

---

# Efficient Finite State Entropy: A Pedagogical Implementation and Performance Analysis

---

Anonymous Author(s)

Affiliation

Address

email

## Abstract

1 Finite State Entropy (FSE) is a table-based realization of Asymmetric Numeral  
2 Systems (ANS) that achieves compression ratios comparable to arithmetic coding  
3 while retaining speeds closer to Huffman coding. This project presents a two-stage  
4 implementation of FSE: first, a clear and pedagogical Python reference integrated  
5 into the Stanford Compression Library (SCL), and second, an optimized C++ port  
6 designed to explore the performance gap between a readable implementation and  
7 production-grade entropy coders. Both implementations maintain bitstream com-  
8 patibility and are evaluated using standard benchmarking frameworks, including  
9 `fullbench` and `lzbench`. Experimental results show that the Python implementa-  
10 tion matches other ANS variants in compression efficiency while outperforming  
11 pure-Python `rANS` and `tANS` in throughput. The C++ implementation achieves  
12 approximately a  $20\times$  speedup over Python, with an additional  $2\text{--}5\times$  improvement  
13 from optimized bit I/O, though it remains slower than Yann Collet’s highly opti-  
14 mized FSE. This work provides both an accessible reference for understanding  
15 FSE and a concrete case study in bridging theoretical compression algorithms and  
16 high-performance systems.

## 17 1 Introduction

18 Modern lossless compression systems typically combine an LZ-style front end with an entropy coder  
19 that converts symbol streams into compact bit representations near the Shannon limit. Classical  
20 entropy coding techniques exhibit a long-standing trade-off: Huffman coding is simple and fast  
21 but restricted to integer-length codewords, while arithmetic and range coding achieve near-optimal  
22 compression ratios at the cost of greater computational complexity.

23 Asymmetric Numeral Systems (ANS), introduced by Duda [1], reformulate entropy coding by  
24 maintaining a single integer state that jointly represents both previously encoded bits and the next  
25 symbol to be processed. Finite State Entropy (FSE), popularized by Collet and used in `Zstandard`,  
26 is a table-based realization of ANS that pushes most arithmetic into precomputed tables, enabling  
27 extremely fast encoding and decoding while retaining compression efficiency comparable to arithmetic  
28 coding.

29 The primary goal of this project is pedagogical and exploratory rather than purely competitive. We  
30 aim to (1) reimplement FSE in pure Python in a way that emphasizes clarity and correctness, (2)  
31 port this implementation to C++ while preserving bitstream compatibility, and (3) quantitatively  
32 evaluate how successive low-level optimizations narrow the performance gap to production-grade  
33 implementations. In doing so, we seek to illuminate both the algorithmic structure of FSE and the  
34 systems-level considerations that dominate real-world performance.

## 35 2 Literature Review

### 36 2.1 Asymmetric Numeral Systems

37 Duda’s ANS framework [1] demonstrates that arithmetic-coding-level compression efficiency can be  
38 achieved using a single integer state rather than an interval. Symbols are mapped to subsets of a finite  
39 state space in proportion to their probabilities, and encoding and decoding proceed by updating this  
40 state while emitting or consuming bits as needed.

41 Two major practical variants exist. Range ANS (rANS) resembles traditional range coding and relies  
42 on integer multiplications and divisions in its inner loop. Table-based ANS (tANS), in contrast,  
43 precomputes the necessary arithmetic into lookup tables, replacing expensive operations with memory  
44 accesses and simple bit manipulations.

### 45 2.2 Finite State Entropy

46 Finite State Entropy is a carefully engineered instance of tANS optimized for modern CPUs [2]. FSE  
47 begins by normalizing symbol frequencies so that their sum equals a power of two,  $2^{\text{tableLog}}$ , defin-  
48 ing the size of the state space. Symbols are then spread across this state space using a deterministic  
49 stepping scheme that approximates their desired probabilities.

