Efficient FSE

Aayush Gupta[1] Mentor: Pulkit/Shubham[1]

[1]EE274: Data Compression

Problem & Motivation

- Modern lossless compressors combine an LZ-style front end with a fast entropy coder (Huffman, range coding, ANS).
- Finite State Entropy (FSE), a table-based ANS, underpins Zstandard's entropy stage and achieves near-arithmetic ratios with Huffman-like speed.
- **Goal:** Rebuild FSE in a readable way (Python) then see how far careful engineering can push performance toward production libraries (C++).

How FSE Works

Core idea: Maintain a single integer state that always lies in a fixed range. Each symbol occupies a subset of states proportional to its probability.

Table construction (per chunk) (size = $2^{\texttt{tableLog}}$):

1. **Normalize counts** so integer frequencies sum to $2^{\texttt{tableLog}}$.
2. **Spread symbols** across a state table so symbol s appears $\texttt{freq}[s]$ times.
3. **Build decode table:** for each state store $(\texttt{symbol}, \texttt{nbBits}, \texttt{newStateBase})$.
4. **Build encode transforms:** per-symbol parameters that reproduce the same state transitions in reverse.

One decode step:

current state

↓

DTable[state] → (symbol, nbBits, newStateBase)

↓

read nbBits bits

↓

$\texttt{newState} = \texttt{newStateBase} + \texttt{bits}$

- Each symbol makes the decoder read either k or $k+1$ bits; the average matches its optimal Shannon code length.

Direction: Encoder scans symbols backwards, pushing out low bits and updating a single FSE state. Decoder starts from the stored state and walks the decode table forward, reading $nbBits$ each step to recover the original sequence.

Bitstream format: `[size][final state][payload]`.

References (1/2)

[1] J. Duda, Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding, arXiv, 2013. https://arxiv.org/abs/1311.2540

[2] Y. Collet, "Finite State Entropy – a new breed of entropy coder" and follow-up posts, Fast Compression Blog. https://fastcompression.blogspot.com

[3] Y. Collet, FiniteStateEntropy library (FSE), GitHub. https://github.com/Cyan4973/FiniteStateEntropy

Implementation: Python → C++

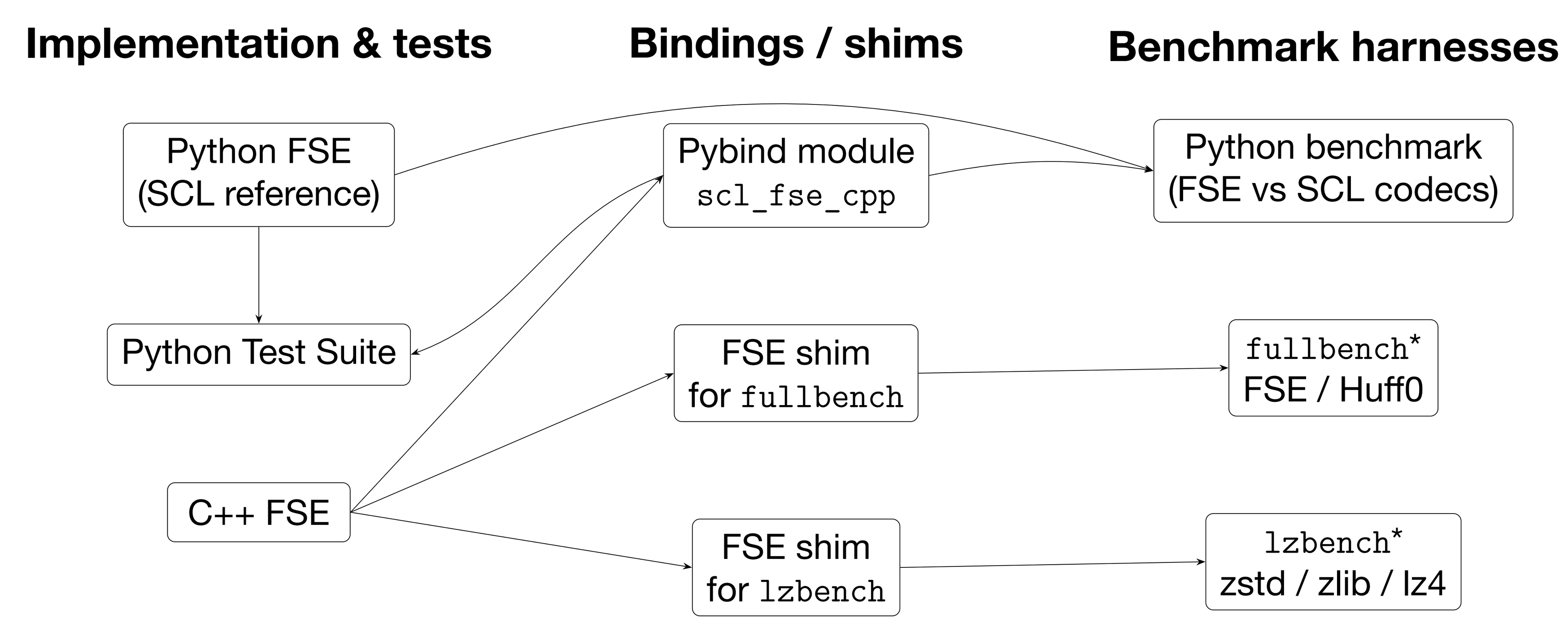
Python FSE (SCL):

