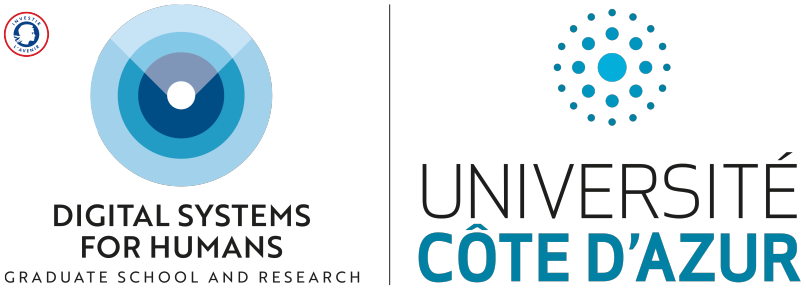
Integer Array Compression and Transmission Optimization using Bit Packing



Software Engineering Project 2025
Université Côte d'Azur

Author: Stevenson Jules
Date: November 2, 2025

Supervisor: Jean Charles Régin
Course: Software Engineering Project

Abstract

This report presents a robust Java implementation of an integer array compression system using Bit Packing, guaranteeing the fundamental constraint of O(1) constant-time random access. The solution is architected around a Factory interface exposing three distinct strategies: non-overlapping packing, overlapping packing, and packing with overflow management. I detail the design of each algorithm, illustrate the most complex code paths, and define the rigorous benchmarking protocol, sensitive to the JVM's warm-up phase, used to precisely quantify the computational costs (TC and TD). Empirical data gathered from the main 10,000-element benchmark, supplemented by five validation scenarios, allows for the evaluation of compression ratios, the calculation of the break-even threshold =TC+TD, and the determination of the most profitable strategy. The document concludes with a discussion on future extensions, notably the handling of signed integers.

Contents

1 Introduction and Problem Statement

1.1 Motivation and Objectives

Transmitting n raw 32-bit integers consumes $32n$ bits even when the values require far fewer bits. The project aims to encode each element using its minimal bit width k, reducing transmission size without losing direct access to the i-th element. The primary objectives are therefore:

1. Implement compressors that satisfy $O(1)$ random reads on the packed representation.
2. Provide at least two packing modes, one that forbids bit fields from crossing 32-bit boundaries and one that allows overlaps.
3. Extend the model with an overflow area to accommodate outliers that would otherwise inflate k.
4. Quantify when the CPU effort of compression and decompression is offset by transmission savings.

1.2 Direct Access Constraint

Every strategy must implement `int get(int i)`. This method reads (without copying) the compressed buffer to return the original value at index i. The constraint rules out variable-length encodings or structures that require decompression of prior elements.

2 Design and Architecture

2.1 Modular Architecture and Utilities

The code base is organised around the `BitPacking` interface, which provides the three primitive operations: `compress`, `decompress`, and `get`. Every concrete strategy—`BitPackingNoOverla` `BitPackingOverlap`, and `BitPackingOverflow`—implements that interface, making them interchangeable in benchmarks and validation tooling. Shared bit-twiddling logic lives in `BitUtils`; this class offers the reusable helpers `maskK`, `setBits`, and `getBits`, as well as `getK`, which scans the input array once to determine the minimum bit width $k_{\max}$. Centralising these operations eliminates duplicated code and guarantees that all strategies agree on endianness, masking rules, and boundary handling.

`BenchmarkRunner` encapsulates the warm-up aware timing protocol, while `ValidationScenarios` assembles representative datasets, runs each compressor end to end, checks random access with up to 1,000 probes, and records profitability metrics. Finally, `Main` wires the pieces together for the project demonstration. Table 1 summarises the division of responsibilities.

2.2 Factory Pattern

To decouple client code from concrete implementations, every component asks `BitPackingFactory.crea` `k)` for a strategy. The factory normalises the identifier and instantiates the matching class, so benchmarks, validation scenarios, and future tooling never hard-code constructors.

