** further calculates and stores the number of significant decimal digits for each value while performing erasure operations, introducing additional computational and storage overhead [17]. For $k$ consecutive repeated values (starting from the first value), *SElf** consumes $64 + (1 + 2) \times k$ bits of storage, and such redundant operations degrade the overall compression performance. To overcome this limitation, *Kangaroo* introduces the VL-RLE optimization before the erasure stage. This method records only the run length of consecutive repeated values, avoiding redundant compression for each repeated value. When a new distinct value is detected, the algorithm performs compression only for that value and simultaneously writes its run length. By doing so, *Kangaroo* effectively reduces redundant computations and improves compression efficiency for high-repetition floating-point values while maintaining correctness.

## 4 Dynamic Reference Search

Although the partitioning based on trailing zeros reduces candidate redundancy, exhaustive traversal within each partition remains costly. To address this issue, this section presents a dynamic reference search strategy that efficiently identifies the optimal reference value by skipping over values that cannot serve as the optimal reference

Let $\mathcal{H}_i$ denote the set of previous values within the temporal window that share identical trailing zeros with the current value $v_i$. Each element in this set, denoted as $h_j \in \mathcal{H}_i$, is a candidate reference value. To efficiently select the optimal reference value from $\mathcal{H}_i$, *Kangaroo* employs a dynamic search strategy that chooses the candidate $h_j$ maximizing the number of leading zeros in the XOR with $v_i$, thereby improving compression efficiency:

$$v_{\text{opt}} = \arg\max_{h_j \in \mathcal{H}_i} \text{lead}(v_i \oplus h_j), \tag{9}$$

where $\text{lead}(\cdot)$ returns the number of leading zeros in the XOR result. By skipping candidate references that cannot yield the maximal leading zeros gain, this strategy reduces computational complexity while maintaining high compression performance.

To minimize redundant computations over the candidate set $\mathcal{H}_i$, we propose a filtering strategy that leverages historical computation results $\text{lead}(h_j \oplus h_{j+1})$. Based on the algebraic property:

$$\text{lead}(v_i \oplus h_j) = \text{lead}((v_i \oplus h_r) \oplus (h_r \oplus h_j)). \tag{10}$$

**Definition 5.** ***LZ-History Array.*** *For a value $v_i$ with candidate set $\mathcal{H}_i$, the LZ-history array $\mathcal{L}_i \in \mathbb{N}^k$ is defined by its components:*

$$\mathcal{L}_i[j] = \text{lead}(h_j \oplus h_{j+1}), \quad \textit{for } 0 \le j \le k-2, \tag{11}$$

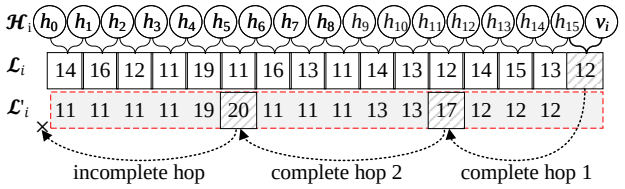



Fig. 7. An example of the optimal reference value selection.

*where* $h_j \in \mathcal{H}_i$.

The LZ-history array $\mathcal{L}_i$ records the number of leading zeros in the XOR results between consecutive values in $\mathcal{H}_i$. Upon the arrival of a new value $v_i$, only one entry is updated:

$$\ell_{k-1} \leftarrow \text{lead}(h_{k-1} \oplus v_i), \tag{12}$$

stored at index $k-1$ of $\mathcal{L}_i$. This incremental update mechanism preserves the computation history and allows for efficient identification of optimal reference values while maintaining $O(1)$ update complexity per new value.

**Example** 1. *As depicted in Fig. 7, given the current value $v_i$ and a history buffer $\mathcal{H}_i$ of size $|\mathcal{H}_i| = 16$ with its associated LZ-history array $\mathcal{L}_i$, the conventional update procedure would compute the exact number of leading zeros in the XOR result between $v_i$ and each element in $\mathcal{H}_i$, storing these counts in $\mathcal{L}'_i$. For instance, the XOR between the last element $v_{15} \in \mathcal{H}_i$ and $v_i$ yields 12 leading zeros, which would be recorded in $\mathcal{L}_i$. However, in our proposed dynamic search strategy, it is unnecessary to compute the exact leading zeros for every element in $\mathcal{H}_i$. Instead, the algorithm efficiently identifies the optimal candidate to maximize leading zeros gain, reducing computational overhead while still ensuring optimal reference selection.*

**Definition** 6. ***Incomplete Hop.*** *For $v_i$ with $\mathcal{H}_i$ and its associated LZ-history array $\mathcal{L}_i$, interval $[a, b)$ is an* incomplete hop *if:*

$$\text{T-LSIS}(\mathcal{L}_i[a:b) \cup \{\ell'_b\}) = \text{T-LNDS}(\mathcal{L}_i[a:b) \cup \{\ell'_b\}), \tag{13}$$

*where $\ell'_b = \text{lead}(h_b \oplus v_i)$.*

**Example** 2. *As illustrated in Fig. 7, the interval $[12, 15)$ constitutes an incomplete hop for the current value $v_i$, since:*

$$\text{T-LSIS}\big(\mathcal{L}_i[12:15) \cup \{12\}\big) = \text{T-LNDS}\big(\mathcal{L}_i[12:15) \cup \{12\}\big) = \langle 12 \rangle.$$

*The same property holds for intervals $[6, 11)$ and $[0, 5)$.*

**Definition** 7. ***Complete Hop.*** *For $v_i$ with $\mathcal{H}_i$ and its associated LZ-history array $\mathcal{L}_i$, the interval $[a, b)$ is a* complete hop *if:*

- ***(1)*** *$[a+1, b)$ is an incomplete hop (Definition 6)*
- ***(2)*** *$\ell_a = \min(\mathcal{L}_i[a+1:b) \cup \{\ell'_b\})$*

*where $\ell'_b = \text{lead}(h_b \oplus v_i)$.*

**Example** 3. *As shown in Fig. 7, interval $[11, 15)$ with reference $v_i$ satisfies both conditions of Definition 7:*

- ***(1)*** *$[12, 15)$ is an incomplete hop (Definition 6)*
- ***(2)*** *$\ell_{11} = \min\big(\mathcal{L}_i[12:15) \cup \{12\}\big) = 12$*

*Therefore, interval* $[11, 15)$ *is considered a complete hop. The symmetric interval* $[5, 11)$ *similarly qualifies as a complete hop.*

Based on Definitions 6 and 7, we partition the candidate references into subintervals. We then rigorously prove the optimization effectiveness of these two interval classes through the following theorems.

Theorem 1. *Given* $v_i$ *with* $\mathcal{H}_i$ *and its associated LZ-history array* $\mathcal{L}_i$*, if* $[a, b)$ *is an incomplete hop, there exists no optimal reference value in* $\mathcal{H}_i[a : b)$ *that is strictly better than* $h_b$*:*

$$\text{lead}(h_j \oplus v_i) \leq \text{lead}(h_b \oplus v_i), \text{for } \forall j \in [a, b). \tag{14}$$

Proof. Let $\mathcal{S} := \text{T-LSIS}(\mathcal{L}_i[a : b) \cup \{\ell'_b\})$ be the selected subsequence according to Definition 6, and denote its $i$-th element by $S_i$. The strict monotonicity $S_i < S_{i+1}$ for consecutive elements is guaranteed by Definition 3.

For any interval $(l, r)$ derived from $\ell_a \cup \mathcal{S}$, the number of leading zeros for the value satisfies:

$$\forall t \in (l, r), \quad \text{lead}(h_t \oplus v_i) = \ell'_r, \tag{15}$$

with the boundary condition $\ell'_r = \begin{cases} \ell'_b & \text{if } r = b \\ \ell_r & \text{otherwise} \end{cases}$.

The conjunction of these results establishes the uniform bound:

$$\forall j \in [a, b), \quad \text{lead}(h_j \oplus v_i) \leq \text{lead}(h_b \oplus v_i). \tag{16}$$

□

**Example** 4. *As illustrated in Fig. 7, the interval* $[6, 11)$ *constitutes an incomplete hop, from which we derive the subsequence* $\mathcal{S} := \text{T-LSIS}(\mathcal{L}_i[6 : 11) \cup \{\ell'_{11}\}) = \langle 11, 13, 17\rangle$*. The corresponding indices* $\langle 8, 10, 11\rangle$ *together with the left boundary 6 partition the interval into three segments:*$[6, 8]$, $(8, 10]$, $(10, 11]$*. Within each segment, all points yield identical leading zeros when XORed with* $v_i$*. Since* $\mathcal{S}$ *is strictly monotonically increasing, there is no value on* $[6, 11)$ *that is better than* $v_{11}$*.*

Theorem 2. *Given the current value* $v_i$*, its candidate set* $\mathcal{H}_i$*, and the corresponding leading-zero history array* $\mathcal{L}_i$*, if the interval* $[a, b)$ *constitutes a complete hop, then within this interval, only* $h_a$ *may serve as a reference value superior to* $h_b$*.*

Proof. By theorem 1, $\text{lead}(h_j \oplus v_i) \leq \text{lead}(h_b \oplus v_i)$ holds for all $j \in (a, b)$. Definition 7 gives $\ell_a = \min(\mathcal{L}_i[a + 1 : b) \cup \{\ell'_b\})$.

