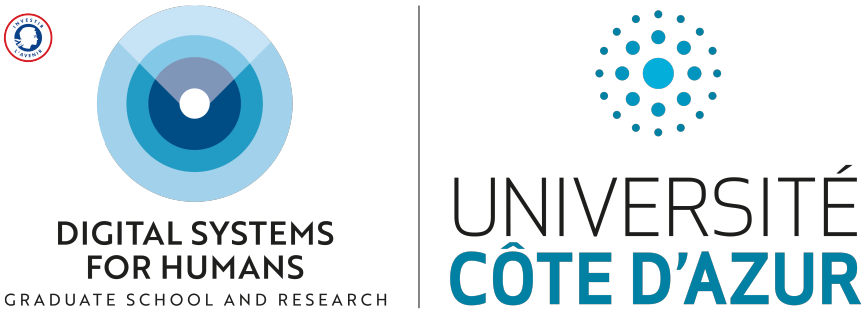
Integer Array Compression and Transmission Optimization using Bit Packing



Software Engineering Project 2025

Université Côte d'Azur

Author: Bierhoff THEOLIEN

Date: November 2025

Supervisor: Jean Charles Régin

Course: Software Engineering Project

Contents

Abstract

This report presents a comprehensive study of integer compression methods based on the principle of *bit packing*, a low-level optimization that reduces storage and transmission costs by encoding integers using only the minimum number of bits necessary to represent their magnitude. Instead of allocating a full 32 bits to every integer, the approach dynamically adjusts the bit-width per element, enabling efficient use of memory and bandwidth. Such efficiency gains become particularly significant in contexts where datasets contain small or sparse values.

Four distinct compression strategies are designed and evaluated: *overlapping*, *non-overlapping*, *overflow-aware*, and *negative-aware*. Each one introduces a unique trade-off between implementation complexity, packing density, and access latency. The overlapping method achieves the highest compression ratio by allowing integers to cross 32-bit boundaries, whereas the non-overlapping approach sacrifices some density for easier decompression and direct access. The overflow-aware technique introduces a secondary data zone to store large outliers separately, avoiding bit-width inflation across the main dataset. Extending this concept, the negative-aware method manages signed integers by isolating negative values and recording their positions, thereby maintaining the correctness of decompression.

Beyond algorithmic design, the report includes an extensive empirical evaluation covering compression throughput, decompression speed, and transmission efficiency. Benchmarks simulate real network conditions using measured mobile bandwidth and latency from Nice, France, to quantify how compression interacts with communication parameters. The findings reveal the precise circumstances in which compression provides substantial time savings, demonstrating how data distribution and network characteristics jointly influence performance.

1 Introduction

The transmission of integer arrays is central to modern data systems, yet integers are typically represented using a fixed 32-bit format, leading to inefficiencies when values are small. Bit packing addresses this issue by using only the minimal number of bits per integer while preserving direct access. This project implements multiple compression strategies, measures their efficiency, and evaluates their suitability for real-world network conditions.

2 Background and Related Work

2.1 Bit Packing Principle

Bit packing encodes each integer using only as many bits as needed. If the largest integer requires k bits, n integers need $n \times k$ bits instead of $32n$. Random access to the i-th element must be maintained without full decompression.

2.2 Implemented Compression Strategies

- **Overlapping Bit Packing (CrossIntBitPacking)** — allows integers to cross 32-bit boundaries, improving density.
- **Non-overlapping Bit Packing (NonCrossIntBitPacking)** — confines integers within 32-bit boundaries for simpler retrieval.
- **Overflow-aware Bit Packing (OverflowBitPacking)** — uses a secondary overflow area for large outliers.
- **Negative-aware Bit Packing (NegativeIntegerBitPacking)** — handles negative integers via overflow encoding.

3 Methodology

3.1 Architecture

A base interface `BitPacking` defines the methods `compress()`, `get()`, and `decompress()`. The `Compressor` factory class automatically selects an appropriate implementation based on dataset sparsity and sign distribution. Sparsity S is measured as

$$S = \frac{\sigma}{\mu}, \tag{1}$$

where σ is the standard deviation and μ the mean of bit widths.