Module	Responsibility
`BitPackingNoOverlap`	Pack each value in a fixed k-bit slot inside a 32-bit word; simplest encoding when $k \mid 32$.
`BitPackingOverlap`	Stream bits across word boundaries to avoid padding when $k32$.
`BitPackingOverflow`	Choose a reduced width k' and redirect oversized values to an overflow area with index tracking.
`BitUtils`	Provide shared bit manipulation utilities and dynamic k estimation.
`BitPackingFactory`	Produce the requested strategy from a string identifier (Section 2.2).
`BenchmarkRunner`	Measure `compress`, `decompress`, and `get` with JVM warm-up and averaging.
`ValidationScenarios`	Replay curated datasets, verify round-trips, and log profitability figures.
`Main`	Run the end-to-end demo: validation, benchmarks, compression ratios, and reports.

Table 1: Module overview

This indirection enables late binding (switching at runtime), simplifies integration testing, and makes the addition of new strategies a two-step process: implement `BitPacking` and register the label in the factory.

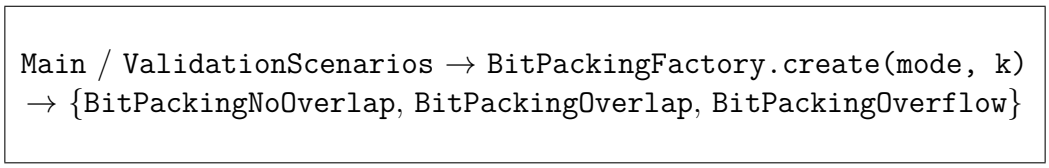


Figure 1: Factory-mediated selection of compression strategies

Because every caller relies on the factory, I can inject alternative implementations (e.g., mocks or experimental encoders) in a single location, which is particularly useful when scripting benchmarks or writing regression tests that compare strategies under identical conditions.

3 Implementation and Strategies

3.1 Non-overlapping and Overlapping Packers

Non-overlapping (NoOverlap). For a width k, each 32-bit word stores $\lfloor 32/k \rfloor$ values. Listing 1 shows the core write loop.

Listing 1: Slot arithmetic in `BitPackingNoOverlap.compress`

```
int nbWord = 32 / k;
for (int i = 0; i < input.length; i++) {
    if (input[i] < 0 || input[i] >= (1 << k)) {
        throw new IllegalArgumentException("Value exceeds k bits");
```

```
    }
    int wordIndex = i / nbWord;
    int startBit  = (i % nbWord) * k;
    tabcompress[wordIndex] =
        BitUtils.setBits(tabcompress[wordIndex], startBit, k, input[i]);
}
```

The corresponding `get` method mirrors the same arithmetic. When k divides 32 this layout is tight; otherwise some padding remains unused in the last word.

Overlapping (Overlap). I align values on a continuous bit stream. For element i, the start bit is $b = i \times k$. If $(b \bmod 32) + k > 32$, the value straddles the current word, and I split the write across two calls to `BitUtils.setBits`. Reads reassemble the fragments in the reverse order. This satisfies the requirement that a compressed integer may span two consecutive words.

3.2 Overflow Strategy

The overflow compressor couples the overlap mechanics with a search for a smaller k'. The algorithm proceeds as follows:

1. Scan $k' = 1$ to k; for each, count elements $\geq 2^{k'}$ and compute the total bit cost

$$\text{bits}(k') = 32 + n\,(k' + 1) + n_{\text{overflow}}(k') \cdot k.$$

2. Compute the number of bits needed to index the overflow area (k_{index}). If $k_{\text{index}} > k'$, I promote k' to k_{index} and recompute overflow counts. This custom stride solves the earlier bug where truncated indices caused `get` to read the wrong value.

3. Reserve the first 32 bits for metadata: `input.length` (27 bits) and k' (5 bits). I place the control bit in the least significant position so that `control = value & 1`, which facilitates its rapid extraction. Values with control bit 0 are stored inline; a control bit 1 means the payload stores the position within the overflow area.