The key inequality from (5) and (10) is:

$$\text{lead}(h_a \oplus v_i) > \ell_a, \tag{17}$$

which has the potential to exceed $\text{lead}(h_b \oplus v_i)$ and become a better reference value. When $\ell_a = \text{lead}(h_b \oplus v_i)$, this necessarily implies the strict inequality:

$$\text{lead}(h_a \oplus v_i) > \text{lead}(h_b \oplus v_i). \tag{18}$$

□

**Example** 5. *As shown in Fig. 7, the interval* $[5, 11)$ *constitutes a complete hop while* $[6, 11)$ *is an incomplete hop, which implies no point in* $[6, 11)$ *is better than* $v_{12}$*. Given* $\ell_5 = \ell_8$*, we derive:*

$$\text{lead}(h_5 \oplus h_8) = \text{lead}(h_8 \oplus v_i) = 11.$$

*Consequently, the computation* $\text{lead}(h_5 \oplus v_i) = 20$ *demonstrates that* $v_5$ *generates more leading zeros than* $v_8$ *when XORed with* $v_i$*, and outperforms* $v_{11}$ *in this example.*

**Algorithm 1** The Optimal Reference Value Selection

**Require:** $v_i$, candidate set $\mathcal{H}_i = \{h_0, \ldots, h_{k-1}\}$, precomputed $\mathcal{L}_i = \{\ell_0, \ldots, \ell_{k-1}\}$
**Ensure:** The optimal reference $v^*_{\text{opt}}$

```
1:  v*_opt ← NULL                                   ▷ Default to no valid candidate
2:  max_lead ← lead(v_i ⊕ h_{k−1})                  ▷ Initialize with last candidate
3:  min_lead ← max_lead                             ▷ Initialize lead threshold for decision-making
4:  for j ← k − 2 downto 0 do                       ▷ Reverse traversal
5:      if ℓ_j < min_lead then
6:          min_lead ← ℓ_j                          ▷ Update minimum threshold
7:      else if ℓ_j = min_lead then
8:          min_lead ← lead(h_j ⊕ v_i)              ▷ Exact recomputation
9:          if min_lead > max_lead then
10:             max_lead ← min_lead                 ▷ New optimal found
11:             v*_opt ← h_j                        ▷ Update optimal reference
12:         end if
13:     end if
14: end for
15: return v*_opt                                   ▷ Return the optimal reference in H_i
```

Theorem 3. *Given $v_i$ with $\mathcal{H}_i$ and its associated LZ-history array $\mathcal{L}_i$, the search space consists of $p$ consecutive complete hops with boundary set $\mathcal{B}_i = \{\mathcal{H}_i[s_p], \ldots, \mathcal{H}_i[s_1], \mathcal{H}_i[s_0 = k-1]\} \subseteq \mathcal{H}_i$, where an additional incomplete hop $[0, s_p)$ exists if and only if $s_p \neq 0$. The optimal reference value for $v_i$ must lie in $\mathcal{B}_i$.*

Proof. We systematically construct the boundary set $\mathcal{B}_i$ through backward traversal of $\mathcal{H}_i$:

(1) Complete Hop Identification: Starting from $h_{k-1}$, we detect complete hops $[s_{j+1}, s_j)$ for $j = 0, ..., p-1$ where each $s_{j+1} < s_j$, using criterion in Theorem 2. This yields ordered boundaries $\mathcal{B}_i = \{\mathcal{H}_i[s_p], \ldots, \mathcal{H}_i[s_1], \mathcal{H}_i[s_0 = k-1]\} \subseteq \mathcal{H}_i$.
(2) Incomplete Hop Handling: When $s_p \neq 0$, the residual segment $[0, s_p)$ forms an incomplete hop. Theorem 1 shows no values on $[0, s_p)$ can surpass $\mathcal{H}_i[s_p]$ in XOR optimization.

The boundary set $\mathcal{B}_i$ contains all candidate optimal reference values, as all other values have been rigorously proven non-optimal. □

Theorem 3 rigorously proves that the optimal reference value for $v_i$ must be contained within the boundary set $\mathcal{B}_i$. By processing the candidate values within $\mathcal{B}_i$, we enhance the overall compression efficiency while preserving optimality guarantees. Building upon Theorem 3, we derive a concrete algorithmic realization of optimal reference value selection as shown in Algorithm 1.

As shown in Algorithm 1, the process first initializes the maximum number of leading zeros *max_lead* and the minimum threshold *min_lead* (Lines 2-3), followed by a reverse traversal of the candidate set $\mathcal{H}_i$ (Line 4). For each candidate value $h_j$, the algorithm initially compares the precomputed $\ell_j$ with the current threshold (Line 5): if $\ell_j$ is less than the threshold, *min_lead* is updated (Line 6); if equal, we recompute the number of leading zeros between $v_i$ and $h_j$, marking the boundary of a complete hop (Line 8). When a superior boundary is identified (Line 9), the algorithm immediately updates the optimal reference value $v^*_{\text{opt}}$ (Lines 10-11). The returned $v^*_{\text{opt}}$ is guaranteed to maximize the leading zeros gain $lead(v_i \oplus h_j)$ among candidates $h_j \in \mathcal{H}_i$ (Line 15). Although the algorithm has a time complexity of $O(k)$, its practical efficiency outperforms linear scanning due to the pruning effect of the min_lead threshold.

To further enhance efficiency, we constrain the optimal reference value selection to cases satisfying $\ell_a = \text{lead}(h_b \oplus v_i)$ in Theorem 2, which guarantees strictly positive gain over $h_{k-1}$ and maintaining near-identical compression performance. We refer to the former as **Compact-Kangaroo** for its superior compression ratio, and the latter as **Fast-Kangaroo** for its accelerated processing speed.

Fast-Kangaroo employs an efficient search strategy that eliminates the overhead of scanning the candidate references. Its core operation consists of two steps: first, it computes the XOR between the current value $v_i$ and the most recent reference $h_{k-1}$, and records the number of leading zeros $\ell_{k-1}$ (consistent with Compact-Kangaroo); second, the algorithm uses $\ell_{k-1}$ as a direct index to retrieve the optimal reference value from a precomputed lookup table. This approach reduces the time complexity of the search from $O(W)$ to $O(1)$, making its performance independent of the window size $W$. In terms of memory footprint, the algorithm only requires maintaining a lookup table whose size is determined by the maximum possible leading zeros (e.g., 64 in 64-bit systems), thereby bounding the space overhead to a constant.

**Example** 6. *As shown in Figure 7, through reverse traversal of the $\mathcal{L}_i$ array and by applying Definitions 7 and 6, we identify two complete hops and one incomplete hop. Leveraging Theorems 2, 1, and 3, we obtain the candidate set $\mathcal{B}_i = \{v_5, v_{11}, v_{15}\}$. The optimization results demonstrate that Compact-Kangaroo identifies $v^*_{opt} = v_5$ with $\text{lead}(v_i \oplus v^*_{opt}) = 20$, while Fast-Kangaroo selects $v^*_{opt} = v_{11}$ achieving $\text{lead}(v_i \oplus v^*_{opt}) = 17$.*

If no candidate references is found in the history buffer, the algorithm defaults to using the preceding value as the reference. Theoretical analysis demonstrates that this approach achieves $O(1)$ search complexity. Experimental results show only a 0.005 compression ratio degradation at $W = 32$ (0.245 vs. 0.240). Consequently, we adopt the **Fast-Kangaroo** scheme with the best cost-effectiveness as our primary solution, hereafter referred to as **Kangaroo**.

## 5 Bit-Flip-Based Erasure

In this section, we present a bit-flip-based erasure strategy aimed at increasing trailing zeros in floating-point values by discarding less significant mantissa bits. While existing algorithms such as *BUFF* [25], *SElf** [19], and *Camel* [26] also exploit trailing zeros, they require additional computation and storage for auxiliary information. Our approach guarantees lossless reconstruction without extra space or computational overhead, and it facilitates the identification of reference values in the previous values that share identical trailing zero patterns with $v_i$.

