UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Project Report

FILE CORRUPTION REPAIR USING REED-SOLOMON CODES

Theodor Negrescu

Coordonator științific

Cristian Rusu

București, ianuarie 2025

Abstract

The aim of this project to implement file corruption detection and repair using Reed-Solomon error-correcting codes.

The code used is an erasure code over the field GF(2^{64}), implemented using $O(n \log n)$ transforms in a polynomial basis introduced by Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang S. Han [3].

Finite field multiplication is implemented using carry-less multiplication, and division is implemented using the extended Euclidean algorithm.

The implementation is a CLI program which can generate parity data for a file, and later detect and repair corruption in that file.

The file is split into N blocks, and an arbitrary number of parity blocks M can be generated, for a total of $N + M$ blocks. Any N blocks are sufficient to recover the original data, so up to M corrupted blocks can be repaired.

As erasure codes require known error locations, a hash of each block is stored in the file header to detect corruption.

The file is not a single Reed-Solomon code, as that would require reading the entire file into memory, limiting the maximum file size which can be processed. Instead, the file is interpreted as a matrix with $N + M$ rows, and each column is an separate Reed-Solomon code.

Contents

Chapter 1

Introduction

1.1 Reed-Solomon Codes

Reed-Solomon codes are a well-known class of error-correcting codes used in a wide range of applications, from data storage to radio communication. They are based on polynomials over finite fields. [4]

The code used for this project is an erasure code over the field GF(2^{64}), implemented using $O(n \log n)$ algorithms introduced in [3].

The basic working principle of this type of Reed-Solomon code is the interpretation of data as values of a polynomial evaluated at points $\omega_0, \omega_1, \ldots, \omega_{k-1}$ in a finite field GF(2^n). As there is only one polynomial of degree $k-1$ or smaller passing through k points, any combination of at least k of the original and redundant points uniquely determines the original polynomial.

Any amount of redundancy can be added, limited only by the field size. As the chosen field is GF(2^{64}), the limit is effectively infinite.

An erasure code is a type of error-correcting code which requires that the locations of corrupted data are known. The code cannot be used to discover the locations of corrupted data by itself. In this case, hashes stored in the metadata of the parity file are used to determine error locations.

Other Reed-Solomon codes do locate errors without requiring hashes, but they are not used in this project, as hashes are a simpler and more efficient solution.

One common code is RS$(255, 223)$, which is used in CDs and DVDs, and uses 8-bit symbols (in the field GF(2^8)). The notation RS(n, k) denotes a code with n total symbols, with k data symbols and $n - k$ parity symbols. The code used in this project has no fixed n or k, they are specified by the user.

The code used in this project is also systematic, meaning that the original data is included in the output. Non-systematic codes do not include the original data, so the receiver must decode the code to obtain the original data, even if no corruption occurred.

1.2 Finite Fields

Finite fields, also known as Galois fields, are mathematical structures which define addition, multiplication, subtraction, and division over a finite set of elements [2] (as opposed to the more commonly known infinite fields, such as the rationals, reals, and complex numbers).

A field must satisfy the following properties:

- Associativity of addition and multiplication: $(a+b)+c = a+(b+c)$ and $(a\cdot b)\cdot c = a\cdot(b\cdot c)$
- Commutativity of addition and multiplication: $a + b = b + a$ and $a \cdot b = b \cdot a$
- Additive and multiplicative identity elements: $a + 0 = a$ and $a \cdot 1 = a$
- Additive inverses: for every a, there exists $-a$ such that $a + (-a) = 0$
- Multiplicative inverses: for every $a \neq 0$, there exists a^{-1} such that $a \cdot a^{-1} = 1$
- Distributivity of multiplication over addition: $a \cdot (b + c) = a \cdot b + a \cdot c$

The theorems of polynomial mathematics used in Reed-Solomon codes only hold in a field, however standard computer arithmetic does not form a field. Typical arithmetic supported natively by CPUs is fixed-size binary arithmetic with overflow, which is equivalent to arithmetic modulo a power of 2. Modular arithmetic only forms a field with a prime modulus, so it cannot be used directly for Reed-Solomon codes.

For example, the operation $x * 2$ is not invertible, as it is equivalent to a left shift, from which the most significant bit of x cannot be recovered.

Fortunately, it is possible to construct a field based on fixed-size integers, such as 64-bit integers.

In a finite field GF(p^m), where p is a prime number and m is a positive integer, the elements are polynomials of degree $m - 1$, with coefficients in GF(p). For the GF(2^n) case, an element in the field is interpreted as a polynomial with n coefficients, where each coefficient is a bit (i.e. a value in GF(2) = $\{0, 1\}$). For the purposes of this project, n is always 64, so the field is GF(2^{64}).

The notation ω_i is used to denote the integer i converted to an element of the field GF(2^{64}) by interpreting its bits as a polynomial, which is a no-op in code, as elements of GF(2^{64}) are stored as 64-bit integers.