50 From this construction, a decode table is built in which each state stores a symbol, the number of bits  
51 to read, and a base for computing the next state. Encoding proceeds in reverse by scanning symbols  
52 backward and using per-symbol transforms to reproduce the same state transitions. Crucially, while  
53 FSE allows flexibility in how frequencies are normalized and states are assigned, correctness depends  
54 on strict consistency between encoder and decoder tables; suboptimal choices primarily degrade  
55 compression efficiency rather than validity.

### 56 2.3 Related Implementations

57 Collet’s reference FSE implementation [3] serves as the performance baseline for this project.  
58 Zstandard [4] combines FSE and Huffman coding with an LZ77 front end, achieving state-of-the-art  
59 performance. Within the Stanford Compression Library (SCL) [5], several entropy coders—including  
60 Huffman, arithmetic coding, rANS, and tANS—are implemented in Python, providing a natural  
61 environment for a pedagogical FSE reference and comparative evaluation.

## 62 3 Methods

### 63 3.1 Python Reference Implementation

64 The Python implementation of FSE is integrated into SCL using its existing `DataEncoder` and  
65 `DataDecoder` abstractions. The implementation follows the standard FSE pipeline: frequency  
66 normalization, symbol spreading, decode table construction, and encoder table generation. Emphasis  
67 is placed on readability and explicitness, with each step implemented in a direct and traceable manner.

68 The resulting bitstream format stores the block size, final state, and payload bits in a manner  
69 compatible with both Python and C++ implementations. Although Python allows considerable  
70 flexibility in normalization and state assignment, the implementation adheres to conventional choices  
71 to produce competitive compression ratios while remaining easy to reason about.

### 72 3.2 C++ Implementation and Optimization

73 The C++ implementation mirrors the Python design to ensure bitstream compatibility, enabling  
74 cross-language validation. The core functionality is implemented as a static library, with optimized  
75 MSB and LSB bit readers and writers. To facilitate testing and benchmarking, the C++ codec is  
76 exposed to Python via `pybind11`, allowing existing SCL tests to be reused without modification.

77 Performance optimizations are introduced incrementally, including improved bit I/O and multi-block  
78 framing. This staged approach allows us to isolate the contribution of each optimization to overall  
79 throughput.

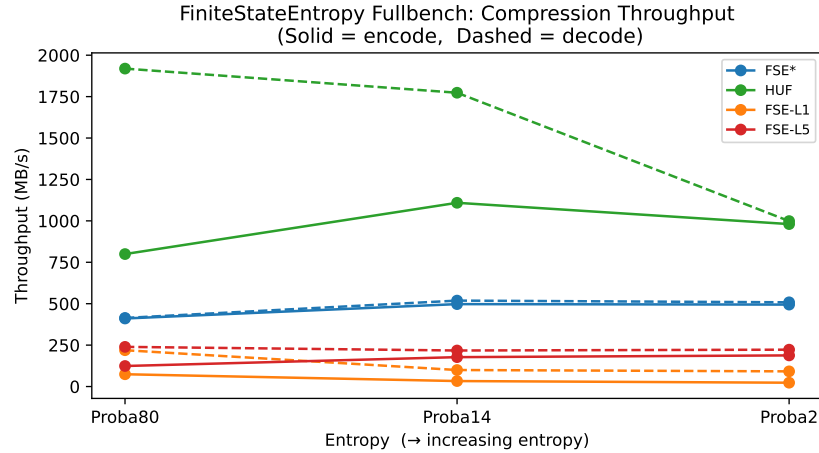


Figure 1: Entropy-only throughput comparison using fullbench.

### 3.3 Testing and Validation

Given the subtle consistency requirements between encoder and decoder tables, testing focuses on verifying round-trip correctness rather than enforcing a single “exact” construction. Unit tests validate normalization, spreading, and table construction, while integration tests ensure that Python-encoded streams can be decoded by the C++ implementation and vice versa. This cross-language validation provides strong assurance of correctness while accommodating the inherent flexibility of FSE’s design space.

