File Corruption Repair using Reed-Solomon Codes

Theodor Negrescu

2025-07-04

Outline

Introduction

Reed-Solomon Codes

Fields

Finite Field Arithmetic

Efficient Multiplication

Transforms

Decoding Details

File Structure

File Processing Architecture

Benchmarks

Conclusion

Introduction

- **Error correction** is an important subject in computer science, relevant for backups, cloud storage, high-reliability systems, and many more applications.
- This presentation provides an overview of the theoretical background and implementation details of a utility which repairs file corruption using **Reed-Solomon codes**.
- The utility, named `rsarc` (Reed-Solomon Archive), is a command-line program written in Rust, available on GitHub and crates.io.

Reed-Solomon Codes

- **Reed-Solomon codes** are a major class of **error correcting codes**. An (n, k) Reed-Solomon code can encode k data symbols into n symbols by generating $n - k$ parity symbols, such that any k symbols can be decoded into the original data.
- The particular Reed-Solomon code used in this project is based on the paper "Novel Polynomial Basis and Its Application to Reed-Solomon Erasure Codes" by Sian-Jheng Lin, Wei-Ho Chung and Yunghsiang S. Han, which allows for efficient $O(n \log n)$ encoding and decoding.
- The encoder interprets the data as a polynomial and oversamples it to obtain redundant points.
- Since any n points of a polynomial of degree $n - 1$ uniquely determine it, the decoder can interpolate the polynomial and evaluate it at the corrupt points to recover the original data, as long as at least k symbols are undamaged.

Fields

- Reed-Solomon codes use polynomial arithmetic, which only functions correctly in a **field**.
- A field is a set over which addition and multiplication are defined, with usual properties of associativity, commutativity, distributivity, and inverses for addition and multiplication.
- Standard CPU arithmetic is not a field because modular arithmetic must have a prime modulus for multiplication to always be invertible. For example, $x * 2$ loses the most significant bit of x because of overflow.
- The field used in this project is $GF(2^{64})$, which is the **finite field** with 2^{64} elements, represented using 64 bit integers.

Finite Field Arithmetic

- The elements of $GF(2^{64})$ are polynomials with 64 coefficients, where each coefficient is a bit, stored as 64 bit integers.
- Finite field addition is polynomial addition, which can be performed using bitwise XOR.
- Finite field multiplication is polynomial multiplication followed by reduction modulo an irreducible polynomial of degree 64.
- The multiplicative inverse is obtained using the Extended Euclidean Algorithm. This is the standard method for inverses in finite fields, both $GF(2^n)$ fields as used here and $GF(p)$ fields used in cryptography.
- Division is by far the slowest operation, however this is insignificant because the factors used for encoding/decoding can be computed once and used for all codes (columns) in the file, so relatively few inversions are needed.

Efficient Multiplication

- Polynomial multiplication is natively supported by modern CPUs using **carry-less multiplication** instructions, which multiply two 64 bit polynomials and return a 128 bit polynomial.
- The reduction step can be performed using additional carry-less multiplications, without requiring any explicit loops or bit manipulation.
- The upper 64 bits of the 128-bit result are multiplied again by the irreducible polynomial, the upper bits of *that* result are again multiplied, then the three low 64 bit halves are XORed together.
- An implementation portable to old CPUs is provided using the Russian peasant algorithm, which uses shift and XOR operations in a loop instead of carry-less multiplication.

Transforms

- For efficient encoding and decoding, standard interpolation and evaluation methods cannot be used, as they are $O(n^2)$.
- Instead, $O(n \log n)$ FFT-like transforms are used, which convert between consecutive values of a polynomial and coefficients in a non-standard basis, with an arbitrary offset.
- For encoding, the data is converted to coefficients and then back to values, but with an offset of k, to obtain the parity symbols. This doubles the size of the data; excess symbols can be discarded if $n < 2k$, or more can be computed if $n > 2k$.
- The decoding process involves multiplying the incomplete data values by the **error locator polynomial** (polynomial which is zero at error locations), taking the **formal derivative**, and dividing by the derivative of the error locator polynomial.