- Implemented in the Stanford Compression Library as a clear, table-based ANS reference.
- Uses SCL's `Frequencies`/`DataBlock` for modeling, then:
  - normalizes counts to $2^{\texttt{tableLog}}$,
  - runs the FSE spread algorithm,
  - builds decode and encode tables.
- Correct and readable, but far too slow to be a practical codec.

C++ FSE:

- Reimplements the same core algorithm in C++.
- Exposed back to Python via `pybind` so existing SCL tests and synthetic benchmarks can be reused unchanged.
- Implement different levels, corresponding to increasing levels of optimization (ex: LSB bit I/O, chunked bit I/O)

Project Pipeline: Implmentation & Evaluation



*`fullbench` and `lzbench` are existing upstream harnesses; my shims register my codec with them.

Benchmarking Setup

- **Datasets:** Canterbury and Silesia corpora treated as byte streams (text, binaries, images), plus synthetic distributions.
- **Benchmarks:** Python SCL; C++ `lzbench` + `fullbench` (FiniteStateEntropy)
- **Metrics:** compression ratio and encode/decode throughput (MB/s).

References (2/2)

[4] Y. Collet et al., Zstandard compression format and source code, GitHub. https://github.com/facebook/zstd

[5] Stanford Compression Library (SCL), documentation and entropy coder implementations, GitHub. https://github.com/stanfordcompression/stanford_compression_library

Preliminary Results

Python reference (SCL):

- FSE slower than Huff, faster than rANS/tANS (same compression ratio).
- SCL codecs 10-100x slower than zlib/zstd.

C++ FSE:

- Python → C++ yields $\sim$20$\times$ speedup and better bit I/O in C++ gives another 2–5$\times$, but still $\sim$2.5$\times$ slower than the target FSE.