### 3.4 Benchmarking Setup

Evaluation uses both custom Python benchmarks and established external harnesses. Entropy-only performance is measured using fullbench, while end-to-end performance is evaluated using lzbench. Experiments are conducted on the Canterbury and Silesia corpora as well as synthetic distributions, reporting compression ratio and encode/decode throughput.

## 4 Results and Analysis

### 4.1 Python Performance

The Python FSE implementation achieves compression ratios comparable to other ANS variants and consistently outperforms pure-Python rANS and tANS in throughput, while remaining slower than Huffman coding. These results align with theoretical expectations: FSE avoids the arithmetic overhead of rANS but incurs more complexity than Huffman’s codeword lookup.

### 4.2 C++ Performance

Porting the implementation to C++ yields approximately a 20× speedup, with optimized bit I/O contributing an additional 2–5× improvement. Figure 1 compares entropy-only throughput against Collet’s reference FSE. While compression ratios are similar, a performance gap of roughly 2.5× remains, attributable to factors such as table rebuild overhead, conservative bit I/O, and lack of interleaved multi-state decoding.

End-to-end benchmarks using lzbench (Figure 2) show that this FSE implementation trails production codecs such as zstd and lz4, which benefit from sophisticated LZ front ends in addition to highly optimized entropy stages.

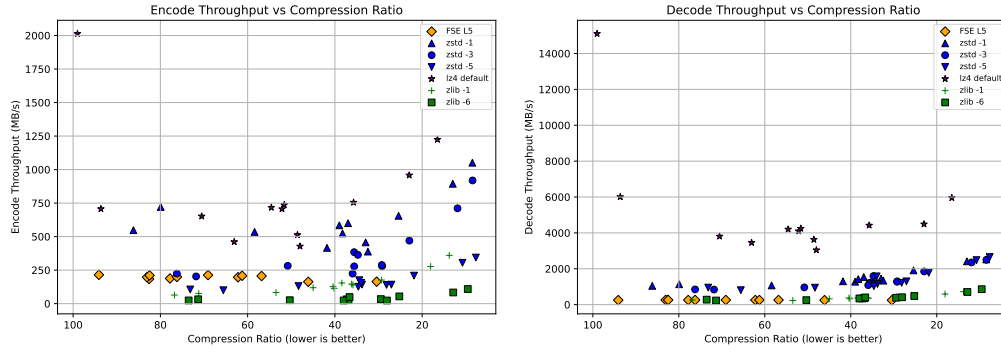


Figure 2: Encode and decode throughput versus compression ratio on the Silesia corpus using lzbenc.

## 5 Conclusions

This project presents a complete, readable implementation of Finite State Entropy in both Python and C++, illustrating how ANS-based entropy coding transitions from theory to practice. The Python version serves as a pedagogical reference, while the C++ port demonstrates the substantial performance gains achievable through low-level optimization. Although a gap remains relative to production-grade FSE, the results clarify where engineering effort yields the greatest returns.

## Limitations and Future Work

Several limitations suggest directions for future work. Header overhead could be reduced through normalized counter compression. Interleaved multi-state decoding would improve instruction-level parallelism, and further bit I/O optimizations could reduce hot-path overhead. Adaptive table sizing and improved handling of low-frequency symbols may also yield better compression–speed trade-offs.

## Acknowledgments

We thank Pulkit and Shubham for their mentorship and the EE274 teaching staff for guidance throughout this project.

## References

- [1] J. Duda. Asymmetric numeral systems. arXiv:1311.2540, 2013.
- [2] Y. Collet. Finite State Entropy. Fast Compression Blog.
- [3] Y. Collet. FiniteStateEntropy library. GitHub.
- [4] Y. Collet et al. Zstandard. GitHub.
- [5] Stanford Compression Library. GitHub.