By erasing the last few bits of the mantissa within the IEEE 754 standard, we increase the number of trailing zeros in the value itself, thereby effectively enhancing the compression efficiency in the subsequent encoding phase. The length of bits to be erased is defined by the following formula:

$$L_{\text{era},\alpha} = \max\left(0,\ 52 - E - \left\lceil \alpha \log_2 10 \right\rceil\right). \tag{19}$$

Here, $\alpha$ denotes the decimal places, as defined in Section 2.2.

If $L_{\text{era},\alpha} = 0$, no erasure operation is performed. Therefore, the subsequent analysis in this paper addresses only cases with a non-zero length of bits to be erased.

To achieve lossless recovery of value after erasure, $\alpha$ of $v_i$ must be known. From Equation 19, we derive the relationship between the decimal places $\alpha$, the exponent $E$, and the erased bits length $L_{\text{era},\alpha}$ of $v_i$:

$$\alpha = \lfloor (52 - E - L_{\text{era},\alpha}) \lg 2 \rfloor. \tag{20}$$

Theorem 4. *Given a value $v_i$ and its erased value $v'$ with corresponding exponents $E$ and $E'$ respectively, then $E = E'$.*

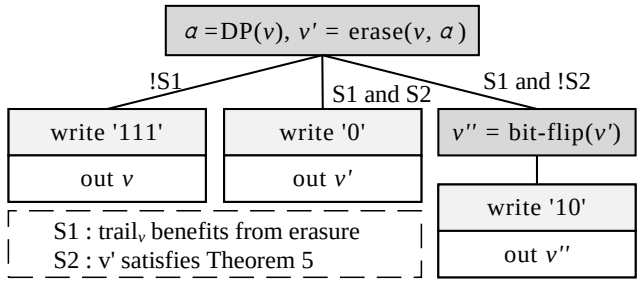



Fig. 8. Workflow of the bit-flip-based erasure.

Proof. According to IEEE 754 standard, the exponent value is entirely determined by bits 62-52. From Equation (19), the erasure operation only modifies bits 51-0, thus $v_i$ and $v'$ have identical exponent bit fields. Therefore their exponent values must satisfy $E = E'$. □

Theorem 5. *Let $v_i$ be a floating-point value and its erased value $v'$. If the following trailing zeros constraint holds:*

$$L_{era,\alpha} \leq trail_{v'} < L_{era,\alpha-1}, \tag{21}$$

*then the $\alpha$ can be uniquely reconstructed as:*

$$\alpha = \lfloor (52 - E' - trail_{v'}) \lg 2 \rfloor . \tag{22}$$

Proof. Starting from Equation 20 under the trailing zeros constraint specified in Theorem 5, we derive the fundamental bounding inequality:

$$\alpha \geq \lfloor (52 - E - trail_{v'}) \lg 2 \rfloor > \alpha - 1. \tag{23}$$

By Theorem 4, we have established $E = E'$ through exponent bit preservation. Combining this with the integer constraint on $\alpha$, the solution is uniquely given by:

$$\alpha = \lfloor (52 - E' - trail_{v'}) \lg 2 \rfloor. \tag{24}$$

□

Theorem 5 states that the decimal places $\alpha$ can be directly recovered from the erased value when the trailing zeros constraint is satisfied. For cases that do not satisfy this constraint, We propose flipping the 0-bit at the $(L_{\text{era},\alpha-1} - 1)$-th position to 1, thereby enforcing compliance with the trailing zeros constraint. This correction operation requires an additional flag bit for identification.

Based on the above analysis, we propose a bit-flip-based erasure strategy (Fig. 8). The process begins by performing an erasure operation on the current value: $v' = erase(v, \alpha)$. We define $S1$ as the condition that the erasure operation yields a positive trailing-zero gain, i.e., $trail(v') > trail(v)$. If $S1$ is not satisfied, the original value $v_i$ is output directly. Otherwise, we further verify whether the erased value satisfies the condition specified in Theorem 5.2. We denote this condition as $S2$, which indicates that the decimal precision $\alpha$ can be derived from $v'$. If $S2$ is satisfied, we store $v'$. If not, we apply a bit-flip operation to generate and store $v'' = bit\text{-}flip(v')$.

According to the observed frequencies of these three cases, we assign variable-length encoding tag bits as follows. Considering that the less frequent VL-RLE is reserved with the tag “110”, the three cases above are assigned the tags “111”, “0”, and “10”, respectively.

**Example** 7. *The three cases illustrated in Fig. 9 demonstrate different data processing mechanisms. For value 1.25, the erasure operation provides no positive gain, so the original value 1.25 is stored directly. When processing value 2.87, the erasure operation produces 2.8671875, yielding positive trailing*

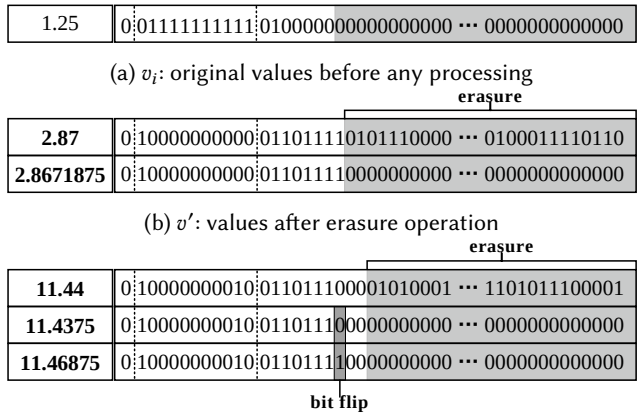



Fig. 9. Illustration of three cases in the bit-flip-based erasure.

*zeros gain while satisfying the recovery conditions in Theorem 5; therefore, the erased result 2.8671875 is stored. For value 11.44, the erasure produces 11.4375, which fails to meet the recovery conditions, triggering a bit-flip; consequently, the re-encoded value 11.46875 is stored.*

## 6 Experiments

### 6.1 Datasets and Experimental Settings

*6.1.1 Datasets.* In this study, we evaluate 26 datasets, including 13 time-series (TS) datasets—three from the Lindorm database—and 13 non-time-series (Non-TS) datasets without temporal correlations. Building on the benchmarks from *Elf* [17] and *ALP* [20], we extend the evaluation to additional domains such as vehicular networks, database systems, and network monitoring. Table2 reports the decimal places (DP) , Coefficient of variation(CV) and the average consecutive length (ACL) for use in subsequent performance analysis.

*6.1.2 Baselines.* We evaluate a comprehensive set of 12 lossless floating-point compression algorithms: 6 state-of-the-art streaming algorithms including *Gorilla* [15], *Chimp* [16], *Chimp*$_{128}$ [16], *Elf* [17], *Elf+* [18], and *SElf** [19];along with 6 widely-used batch algorithms: *Elf** [19], *Buff* [25], *XZ* [8], *Zstd* [7], *Snappy* [43] and *ALP* [20]. All streaming compressors except *Chimp*$_{128}$ use the immediate predecessor value, while *Chimp*$_{128}$ leverages 128 previous values. Batch-based algorithms operate on data chunks. Prior to compressing the current batch, they typically perform statistical analysis to extract global characteristics, thereby optimizing compression performance through holistic information utilization. In our experiments, we use *Fast-Kangaroo* (hereafter referred to as *Kangaroo*) as our proposed method for comparison.

*6.1.3 Metrics.* We evaluate lossless compression algorithms using three core metrics: the compression ratio $\rho = S_{\text{comp}}/S_{\text{orig}}$, the compression speed $v_{\text{comp}} = S_{\text{orig}}/T_{\text{comp}}$ (MB/s), and the decompression speed $v_{\text{dec}} = S_{\text{orig}}/T_{\text{dec}}$ (MB/s), where $S_{\text{orig}}$ and $S_{\text{comp}}$ are the sizes of the original and compressed data, and $T_{\text{comp}}$ and $T_{\text{dec}}$ are the compression and decompression times.