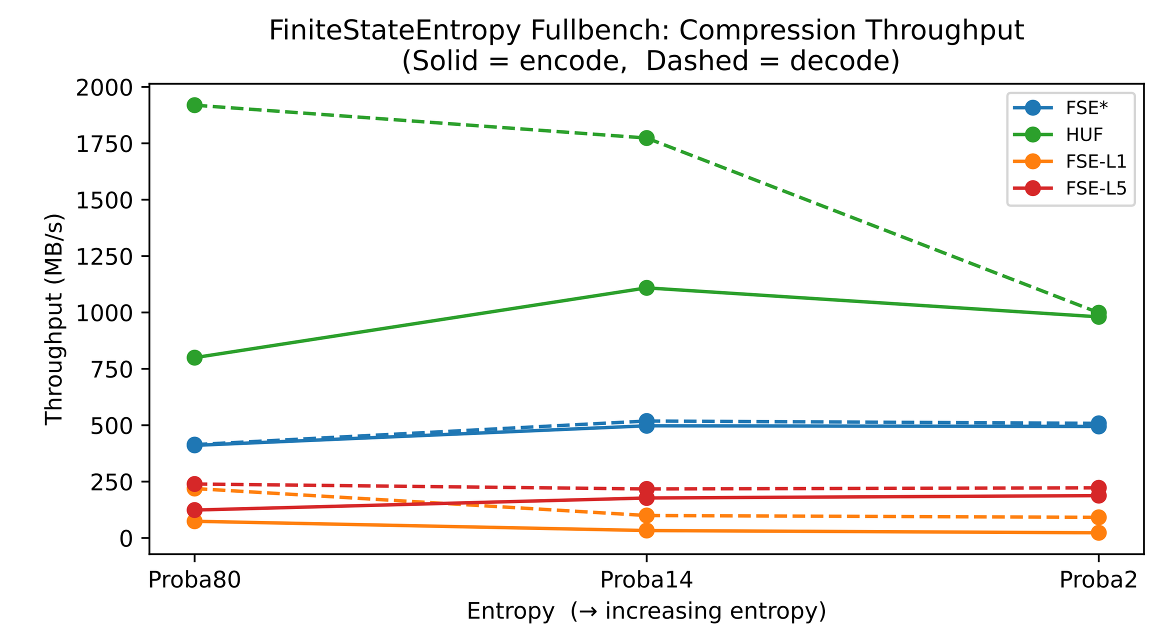


Figure 1. Entropy coder only: my C++ FSE vs Collet's FSE/Huff0 in (`fullbench`) (synthetic data)

- **lzbench:** My FSE trails in throughput and compression significantly (those codecs add powerful LZ front ends to already optimized entropy stages).

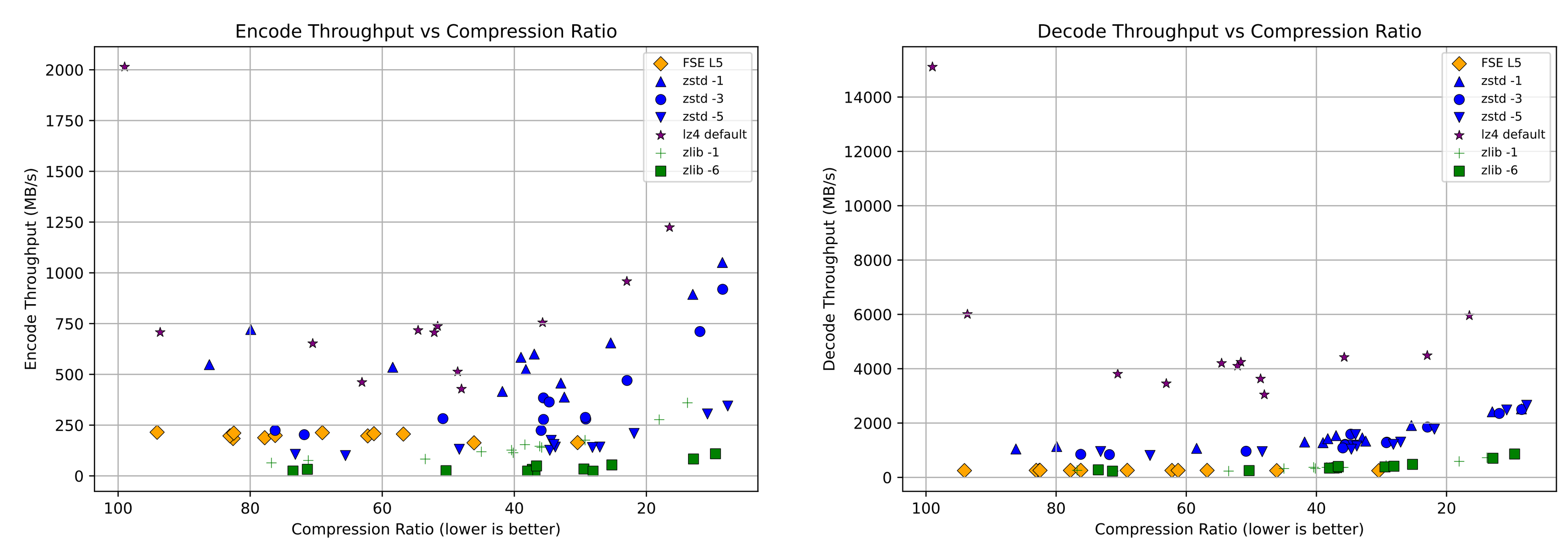


Figure 2. Throughput vs compression ratio on the Silesia corpus (`lzbench`). Each point represents a different file in the corpus.

Next Steps

- Per-block headers are large (1KB histogram) → NCount compression.
- Encoding/decoding multiple streams in parallel ("loop unrolling")
- Better handling of low frequency symbols
- Adaptive table sizes
- Further reduce bitreader/bitwriter overhead