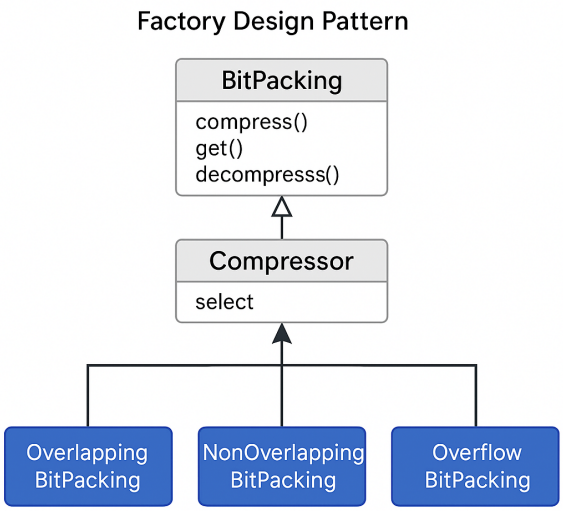


Figure 1: Factory Design Pattern used to select compression strategy automatically based on input characteristics.

3.2 Overflow and Negative Handling

To mitigate inefficiency caused by a few large outliers, `OverflowBitPacking` separates values whose bit length significantly exceeds the mean plus a sparsity-scaled deviation. These values are stored in a secondary compressed array, called the overflow area. A high bit in the normal area indicates whether an element points to this overflow region.

The notion of *sparsity* is derived from the statistical dispersion of bit lengths within the array. It is computed as the ratio of the standard deviation (σ) to the mean (μ) of the bit-width distribution:

$$S = \frac{\sigma}{\mu}. \tag{2}$$

This dimensionless metric behaves similarly to the coefficient of variation, quantifying how much the bit lengths vary relative to their average. When the bit-length distribution roughly follows a normal law, a high sparsity indicates that a few integers deviate significantly from the mean — these become natural candidates for the overflow area. Conversely, when sparsity is low, most integers are of comparable size, and overflow separation brings little benefit. In essence, sparsity plays a role analogous to the z-score thresholding in Gaussian models, where outliers are identified as values lying several standard deviations away from the mean.

Handling negative integers introduces a different kind of complexity. In Java, integers are stored using a two's complement representation, where the most significant bit (MSB) acts as a sign indicator. As a result, negative numbers exhibit leading ones rather than leading zeros. This representation complicates bit-packing because direct application of the standard unsigned logic would misinterpret sign bits as part of the magnitude, leading to incorrect decompression.

To address this, the `NegativeIntegerBitPacking` class treats all negative numbers as special overflow cases. Their absolute values are stored in a secondary compressed array (the overflow area), while the positions of these negative entries are tagged within the normal area using one additional control bit. This approach maintains correctness and direct access but increases the bit-width requirement slightly.

It is important to note that this negative-handling mechanism is only a first exploration. The implementation correctly supports signed integers, but its performance has not yet been rigorously evaluated. Future work could refine this component by incorporating more compact sign encoding schemes or by extending the overflow-based logic to signed-magnitude hybrid models.

4 Experimental Protocol

4.1 Benchmarking and Network Model

The class `CompressionPerformanceEvaluator` generates synthetic datasets of varying sizes (256–8192 elements) and bit-width distributions. These datasets are created through command-line parameters (CLI flags) that control properties such as maximum bit width, spike ratio, and minimum spike bit size. This parameterization allows fine-grained control over the data distribution — from uniformly small integers to sparse arrays containing occasional large spikes — providing a systematic way to evaluate how each compression type behaves under different statistical conditions.

The benchmark then runs all compression implementations and computes several metrics: throughput (in million integers per second), average bits per integer, and total transmission time.

The network model includes configurable latency (in milliseconds) and bandwidth (in Mbps). The total transmission time is computed as:

$$T_{total} = T_{compress} + T_{transfer} + T_{decompress}, \tag{3}$$

where $T_{transfer} = L + \frac{B_{compressed}}{\text{bandwidth}}$ and L is the network latency.

By varying these parameters, the evaluation framework measures not only the raw computational performance of each compressor but also its effectiveness in real-world transmission scenarios, enabling direct comparisons between overlapping, non-overlapping, and overflow-aware strategies.

5 Results and Discussion

5.1 Empirical Results under Realistic Network Settings (Nice, FR)

To ground the analysis in a practical context, we used a baseline representative of mobile upload conditions in Nice, France: latency ~53 ms and upload bandwidth ~24.6 Mbps (used as the transmission bandwidth in our model).[1]

We ran multiple configurations by varying message size, maximum bit width for generated values (`-maxBits`), and the frequency/severity of spikes (`-spikeRatio`, `-spikeMinBits`). Below, we highlight the outcomes that differentiate the three strategies.

Small bit-width, massive transfer (`-maxBits=5`, message size 10^9). Overlapping: 5.00 bits/int; total pipeline time ~270,163 ms.
Non-overlapping: 5.34 bits/int; ~275,691 ms.
Overflow-aware: 15.36 bits/int; ~738,875 ms.
Takeaway: With uniformly small values and no spikes, overlapping is best; non-overlapping is close; overflow-aware adds overhead.