*6.1.4 Settings.* We implemented the *Kangaroo* algorithm in Java. All experiments were conducted on an Alibaba Cloud server with the following specifications: Intel(R) Xeon(R) Platinum 8369B CPU @ 2.70GHz (8 cores, 16 threads). Following *Chimp's* methodology, for any given dataset,

Table 2. Details of Datasets.

| | | Dataset | Records | DP | CV | ACL | Domain |
|---|---|---|---|---|---|---|---|
| Time Series | Public | Air-pressure (AP) [27] | 525600 | 0~5 | 0.01 | 3 | Meteorology |
| | | Bird-migration (BM) [28] | 17964 | 1~5 | 0.66 | 1 | Ecology |
| | | DEPADD-demand (DD) [29] | 2106 | 1~11 | 0.11 | 1 | Electricity |
| | | HMI-xAcc (HMI) [30] | 143160 | 1~2 | 16.92 | 1 | IoV |
| | | InD-xVelocity (IND) [31] | 1059267 | 0~5 | 14.01 | 3 | IoV |
| | | LW-precipitation (LW) [32] | 15341 | 0~1 | 2.24 | 1 | Hydrology |
| | | Network-flow (NF) [33] | 35250 | 1~9 | 5.11 | 1 | Networking |
| | | Online-Retail/5 (OR) [34] | 541909 | 1~3 | 20.98 | 1 | E-commerce |
| | | Phone-gyro (PG) [35] | 41797896 | 1~16 | 93.83 | 1 | IoT |
| | | Stocks-USA (SU) [36] | 100000 | 1~2 | 0.54 | 1 | Finance |
| | Private | Decel (DE) | 5720095 | 0~7 | 0.15 | 150529 | IoV |
| | | Latpos (LAT) | 124996 | 0~8 | 121.55 | 18 | IoV |
| | | Rln-poly (RP) | 124996 | 0~8 | 34.97 | 1 | IoV |
| Non-Time Series | Public | Blockchair (BC) [37] | 389717 | 1~4 | 47.99 | 1 | Blockchain |
| | | CMSprovider1 (CMS) [38] | 18575752 | 1~10 | 2.20 | 2 | Healthcare |
| | | Food-prices (FP) [39] | 2050638 | 1~4 | 16.68 | 3 | Economics |
| | | Gov1/10 (G10) [38] | 11387079 | 1~2 | 4371.42 | 10 | Monetary |
| | | Gov1/26 (G26) [38] | 11387079 | 1~2 | 134.15 | 215 | Monetary |
| | | Gov1/30 (G30) [38] | 11387079 | 1~2 | 25.70 | 8 | Monetary |
| | | GVGS-JPSales (GJ) [40] | 16719 | 1~2 | 3.98 | 1 | Gaming |
| | | Medicare2/26 (M26) [38] | 9153273 | 0~11 | 7.39 | 2 | Healthcare |
| | | Medicare2/27 (M27) [38] | 9153273 | 0~11 | 3.35 | 1 | Healthcare |
| | | Medicare2/28 (M28) [38] | 9153273 | 0~10 | 7.23 | 2 | Healthcare |
| | | SD-bench (SB) [41] | 8927 | 1~4 | 1.64 | 1 | Technology |
| | | Tpcxbb-store (TS) [42] | 98740116 | 1~2 | 2.03 | 1 | Retail |
| | | Tpcxbb-web (TW) [42] | 123347835 | 1~2 | 1.80 | 1 | E-commerce |

we partition every 1,000 consecutive values into a block. The compression algorithm evaluates performance metrics on each individual block, with the final results obtained by averaging all block-level measurements. In particular, for the highly competitive *ALP* algorithm, to ensure a fair comparison, we also implemented *Kangaroo* in C++ and compared it with the open-source C++ implementation of *ALP* under different batch sizes.

### 6.2 Compression Ratio

As shown in Table 3, we report the compression ratio to systematically compare streaming and batch algorithms for floating-point compression on both time-series (TS) and non-time-series (Non-TS) datasets. The window size parameter $W$ in *Kangaroo* is set to 32, which achieves the best performance across our datasets, as demonstrated in Subsection 6.4.

Overall, *Kangaroo* demonstrates strong advantages in both streaming and batch compression scenarios. For streaming algorithms, *Kangaroo* improves the average compression ratio by 23.2%–60.4% compared to classical methods such as *Gorilla*, *Chimp*, and their variants (e.g., $(0.619 - 0.245)/0.619 \approx 60.4\%$). These gains stem primarily from its dynamic reference encoding mechanism and low erasure overhead. For batch-based algorithms, although can exploit statistical information to achieve better compression ratios, both general-purpose compressors and existing floating-point-specific compressors still fail to fully leverage the structural characteristics of floating-point values, resulting in suboptimal performance on datasets with certain data characteristics. In contrast, *Kangaroo* outperforms the most competitive batch compressors (e.g., *Xz* and

Table 3. Overall comparison with baselines, Batch Size=1000(the best results are in bold and shaded background).

| Dataset | Streaming | | | | | | | Batch | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Gorilla | Chimp | Ch.128 | Elf | Elf+ | SElf* | Kangaroo | ALP | Elf* | Buff | Xz | Zstd | Snappy |
| AP | 0.247 | 0.246 | 0.234 | 0.185 | 0.126 | 0.117 | **0.091** | 0.262 | 0.115 | 0.378 | 0.151 | 0.176 | 0.224 |
| BM | 0.742 | 0.655 | 0.410 | 0.356 | 0.312 | 0.290 | **0.223** | 0.316 | 0.254 | 0.407 | 0.360 | 0.428 | 0.514 |
| DD | 0.922 | 0.878 | 0.707 | 0.913 | 0.912 | 0.829 | 0.733 | 0.792 | 0.793 | 0.878 | **0.534** | 0.664 | 0.850 |
| HMI | 0.689 | 0.562 | 0.222 | 0.243 | 0.234 | 0.193 | 0.111 | 0.183 | 0.191 | 0.253 | **0.082** | 0.117 | 0.201 |
| InD | 0.299 | 0.270 | 0.340 | 0.155 | 0.147 | 0.133 | 0.104 | **0.103** | 0.124 | 0.131 | 0.228 | 0.267 | 0.341 |
| LW | 0.661 | 0.588 | 0.279 | 0.249 | 0.252 | 0.218 | 0.141 | 0.147 | 0.214 | 0.253 | **0.128** | 0.164 | 0.229 |
| NF | 0.900 | 0.861 | 0.696 | 0.677 | 0.652 | 0.595 | **0.437** | 0.720 | 0.592 | 0.753 | 0.635 | 0.669 | 0.744 |
| OR | 0.743 | 0.674 | 0.283 | 0.320 | 0.286 | 0.244 | 0.160 | 0.246 | 0.242 | 0.320 | **0.152** | 0.201 | 0.269 |
| PG | 1.030 | 0.989 | 0.727 | 0.797 | 0.762 | 0.655 | **0.528** | 0.661 | 0.653 | 1.002 | 0.568 | 0.597 | 0.740 |
| SU | 0.671 | 0.633 | 0.214 | 0.217 | 0.165 | 0.139 | 0.149 | 0.147 | **0.121** | 0.253 | 0.157 | 0.220 | 0.303 |
| DE | 0.017 | 0.033 | 0.142 | 0.049 | 0.063 | 0.050 | **0.001** | 0.003 | 0.051 | 0.020 | 0.008 | 0.004 | 0.049 |
| LAT | 0.073 | 0.082 | 0.155 | 0.076 | 0.090 | 0.072 | **0.015** | 0.070 | 0.071 | 0.124 | 0.023 | 0.024 | 0.066 |
| RP | 0.582 | 0.533 | 0.270 | 0.342 | 0.288 | 0.273 | **0.152** | 0.418 | 0.264 | 0.478 | 0.169 | 0.197 | 0.250 |
| **TS AVG.** | 0.583 | 0.539 | 0.360 | 0.352 | 0.330 | 0.293 | **0.219** | 0.313 | 0.283 | 0.404 | 0.246 | 0.287 | 0.368 |
| BC | 0.900 | 0.819 | 0.688 | 0.562 | 0.521 | 0.482 | **0.426** | 0.554 | 0.462 | 0.560 | 0.571 | 0.622 | 0.759 |
| CMS | 0.575 | 0.543 | 0.439 | 0.434 | 0.398 | 0.373 | **0.285** | 0.570 | 0.354 | 0.735 | 0.392 | 0.433 | 0.501 |
| FP | 0.593 | 0.436 | 0.384 | 0.271 | 0.263 | 0.239 | 0.237 | 0.377 | **0.230** | 0.444 | 0.255 | 0.312 | 0.411 |
| G10 | 0.899 | 0.715 | 0.534 | 0.478 | 0.473 | 0.426 | **0.375** | 0.498 | 0.416 | 0.550 | 0.382 | 0.457 | 0.599 |
| G26 | 0.021 | 0.036 | 0.144 | 0.050 | 0.065 | 0.053 | **0.003** | 0.006 | 0.052 | 0.027 | 0.010 | 0.007 | 0.052 |
| G30 | 0.145 | 0.137 | 0.201 | 0.112 | 0.125 | 0.110 | **0.055** | 0.115 | 0.106 | 0.174 | 0.068 | 0.075 | 0.130 |
| GJ | 0.565 | 0.563 | 0.195 | 0.206 | 0.199 | 0.183 | 0.108 | **0.081** | 0.175 | 0.240 | 0.090 | 0.115 | 0.185 |
| M26 | 0.526 | 0.517 | 0.448 | 0.443 | 0.436 | 0.404 | **0.284** | 0.465 | 0.401 | 0.755 | 0.388 | 0.394 | 0.454 |
| M27 | 0.920 | 0.857 | 0.761 | 0.740 | 0.705 | 0.651 | **0.554** | 0.677 | 0.648 | 0.754 | 0.768 | 0.771 | 0.839 |
| M28 | 0.560 | 0.558 | 0.447 | 0.482 | 0.472 | 0.439 | **0.292** | 0.474 | 0.437 | 0.766 | 0.378 | 0.387 | 0.455 |
| SB | 0.787 | 0.706 | 0.291 | 0.319 | 0.289 | 0.265 | 0.172 | 0.233 | 0.253 | 0.378 | **0.125** | 0.166 | 0.264 |
| TS | 1.020 | 0.907 | 0.492 | 0.508 | 0.491 | 0.434 | 0.363 | **0.347** | 0.434 | 0.378 | 0.381 | 0.488 | 0.631 |
| TW | 1.018 | 0.898 | 0.503 | 0.507 | 0.485 | 0.429 | 0.373 | **0.353** | 0.428 | 0.395 | 0.395 | 0.499 | 0.645 |
| **NON-TS** | 0.656 | 0.592 | 0.425 | 0.393 | 0.379 | 0.345 | **0.271** | 0.365 | 0.338 | 0.473 | 0.323 | 0.364 | 0.456 |
| **ALL AVG.** | 0.619 | 0.565 | 0.393 | 0.373 | 0.354 | 0.319 | **0.245** | 0.339 | 0.311 | 0.439 | 0.285 | 0.325 | 0.412 |