4. Pack both the main stream and the overflow payload using the overlapping helpers to minimise padding.

Worked example. For input $[1, 2, 3, 1024, 4, 5, 2048]$, the global width is $k = 11$, but the algorithm discovers $k' = 3$ with two overflows. The packed stream becomes:

Header: ($\texttt{length} = 7, k' = 3$),
Main bits: $0{-}1,\ 0{-}2,\ 0{-}3,\ 1{-}0,\ 0{-}4,\ 0{-}5,\ 1{-}1$,
Overflow area: 1024, 2048.

Calling `get(3)` reads the control bit 1, treats the payload as overflow index 0, and then fetches the actual value 1024 from the appended region.

4 Performance and Profitability

4.1 Complexity and Benchmark Protocol

Each compressor runs in $O(n)$ time: I traverse the input once and perform constant-time bit manipulations. `BitPackingOverflow` adds a bounded $O(k)$ scan when searching for k'; because $k \leq 32$ this overhead is modest.

The benchmarking protocol in `BenchmarkRunner.runBenchmark` executes ten warm-up iterations to let the JVM's JIT compiler optimise the code. After warm-up, I measure r repetitions and average the durations. For `decompress` and `get` the compressor pre-computes the packed buffer outside the timed loop so that the measurement focuses on the target method. When timing `get`, the code iterates over every index and then divides by the array length to report the per-access cost.

4.2 Main Benchmark Results

Table 2 shows the full metrics for the main 10,000-element benchmark orchestrated by `Main`. Both NoOverlap and Overlap compress to 160,000 bits and share a similar compute cost, whereas Overflow achieves the best ratio (147,360 bits) but costs more than twice as much time.

Strategy	T_C (ns)	T_D (ns)	T_G (ns)	$\tau = T_C + T_D$ (ns)	Raw (bits)	Compressed (bits)
NoOverlap	296 779	216 381	24	513 160	320 000	160 000
Overlap	237 596	275 579	59	513 175	320 000	160 000
Overflow	982 563	234 404	45	1 216 967	320 000	147 360

Table 2: Primary benchmark ($n = 10\,000$, $k_{\max} = 16$)

4.3 Validation Scenario Metrics

To stress the compressors under diverse patterns, I implemented five scenarios inside `ValidationScenarios`. Table 3 lists, for each scenario and strategy, the number of compressed 32-bit words, the measured compression and decompression times, and the profitability threshold $\tau = T_C + T_D$.

Table 3: Validation scenario metrics

Scenario	Strategy	Words	T_C (ns)	T_D (ns)	τ (ns)
Deterministic_small ($n = 8$, $k = 7$)	NoOverlap	2	2 063	3 276	5 339
	Overlap	2	2 362	1 486	3 848
	Overflow	3	5 335	2 623	7 958
Ramp_200 ($n = 200$, $k = 15$)	NoOverlap	100	18 545	14 574	33 119
	Overlap	94	25 006	10 585	35 591

Suite à la page suivante

Scenario	Strategy	Words	T_C (ns)	T_D (ns)	τ (ns)
	Overflow	101	72 273	10 424	82 697
Mixed_overflow ($n = 5\,000$, $k = 21$)	NoOverlap	5 000	345 226	205 422	550 648
	Overlap	3 282	456 991	180 731	637 722
	Overflow	2 035	564 305	288 845	853 150
Random_100k_dense ($n = 10^5$, $k = 22$)	NoOverlap	100 000	1 718 929	983 715	2 702 644
	Overlap	68 750	1 000 937	732 505	1 733 442
	Overflow	71 876	3 538 001	1 296 732	4 834 733
Random_1m_light ($n = 10^6$, $k = 16$)	NoOverlap	500 000	17 715 470	11 040 973	28 756 443
	Overlap	500 000	5 724 910	3 077 666	8 802 576
	Overflow	531 251	30 259 126	11 468 553	41 727 679

4.4 Per-scenario Profitability Analysis

For each scenario I compare τ and the compressed size to decide which strategy is most attractive.