Small bit-width, tiny transfer (`-maxBits=3`, message size 10^3). All methods are beneficial by ~0.6–1.1 ms versus raw; differences are minimal because fixed latency dominates.
Takeaway: For small payloads, method choice matters little.

Medium bit-width (`-maxBits=16`, message size 10^9). Overlapping and non-overlapping both average 16 bits/int; non-overlapping is fastest end-to-end (~690,195 ms) vs overlapping (~735,917 ms); overflow-aware is worse (24.02 bits/int).
Takeaway: When k aligns with 32-bit boundaries, non-overlapping benefits from simpler packing/access.

[1] Values taken from a public measurement page; observed November 2025.

Boundary sensitivity (`-maxBits=13`, message size 10^9). Overlapping: 13.00 bits/int; ~593,157 ms.
Non-overlapping: rounded to 16.00 bits/int; ~687,932 ms.
Overflow-aware: 19.53 bits/int; ~925,312 ms.
Takeaway: When k does not divide 32, non-overlapping pays alignment overhead; overlapping wins decisively.

High bit-width (`-maxBits=17`, message size 10^9). Overlapping: 17.00 bits/int; beneficial (~778,590 ms).
Non-overlapping: forced to 32.00 bits/int; worse than raw (~1,344,830 ms).
Overflow-aware: 25.51 bits/int; beneficial but inferior to overlapping (~1,145,518 ms).
Takeaway: Above 16 bits/value, once alignment breaks, non-overlapping can become non-viable.

Rare large spikes (`-maxBits=5`, `-spikeRatio=0.01`, `-spikeMinBits=16`). For 10^3 ints: only overflow-aware is clearly beneficial (7.51 bits/int; savings ~0.90 ms). For 10^9 ints: overflow-aware yields large savings (~882,481 ms), while overlapping/non-overlapping can be worse than raw.
Takeaway: Even 1% spikes, if much larger (16+ bits), make overflow-aware dominant.

Frequent large spikes (`-spikeRatio=0.50`, `-spikeMinBits=16`). All methods are worse than raw; overflow-aware averages 42.03 bits/int (too many spikes for overflow to help).
Takeaway: When spikes are common, compression can be counter-productive; consider raw transfer or redesign (multi-tier overflow, blockwise adaptivity).

Extreme spikes (`-spikeMinBits=31`, `-spikeRatio=0.01`). Overflow-aware dominates (7.43 bits/int) with large savings; overlapping/non-overlapping drift to ~31–32 bits/int and are not beneficial.
Takeaway: Overflow-aware excels when a tiny fraction of values are dramatically larger than the bulk.

5.2 Empirical Summary

Table 1 summarizes key results under various configurations (times are end-to-end pipeline times in milliseconds; lower is better).

Condition	**Recommended Strategy**	**Why**
k does not divide 32 (e.g., 13, 17)	Overlapping	Avoids alignment waste; packs across word boundaries.
k aligns with 32 and distribution is tight (e.g., 8, 16)	Non-overlapping	Simpler addressing; faster end-to-end.
Spiky distribution with small spike ratio ($\sim$ 1%) and spikes $\gg k$	Overflow-aware	Isolates large outliers; major bit savings.
Spikes are frequent ($\geq 50\%$)	None / reconsider compression	Overhead dominates; raw transfer or different codec.
Small messages ($\leq 10^3$ ints)	Any (difference negligible)	Fixed latency dominates total time.

Table 1: Empirical takeaways for choosing a compression strategy.

6 Conclusion and Perspectives

This study demonstrated how design choices affect both storage and transmission efficiency. Overlapping and non-overlapping bit packing achieve high throughput for compact data, while overflow-aware packing is robust to heterogeneity.

Limitations for negative integers: the current implementation treats negatives via an overflow path. This is correct but not yet optimized: it adds control-bit overhead, may increase random-access cost, and was not benchmarked. Future work should evaluate: (i) zigzag encoding to map signed values to small unsigned magnitudes, (ii) per-block sign bitmaps (pack signs separately to keep magnitudes dense), (iii) signed-magnitude or Golomb/Rice variants tuned to signed data, and (iv) SIMD-friendly pack/unpack for the negative path.

Additional research may explore SIMD vectorization (intrinsics), blockwise adaptation of k, and hybrid schemes combining delta encoding with bit packing.

Keywords: Integer compression, Bit packing, Network transmission, Overflow handling, Negative integers, Java benchmarking.

Webography

- Mobile network performance data for Nice, France: https://www.speedgeo.net/statistics/france/nice