*ALP*), achieving compression ratio improvements of 14.0% and 27.7%, respectively, highlighting its strong efficiency and broad applicability.

**When Kangaroo Performs Best.** *Kangaroo* performs best on datasets with large fluctuations in decimal precision and strong local correlations. For example, on the CMS dataset (widely used in prior evaluations), *Kangaroo* achieves a 23.6% compression ratio improvement over the best streaming competitor *SElf**. This improvement is mainly due to its dynamic reference selection, which identifies more effective reference values within the history window and captures local similarity more accurately. Compared with general-purpose batch compressors (e.g., *Xz*), *Kangaroo* better exploits floating-point structural characteristics. Compared with the floating-point–specific batch compressor *ALP*, *Kangaroo* improves compression by approximately 50% on CMS, because *ALP* treats many values as outliers under high decimal precision variability, introducing substantial exception-handling overhead. Moreover, the proposed VL-RLE mechanism further enhances *Kangaroo*'s ability to compress repeated values. For instance, on the DE dataset, *Kangaroo* compresses the data to 0.001 of the original size, achieving a 66.7% improvement over *ALP* (which incurs fixed metadata overhead). This result surpasses the traditional lower bound of streaming compressors; for example, *Gorilla* can only compress the same dataset to 0.017 due to its per-value marker bit requirement.

Table 4. Average compression and decompression speed (MB/s) across all datasets. Speedup is computed as Kangaroo speed divided by the algorithm speed.

| Algorithm | Speed (MB/s) | | | |
|---|---|---|---|---|
| | Comp | Speedup | Decomp | Speedup |
| **Kangaroo** | 271 | - | 444 | - |
| Gorilla | 420 | 0.65x | 465 | 0.96x |
| Chimp | 349 | 0.78x | 361 | 1.23x↑ |
| Ch.128 | 258 | 1.05x↑ | 335 | 1.33x↑ |
| Elf | 165 | 1.64x↑ | 198 | 2.24x↑ |
| Elf+ | 191 | 1.42x↑ | 218 | 2.04x↑ |
| SElf* | 127 | 2.13x↑ | 201 | 2.21x↑ |
| Elf* | 102 | 2.66x↑ | 218 | 2.04x↑ |
| Buff | 102 | 2.66x↑ | 93 | 4.77x↑ |
| Xz | 7 | 38.71x↑ | 52 | 8.54x↑ |
| Zstd | 42 | 6.45x↑ | 175 | 2.54x↑ |
| Snappy | 67 | 4.04x↑ | 233 | 1.91x↑ |

**When Kangaroo Performs Worse.** When the optimal reference value within the history window is most often the immediate predecessor, *Kangaroo* may perform slightly worse than predecessor-based XOR compressors such as *Elf**. However, since *Kangaroo* omits unnecessary trailing-zero markers, the degradation is limited and can be mitigated by reducing the window size $W$. *Kangaroo* also underperforms *ALP* on four datasets where floating-point values can be efficiently mapped to integers, making them particularly suitable for integer-based compression schemes. Additionally, *Kangaroo* is slightly worse than *Xz* on five datasets, with the largest gap observed on DD. This dataset contains many high-precision floating-point values, which are challenging for most floating-point compressors. On the remaining datasets, the performance gap is small and mainly due to weak local value correlation and stronger bit-level similarity than value-level similarity.

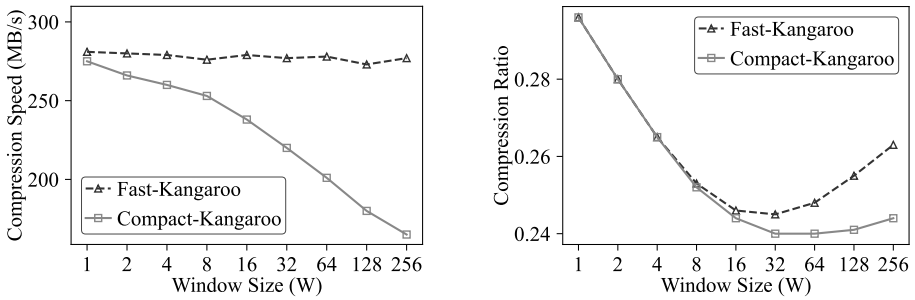
### 6.3 [De]compression Speed

Since *ALP* is highly sensitive to batch size and is designed as a batch-based compressor, it is not fully aligned with the streaming compression scenario considered in this paper. Therefore, we exclude it from the overall performance comparison in this subsection and provide a detailed analysis in Subsection 6.6.

**Compression Speed.** *Kangaroo* achieves higher compression ratios than existing streaming compressors while maintaining competitive compression speed. Its compression speed is comparable to $Chimp_{128}$, which also uses extended historical references, while the overhead introduced by its preprocessing erasure mechanism remains minimal. Compared with erasure-based compressors such as *SElf**, *Kangaroo* avoids costly statistical modeling and adaptive parameter tuning, achieving a 2.13× speedup. The integrated VL-RLE module further enables an 11.75× speedup on the DE dataset.

**Decompression Speed.** *Kangaroo*'s decompression speed is comparable to that of *Gorilla*, as it also supports $O(1)$ access to XOR references. It is 2.21× faster than SElf*, 8.54× faster than the best batch compressor Xz, and 2.04× faster than Elf*. The simplified design and the integration of VL-RLE allow repeated floating-point values to be efficiently decompressed.

We investigate the impact of observation window size $W$ on the compression performance of *Fast-Kangaroo* and *Compact-Kangaroo* algorithms (note that decompression time remains unaffected by




(a) Compression Speed (MB/s) (b) Compression Ratio

Fig. 10. Performance comparison between Fast-Kangaroo and Compact-Kangaroo under different window sizes $W$.

$W$). As shown in Figure 10a, the compression speed of *Compact-Kangaroo* decreases monotonically as $W$ increases, due to the growing number of candidate values that need to be checked. When $W$ reaches 256, the speed drops to 165 MB/s, matching the compression speed of the *Elf* algorithm. This demonstrates that the run-length encoding and proposed bit-flip-based erasure strategy in the preprocessing stage improve efficiency.

Compared to *Compact-Kangaroo*, *Fast-Kangaroo* sacrifices a small amount of performance to achieve $O(1)$ time complexity in locating the optimal XOR reference. Consequently, its compression speed remains nearly constant (with minor fluctuations due to experimental environment) as $W$ increases.