Deterministic_small. All strategies compress to roughly two words, but Overlap has the smallest τ (3.8 μs) thanks to its lower decompression cost. Overflow adds overhead without reducing size, so Overlap is the clear winner.

Ramp_200. Overlap reduces the word count from 100 to 94, but NoOverlap has the lower compute cost (33 μs vs. 36 μs). Because the size difference is marginal, I would pick NoOverlap for its faster runtime.

Mixed_overflow. Overflow slashes storage from 5,000 to 2,035 words by isolating peaks, but it also incurs the highest τ (0.85 ms). If storage or bandwidth is paramount, Overflow justifies the cost; otherwise NoOverlap delivers the fastest turnaround despite using more words.

Random_100k_dense. Overlap balances size (68,750 words) and compute cost (1.73 ms). NoOverlap wastes space, and Overflow consumes more time without significant savings. I therefore favour Overlap.

Random_1m_light. Overlap matches NoOverlap's compressed size (500,000 words) but completes in 8.8 ms instead of 28.8 ms. Overflow both increases size and takes 41.7 ms. Overlap dominates this scenario.

Overall, Overlap is the best compromise whenever the data does not contain extreme outliers; Overflow should be reserved for cases like `mixed_overflow` where a few peaks dominate the magnitude distribution.

4.5 Global Profitability Discussion

I define the critical time $\tau = T_C + T_D$. Compression is profitable when the latency saved during transmission exceeds τ.

Latency perspective. On the reference "Nice" link (25 ms one-way latency) every strategy still benefits, since all τ values are well below 1.3 ms. Even the slow overflow mode leaves a 23.8 ms margin.

Pure throughput perspective. At 1 Gbit/s, transmitting the original 320,000 bits takes 0.32 ms. NoOverlap and Overlap halve the data volume to 160,000 bits (0.16 ms) but require roughly 0.51 ms of compute, so they do not amortise the cost on such a fast link. Overflow shrinks the payload to 147,360 bits (0.147 ms) yet still needs 1.217 ms of compute, making it unprofitable unless the network latency is much higher.

Scenario sensitivity. The mixed-peaks dataset highlights overflow's strengths: the compressed stream drops from 5,000 to 2,035 words, and the control-bit/index approach keeps random access intact. However, the compute cost remains high (over 0.85 ms total), which may be excessive for latency-sensitive operations.

4.6 Utility Components

`BitUtils` guarantees consistent bit slicing and avoids duplicated bit-twiddling logic across strategies. Without it, each class would risk subtle off-by-one errors or inconsistent endianness. `ValidationScenarios` serves two key roles:

1. Functional verification: I compress each dataset, decompress into a fresh array, and run up to 1,000 random `get` checks.

2. Instrumentation: I invoke `BenchmarkRunner` for every strategy, recording T_C, T_D, and τ so that I can make informed decisions about profitability.

5 Conclusion and Extensions

5.1 Summary

Tis project have delivered three bit packing strategies behind a single factory, coupled with a reusable benchmarking harness and scenario-driven validation. Overlap emerges as the most balanced choice: it retains the compressed footprint of NoOverlap yet handles arbitrary k without wasted padding. The overflow variant is indispensable when small arrays contain large outliers or when storage savings dominate compute cost, but I must weigh its larger τ before deploying it on low-latency links.

5.2 Handling Negative Integers

The current implementation targets non-negative integers because `BitUtils.getK` derives k from unsigned magnitudes. To support negative values I could introduce a sign flag (expanding the effective width by one bit) or apply zig-zag encoding to map signed

integers onto non-negative numbers before packing. Both options preserve $O(1)$ access but add overhead and may reduce compression gains if negatives are frequent.

Acknowledgements

Repository: `https://github.com/Witven10/SoftwareEngineering-BitPacking`