Decoding Details

- The decoding process is justified by the product rule. Let d be the data and e the error locator polynomial. $(e * d)' = e' * d + e * d'$. At error points, $e(x) = 0$, so $(e * d)'(x) = e'(x) * d(x)$. Dividing by $e'(x)$ yields $d(x) = (e * d)'(x)/e'(x)$.
- The formal derivative exists in all fields, regardless of the absence of limits and continuity, and has most of the properties expected of a derivative. Computing the formal derivative in the non-standard basis is $O(n \log n)$, so does not change the total complexity.
- $e = (x - e_0)(x - e_1)(x - e_2)\ldots$, where e_i are error locations. It is computed by multiplying the factors together in a recursive algorithm which computes each half, then multiplies them by transforming to values, multiplying, and transforming back to coefficients, for total $O(n \log^2 n)$ complexity. This is also shared across all codes, same as the inverses.

File Structure

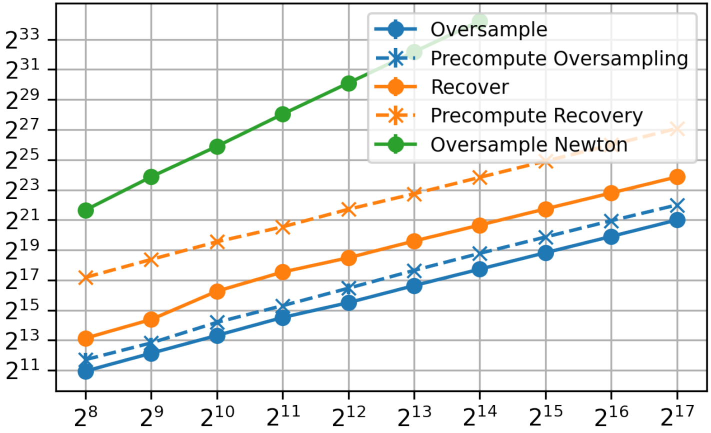
- The input file is not encoded using a single Reed-Solomon code, but is instead split into blocks of size specified by the user, and a chosen number of parity blocks is generated.
- Parity blocks, along with metadata and block hashes, are stored in a separate parity file. Hashes are required to know the error locations. The code used is an **erasure code**, which means it requires knowledge of which symbols are corrupt.
- Each block is **not** an independent code. Instead, each code spans across all data and parity blocks. In other words, the data and parity files are interpreted as a matrix, where each column is a code, and each row is a block.
- Zero padding is implicitly applied to the last block if the data file is not divisible by the block size. Additionally, zero blocks are added to pad to a power of two, as the code requires that n and k are powers of two.

File Processing Architecture

- Since codes are non-contiguous, reading them one at a time would perform one system call per symbol, and would be an inefficient disk access pattern.
- To mitigate this, multiple codes are read at once, filling a large buffer of interleaved symbols from multiple codes. The buffer is automatically sized based on available memory in order to optimize I/O as much as possible.
- After the large buffer is filled, multiple threads encode or decode codes from it in parallel. Once all codes have been processed, the large buffer is sent back to the reader thread to be filled again. If there is enough memory, two large buffers are created to read while processing.
- For writing, a single thread uses memory-mapped I/O to directly write the symbols to the output file. This relies on the operating system to coalesce writes and minimize flushes to disk. Since generally there is much more reading than writing - parity is usually only a percentage of data - this is acceptable.

Benchmarks

Benchmarks demonstrate the $O(n \log n)$ complexity of the implementation, compared to a $O(n^2)$ encoding algorithm using Newton interpolation and Horner's method for evaluation.



Conclusion

- This has been a short presentation of the theory and implementation behind the `rsarc` file corruption repair utility, based on Reed-Solomon codes, using transforms introduced in the paper "Novel Polynomial Basis and Its Application to Reed-Solomon Erasure Codes".
- Thank you for your time and attention!