Both algorithms exhibit a non-monotonic trend in compression ratio (Figure 10b), reaching their minimum values at $W = 32$. This occurs because while larger $W$ provides more XOR references, it also increases the index size accordingly. When $W < 8$, the compression ratios of both algorithms are nearly identical, since the boundary set $\mathcal{B}$ within small windows is sufficiently small that *Fast-Kangaroo* can find the identical optimal XOR reference. For $W \geq 8$, as $W$ increases, the boundary set $\mathcal{B}$ expands, allowing *Compact-Kangaroo* to find better XOR targets than *Fast-Kangaroo* at the cost of more computation time. As the window size $W$ increases, this performance difference becomes more pronounced, since a larger window provides more candidate reference value selections.

Although *Compact-Kangaroo* utilizes the expanded search space more effectively than *Fast-Kangaroo*, the performance gain diminishes relative to the computational cost. At $W = 32$, for instance, *Compact-Kangaroo* achieves only a 2% better compression ratio than *Fast-Kangaroo*, but suffers a 21% reduction in speed. We therefore conclude that *Fast-Kangaroo* offers better cost-effectiveness for most applications. In our experiments, we compare *Fast-Kangaroo* with other baseline algorithms. However, *Compact-Kangaroo* may still be preferable in scenarios where space efficiency outweighs time efficiency.

### 6.4 Ablation Study of Core Modules

To understand the contribution of each component, we conduct an ablation study on the three core modules of *Kangaroo*: Dynamic Reference (DR), Bit-Flip Erasure (BF), and VL-RLE (VL). We evaluate each module individually and in combination, using Gorilla as the baseline. When VL-RLE is enabled, the original repeated-value handling branch is removed, as its functionality is subsumed by VL-RLE. We also analyze dataset characteristics, including coefficient of variation (CV) and average run length (ARL), to study how data properties affect module effectiveness. For

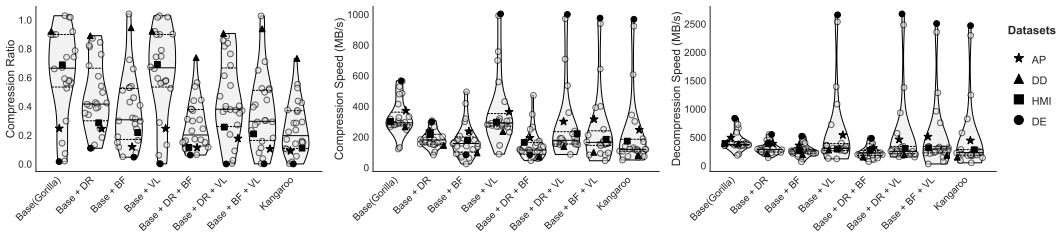



Fig. 11. Ablation study on Kangaroo components.

each configuration, we measure compression ratio, compression speed, and decompression speed. Figure 11 presents violin plots summarizing module impacts across datasets.

DR improves compression on datasets with high variability (e.g., HMI) by increasing the likelihood of selecting suitable reference values and consistently provides gains on low-CV datasets (e.g., AP). This is because DR evaluates references based on both numerical similarity and structural properties, such as potential for trailing-zero sharing, reducing metadata overhead. VL-RLE substantially improves compression and decompression throughput on datasets with long run lengths (e.g., DE), common in IoT workloads, and introduces no overhead on datasets with minimal repetition (e.g., DD, ARL = 1) since it removes fixed-marker encoding and unnecessary preprocessing.

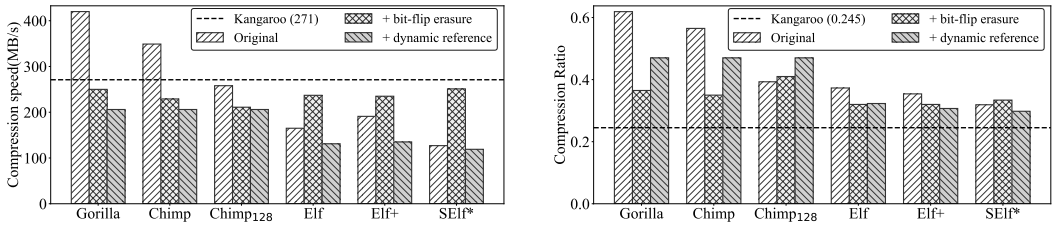
Each module individually improves compression ratio, with BF contributing the largest gain (up to 39.9%) by increasing zero density in mantissa bits. Combining BF with DR further benefits high-variability datasets, yielding an average 29% improvement. Adding VL lowers the compression ratio floor and simultaneously improves throughput, achieving on average 7% better compression, 50% higher compression speed, and 88.9% higher decompression speed, effectively offsetting BF and DR overheads.

*Kangaroo* is not universally optimal. On highly redundant datasets such as DE, disabling BF slightly improves compression ratio (from 0.00096 to 0.00091), because the overhead of storing erasure markers outweighs the benefit of removing trailing zeros. With an average run length of 150,528 under our 1000-value block configuration, most values lack an intra-block reference, making BF less effective. In such cases, increasing block size or disabling BF can mitigate this effect.

### 6.5 Cross-Algorithm Evaluation of Kangaroo Components

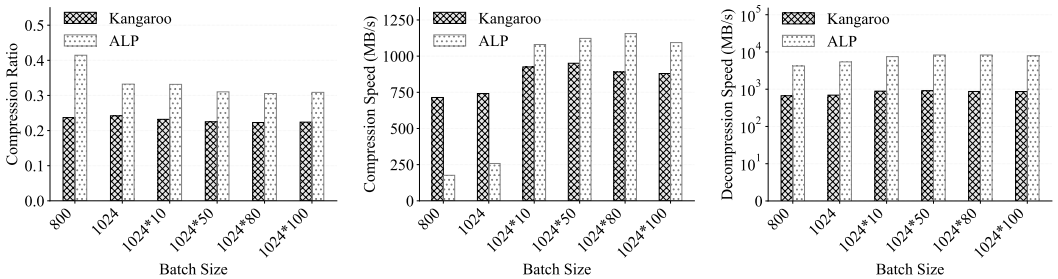
We integrate *Kangaroo*'s Bit-Flip Erasure (BF) and Dynamic Reference (DR) into six streaming compressors and evaluate their impact on compression ratio and compression speed in Figure 12. For compressors without existing erasure (e.g., *Gorilla* and *Chimp*), BF trades off some compression speed to achieve a significant improvement in compression ratio. For compressors with existing erasure (*Elf*, *Elf+*, and *SElf**), the marginal gain of BF diminishes, and *SElf** slightly degrades due to conflicts with its adaptive bit-selection model; however, replacing the original erasure stage with BF substantially improves throughput (e.g., 1.98× speedup for *SElf**), demonstrating a favorable efficiency–overhead trade-off. DR improves compression for predecessor-based compressors but can slightly reduce speed due to expanded reference search. For *SElf**, DR achieves a 6.58% compression gain with 6.30% speed loss, while for *Elf* and *Elf+*, DR yields limited benefit but noticeable slowdown, as strong existing erasure leaves little residual structure and DR introduces extra memory accesses and branching.

Applying BF or DR to $Chimp_{128}$ results in worse compression ratio and speed, as BF increases trailing zeros, which hinders $Chimp_{128}$ from effectively mapping values to suitable references, and DR offers limited benefit since $Chimp_{128}$ already searches 128 prior values. Overall, no baseline




(a) Compression Speed (MB/s) (b) Compression Ratio

Fig. 12. speed and compression ratio improvement of bit-flip erasure and dynamic reference.




(a) Compression Ratio (b) Compression Speed (MB/s) (c) Decompression Speed (MB/s)

Fig. 13. Compression performance under different batch sizes.

augmented with individual components surpasses *Kangaroo* in compression ratio. While *Gorilla* and *Chimp* remain faster due to simpler designs, they achieve inferior compression. BF and DR exhibit synergistic effects: BF increases trailing-zero density, enlarging DR's effective reference pool and enabling better reference selection. Their joint design is essential for achieving state-of-the-art compression ratio and speed.

### 6.6 Impact of Batch Size on Performance

The performance of *ALP* is highly sensitive to batch size. We compare Kangaroo and *ALP* using a small batch of 800 and multiples of 1024 (1×, 10×, 50×, 80×, 100×), with experiments conducted using the C++ version of Kangaroo and the open-source SIMD C++ implementation of *ALP*. As shown in Figure 13, Kangaroo's compression ratio is largely insensitive to batch size and consistently outperforms *ALP*. For *ALP*, the compression ratio is lower at the small batch of 800, which does not meet its minimum requirement of 1024, and improves slightly with larger batches due to metadata sharing across vector groups, though the gain is limited. Kangaroo achieves better compression speed with large batches, benefiting from longer runs that avoid repeated processing, whereas *ALP* suffers at small batches because the sampling stage dominates; increasing the batch size amortizes sampling overhead and allows SIMD optimizations to take effect, but excessively large batches (e.g., 100×1024) may slightly reduce compression speed due to increased outliers. For decompression, *ALP* consistently outperforms Kangaroo across all batch sizes as it avoids sampling and fully leverages SIMD optimizations, with larger batches further improving performance. Overall, *ALP* is better suited for decompression-intensive workloads, while Kangaroo excels in streaming small-batch or compression-ratio-critical scenarios, mainly because *ALP*'s shared parameters cannot fully adapt to datasets, resulting in more outliers and higher storage overhead.

## 7 Related Work

In this section, we introduce the algorithms used for lossless compression of floating-point values.

### 7.1 General Compression Algorithms

General compression algorithms aim to support diverse data types through distinct technical approaches, making specific trade-offs between compression ratio, speed, and computational overhead. Mainstream algorithms such as *LZ77* [5] employs sliding-window dictionary encoding by detecting repeated data patterns and replacing them with compact representations, demonstrating high efficiency for data with significant byte-level redundancy like text and source code. *Brotli* [6] combines static and dynamic dictionaries to achieve deep compression for web resources through context modeling. As a modern representative, *Zstandard (Zstd)* [7] integrates a multi-strategy architecture including dictionary matching and finite state entropy coding to achieve a highly configurable balance between speed and compression ratio. In contrast, *XZ* [8] utilizes large dictionary windows and high-order statistical models to pursue maximum compression rates at the cost of higher computational and memory overhead. While these algorithms perform excellently in their respective target scenarios, their general design often leads to efficiency limitations when meeting the high throughput and low latency requirements of streaming floating-point compression.

### 7.2 Floating-Point Specialized Compression Algorithms

General compression algorithms exhibit suboptimal performance for floating-point data. Consequently, specialized compression have emerged, where model-prediction-based approaches[44–48] are less prevalent than XOR-based methods due to their higher computational overhead. XOR-based compression techniques primarily fall into three categories: (1) immediate-predecessor-based compression algorithms, (2) sliding-window-based compression algorithms, and (3) batch-based compression algorithms.

*7.2.1 Immediate-predecessor-based Compression Algorithms.* Floating-point compression algorithms based on immediate predecessors exploit correlations between consecutive data points for high efficiency. This relies on a property of IEEE 754 representation: adjacent values tend to have similar high-bit and low-bit patterns, reflected in the leading and trailing zeros of XOR results. *Gorilla* [15] is a baseline method that XORs adjacent values and encodes leading zeros (5 bits) and trailing zeros (6 bits) separately. Subsequent improvements include *Chimp* [16], which optimizes non-uniform leading-zero distributions with 3-bit approximate encoding; *Elf* [17], which introduces mantissa bit-erasure and shows that 4-bit Decimal Significand Count markers suffice for lossless reconstruction; *Elf+* [18], which further leverages continuity in Decimal Significand Count for encoding efficiency; and *SElf** [19], which uses adaptive Huffman coding and dynamic programming to jointly optimize leading and trailing zeros, improving compression ratio at some cost to speed. However, all these methods rely solely on the immediate predecessor as the reference, ignoring potentially better references within the historical window, which can degrade compression efficiency when consecutive values vary substantially.

*7.2.2 Sliding-window-based compression algorithms.* $Chimp_{128}$ [16] extends beyond adjacent-value encoding by maintaining a sliding window of 128 previous values to identify optimal XOR reference points. This method employs 14-bit LSB hashing to ensure real-time performance. However, by ignoring leading zeros patterns and not necessarily selecting the previous values that maximizes trailing zeros, the approach may fail to identify globally optimal reference points. This limitation suggests that improved window-based compression schemes considering both leading and trailing zero patterns remain an area for further research.

*7.2.3 Batch-based compression algorithms.* Batch-based compression algorithms achieve better compression ratios through comprehensive analysis of data characteristics. The *Buff* [25] algorithm proposes an integer/fraction separation architecture and precision-aware erasure strategy specifically for bounded low-precision data. The *ALP* [20] algorithm adaptively selects compression schemes through a two-level sampling mechanism: employing *pseudoDecimal* [49] conversion with *FFOR* [50] compression for low-precision data, and bit-packing with skewed dictionary techniques for high-precision data. While ALP's vectorized implementation delivers exceptional decompression speed, its compression performance degrades with outlier presence. *Elf** extends *SElf** [19] to batch processing mode by utilizing intra-batch statistics rather than inter-batch prediction. These batch-based methods require preliminary data analysis, leading to slower compression speeds compared to streaming methods, yet they achieve superior compression ratios over *SElf**. However, batch compression algorithms share limitations with general compression schemes—specifically, higher memory overhead and reduced compression speed—making them unsuitable for streaming scenarios.

*7.2.4 GPU-based compression algorithms.* Recent work has explored GPU-based lossless floating-point compression to mitigate data transfer bottlenecks in scientific computing and analytics workloads. Delta-based compressors such as GFC [51] and MPC [52] exploit warp-level parallelism and bit-level transformations to achieve high throughput, while transform-based pipelines such as ndzip-GPU [53] parallelize predictive coding and bit transposition across thousands of threads. Neural-network-based compressors such as Dzip [54] represent an emerging direction, but their throughput remains several orders of magnitude lower than traditional compressors, limiting practical deployment. More recently, G-ALP [55] extends the ALP integer-mapping approach to GPUs and integrates it into the FastLanes [50] framework, enabling automatic schema selection and register-level decompression. However, G-ALP is designed for batch-oriented processing and relies on block-based metadata and exception handling, which introduce nontrivial overhead and latency. Overall, prior GPU compressors primarily target batch workloads and maximize parallel throughput, whereas *Kangaroo* explores a complementary design point by targeting low-latency streaming floating-point compression with lightweight dynamic reference selection and erasure mechanisms.

## 8 Conclusion and Future Work

This paper proposes *Kangaroo*, an efficient and compact lossless floating-point compression algorithm. Unlike immediate-predecessor-based reference algorithm, *Kangaroo* utilizes previous values as compression references and employs a dynamic reference search for rapid optimal reference retrieval. Additionally, the proposed bit-flip-based erasure strategy further enhances trailing zeros gains. Experimental results on 26 datasets demonstrate that *Kangaroo* achieves significant improvements in both compression ratio and speed compared to state-of-the-art methods.

Future work will focus on two directions: (1) developing GPU-accelerated implementations based on parallel computing architectures to further enhance real-time processing capability, and (2) creating an error-bounded lossy compression extension of *Kangaroo* for precision-tolerant applications, providing flexible solutions for diverse accuracy requirements.

## Acknowledgments

The work is supported by National Key R&D Program of China (2024YFF0617702), NSFC (Nos. U22A2025, 62232007, U23A20309), 111 Project (No. B16009), CCF-ApsaraDB Research Fund(No. CCF-Aliyun2024006).

## References


[1] Ruiyuan Li, Huajun He, Rubin Wang, Sijie Ruan, Tianfu He, Jie Bao, Junbo Zhang, Liang Hong, and Yu Zheng. Trajmesa: A distributed nosql-based trajectory data management system. *IEEE Transactions on Knowledge and Data Engineering*, 35(1):1013–1027, 2023.

[2] Alibaba Cloud. Lindorm database, 2024.

[3] Qianyu Ouyang, Chunhui Shen, Wenlong Yang, Peng Yu, Qiang Xiao, Jianhui Lei, Yadong Chen, Qilu Zhong, Xiang Wang, Yong Lin, et al. Lindorm-uwc: An ultra-wide-column database for internet of vehicles. *Proceedings of the VLDB Endowment*, 17(12):4117–4129, 2024.

[4] Chunhui Shen, Qianyu Ouyang, Feibo Li, Zhipeng Liu, Longcheng Zhu, Yujie Zou, Qing Su, Tianhuan Yu, Yi Yi, Jianhong Hu, et al. Lindorm tsdb: A cloud-native time-series database for large-scale monitoring systems. *Proceedings of the VLDB Endowment*, 16(12):3715–3727, 2023.

[5] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

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

[7] Y Collet. Zstd github repository from facebook, 2016. Retrieved May 4, 2025 from https://github.com/facebook/zstd.

[8] XZ Utils Contributors. The .xz file format, 2023. Retrieved May 4, 2025 from https://tukaani.org/xz/xz-file-format.txt.

[9] Ruiyuan Li, Zechao Chen, Ruyun Lu, Xiaolong Xu, Guangchao Yang, Chao Chen, Jie Bao, and Yu Zheng. Serf: Streaming error-bounded floating-point compression. *Proceedings of the ACM on Management of Data*, 3(3):1–27, 2025.

[10] Yang Shi, Xiangyu Zou, Xinyu Chen, Sian Jin, Dingwen Tao, Cai Deng, Yufan Chen, and Wen Xia. Machete: An efficient lossy floating-point compressor designed for time series databases. In *2024 Data Compression Conference (DCC)*, pages 532–541, Snowbird, UT, USA, 2024. IEEE, IEEE.

[11] Grant Wilkins, Sheng Di, Jon C. Calhoun, Zilinghan Li, Kibaek Kim, Robert Underwood, Richard Mortier, and Franck Cappello. Fedsz: Leveraging error-bounded lossy compression for federated learning communications. In *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*, pages 577–588, Jersey City, NJ, USA, 2024. IEEE.

[12] Bing Lu, Yida Li, Junqi Wang, Huizhang Luo, and Kenli Li. Zfp-x: Efficient embedded coding for accelerating lossy floating point compression. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1041–1050, St. Petersburg, FL, USA, 2023. IEEE, IEEE.

[13] Gongjin Sun and Sang-Woo Jun. Zfp-v: Hardware-optimized lossy floating point compression. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 117–125, Tianjin, China, 2019. IEEE, IEEE.

[14] Robert Underwood, Sheng Di, Jon C. Calhoun, and Franck Cappello. Fraz: A generic high-fidelity fixed-ratio lossy compression framework for scientific floating-point data. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 567–577, New Orleans, LA, USA, 2020. IEEE.

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

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

[17] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. Elf: Erasing-based lossless floating-point compression. *Proceedings of the VLDB Endowment*, 16(7):1763–1776, 2023.

[18] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, Songtao Guo, Ming Zhang, and Yu Zheng. Erasing-based lossless compression method for streaming floating-point time series. *arXiv preprint arXiv:2306.16053*, 2023.

[19] Zheng Li, Ruiyuan Li, Xiaolong Xu, Yi Wu, Chao Chen, Tong Liu, Jiaxing Shang, and Yu Zheng. Adaptive encoding strategies for lossless floating-point compression. *IEEE Internet of Things Journal*, 12(14):26071–26085, 2025.

[20] Azim Afroozeh, Leonardo X Kuffo, and Peter Boncz. Alp: Adaptive lossless floating-point compression. *Proceedings of the ACM on Management of Data*, 1(4):1–26, 2023.

[21] Noushin Azami, Alex Fallin, and Martin Burtscher. Efficient lossless compression of scientific floating-point data on cpus and gpus. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 395–409, New York, NY, USA, 2025. Association for Computing Machinery.

[22] Fabian Knorr, Peter Thoman, and Thomas Fahringer. ndzip-gpu: efficient lossless compression of scientific floating-point data on gpus. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–14, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Nathaniel Fout and Kwan-Liu Ma. An adaptive prediction-based approach to lossless compression of floating-point volume data. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2295–2304, 2012.

[24] Peter Lindstrom and Martin Isenburg. Fast and efficient compression of floating-point data. *IEEE transactions on visualization and computer graphics*, 12(5):1245–1250, 2006.
[25] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J Elmore. Decomposed bounded floats for fast compression and queries. *Proceedings of the VLDB Endowment*, 14(11):2586–2598, 2021.
[26] Yuanyuan Yao, Lu Chen, Ziquan Fang, Yunjun Gao, Christian S. Jensen, and Tianyi Li. Camel: Efficient compression of floating-point time series. *Proc. ACM Manag. Data*, 2(6), December 2024.
[27] National Ecological Observatory Network (NEON). Barometric pressure (dp1.00004.001), 2025.
[28] InfluxData. Influxdb 2.0 sample data, 2023. Retrieved May 4, 2025 from https://github.com/influxdata/influxdb2-sample-data.
[29] Alex Kozlov. Daily electricity price and demand data, 2020.
[30] Mirror-Traffic. Highway merge interaction dataset, 2025.
[31] Julian Bock, Robert Krajewski, Tobias Moers, Steffen Runde, Lennart Vater, and Lutz Eckstein. The ind dataset: A drone dataset of naturalistic road user trajectories at german intersections. In *2020 IEEE Intelligent Vehicles Symposium (IV)*, pages 1929–1934, Las Vegas, NV, USA, 2020. IEEE.
[32] European Climate Assessment & Dataset (ECA&D). London weather data, 2021. Retrieved May 4, 2025 from https://www.kaggle.com/datasets/emmanuelfwerr/london-weather-data.
[33] Télécom SudParis. Dataset of legitimate IoT data. https://www.data.gouv.fr/en/datasets/dataset-of-legitimate-iot-data/#/resources, 2022. Accessed: May 4, 2025.
[34] Daqing Chen and UCI Machine Learning Repository. Online retail dataset. https://archive.ics.uci.edu/dataset/352/online+retail, 2012. Accessed: May 4, 2025.
[35] Xinyu Chen, Jiannan Tian, Ian Beaver, Cynthia Freeman, Yan Yan, Jianguo Wang, and Dingwen Tao. Fcbench: Cross-domain benchmarking of lossless compression for floating-point data. *Proceedings of the VLDB Endowment*, 17(7), 2021.
[36] Spring. Financial data set used in infore project. Zenodo, 2020. Data set.
[37] Blockchair. Blockchair database dumps, 2025. Retrieved May 4, 2025 from https://gz.blockchair.com/bitcoin/transactions/.
[38] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. Get real: How benchmarks fail to represent the real world. In *Proceedings of the Workshop on Testing Database Systems*, DBTest '18, New York, NY, USA, 2018. Association for Computing Machinery.
[39] World Food Programme (WFP). Global food prices database (wfp), 2021.
[40] Timo Poutanen. Global video game sales & ratings, 2023. Retrieved May 4, 2025 from https://www.kaggle.com/datasets/thedevastator/global-video-game-sales-ratings.
[41] PassMark Software. Ssd and hdd benchmarks, 2022.
[42] Transaction Processing Performance Council(TPC). Tpcx-bb, 2023. Online.
[43] Google. Snappy | a fast compressor/decompressor, 2023. Retrieved May 4, 2025 from https://github.com/google/snappy.
[44] Y. Sazeides and J.E. Smith. The predictability of data values. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 248–258, 1997.
[45] Peter Lindstrom and Martin Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, 2006.
[46] P. Ratanaworabhan, Jian Ke, and M. Burtscher. Fast lossless compression of scientific floating-point data. In *Data Compression Conference (DCC'06)*, pages 133–142, 2006.
[47] Martin Burtscher and Paruj Ratanaworabhan. High throughput compression of double-precision floating-point data. In *2007 Data Compression Conference (DCC'07)*, pages 293–302, 2007.
[48] Martin Burtscher and Paruj Ratanaworabhan. Fpc: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers*, 58(1):18–31, 2009.
[49] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. Btrblocks: Efficient columnar compression for data lakes. *Proceedings of the ACM on Management of Data*, 1(2):1–26, 2023.
[50] Azim Afroozeh and Peter Boncz. The fastlanes compression layout: Decoding> 100 billion integers per second with scalar code. *Proceedings of the VLDB Endowment*, 16(9):2132–2144, 2023.
[51] Molly A. O'Neil and Martin Burtscher. Floating-point data compression at 75 gb/s on a gpu. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, New York, NY, USA, 2011. Association for Computing Machinery.
[52] Annie Yang, Hari Mukka, Farbod Hesaaraki, and Martin Burtscher. Mpc: A massively parallel compression algorithm for scientific data. In *Proceedings of the 2015 IEEE International Conference on Cluster Computing*, CLUSTER '15, page 381–389, USA, 2015. IEEE Computer Society.
[53] Fabian Knorr, Peter Thoman, and Thomas Fahringer. ndzip-gpu: efficient lossless compression of scientific floating-point data on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage*

*and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[54] Mohit Goyal, Kedar Tatwawadi, Shubham Chandak, and Idoia Ochoa. Dzip: improved general-purpose loss less compression based on novel neural network modeling. In *2021 Data Compression Conference (DCC)*, pages 153–162, 2021.

[55] Sven Hepkema, Azim Afroozeh, Charlotte Felius, Peter Boncz, and Stefan Manegold. G-alp: Rethinking light-weight encodings for gpus. In *Proceedings of the 21st International Workshop on Data Management on New Hardware*, DaMoN '25, New York, NY, USA, 2025. Association for Computing Machinery.

Received October 2025; revised January 2026; accepted February 2026