It is important to note that these polynomials are not the same as the ones used in Reed-Solomon codes to represent data and parity information. Elements of GF(2^n) are simply n bit integers with more complex arithmetic. They are polynomials over GF(2), with n coefficients. Reed-Solomon polynomials can be arbitrarily long. They are polynomials over GF(2^{64}), with an arbitrary number of coefficients, and each coefficient is itself a polynomial over GF(2^2) with 64 coefficients.

Finite field addition is defined as polynomial addition. In a general field GF(p^m), this would be implemented as pairwise addition of the coefficients of two polynomials, modulo p.

In binary finite fields, addition is equivalent to XOR, as the coefficients are bits. Therefore, $x + x = 0$, and $x = -x$.

Multiplication is defined as polynomial multiplication, followed by reduction modulo an irreducible polynomial of degree 64 (with 65 coefficients, where the highest coefficient is 1).

The irreductible polynomial used for this project is $x^{64} + x^4 + x^3 + x + 1$ [5]. The choice of irreducible polynomial does not affect correctness, and all possible fields corresponding to irreducible polynomials of degree 64 are isomorphic.

A simple but inefficient way to compute the multiplicative inverse of x is to compute $x^{2^{64}-2}$ using exponentiation by squaring. In the early stages of this project, this was how the multiplicative inverse was calculated.

The extended Euclidean algorithm is more efficient, and is used in the final implementation.

Chapter 2

Finite Field Arithmetic

As previously mentioned, Reed-Solomon codes require the use of non-standard arithmetic - arithmetic over finite fields - because modular arithmetic with a non-prime modulus does not have an inverse for all elements.

Addition in GF(2^{64}) is extremely simple, as it is equivalent to XOR. Multiplication, however, is less efficient than standard multiplication, and divison even less so.

The constant POLYNOMIAL refers to the irreducible polynomial $x^{64} + x^4 + x^3 + x + 1$, with x^{64} omitted, as it would not fit in a 64-bit integer.

2.1 Russian Peasant Algorithm

The Russian peasant algorithm multiplies two values in GF(2^{64}) without requiring 128-bit integers. It incrementally performs the multiplication by adding intermediate values into an accumulator, and slowly shifting the values to be multiplied and applying polynomial reduction.

The state of the algorithm consists of the two values to be multiplied a and b, and an accumulator.

The algorithm must be executed at most 64 times, before b is guaranteed to become zero. Then, the accumulator contains the result.

At each iteration, if the low bit of b is set, the accumulator is XORed with a. Then, a is shifted left, and b is shifted right.

This is justified because, at each step, we multiply the lowest coefficient of b with a, and add the result (either 0 or a) to the accumulator. Then, moving on to the next coefficient of b, we divide b by x and multiply a by x, which is equivalent to shifting a left and b right.

If the high bit of a was set before shifting, a is XORed with the irreducible polynomial. This is because, conceptually, a now has a 65th bit (a coefficient x^{64}), which requires reduction, done by subtracting the irreducible polynomial using XOR.

Algorithm 1 Russian Peasant Multiplication

```
function MULTIPLY(a, b)
    acc ← 0
    for i ← 1 to 64 do
        if b & 1 then
            acc ← acc ⊕ a
        end if
        carry ← a & (1 ≪ 63)
        a ← a ≪ 1
        b ← b ≫ 1
        if carry then
            a ← a ⊕ POLYNOMIAL
        end if
    end for
    return acc
end function
```

This algorithm is fairly simple and easy to implement, but multiplication can be done more efficiently on modern CPUs with special instructions. Still, this algorithm is necessary as a fallback, for CPUs which don't support 128-bit carry-less multiplication.

2.2 Carry-less Multiplication

GF(2^{64}) multiplication can be performed using three 128-bit carry-less multiplication operations. Modern CPUs have support for this operation, as it is useful for cryptographic algorithms, computing checksums, and other applications. [1]

The terms "upper half" and "lower half" will be used to refer to the most significant 64 bits and least significant 64 bits of a 128-bit integer, respectively.

By multiplying a and b using carry-less multiplication, we obtain a 128-bit result. We must reduce the upper half to a 64-bit result, which can then be XORed with the lower half to obtain the final result.

This can be done by multiplying the upper half of the result by the irreducible polynomial. Then, the lower half of the result is the product reduced modulo the irreducible polynomial.

To understand why this works, consider the process of reduction. The irreducible polynomial is aligned with each set bit in the upper half of the result, and XORed with the result. This is effectively what carry-less multiplication does.

There is a complication, however. A third multiplication is required to ensure full reduction, as the highest bits of the upper half can affect the lowest bits of the upper half.

For example, consider $x^{127} + x^{67} + x^{66} + x^{64}$. After aligning the irreducible polynomial with the highest bit and XORing, all bits in the upper half are zero. At this point, the reduction is complete, but the multiplication does not know to stop here. The irreducible polynomial will also be aligned with the other three bits, and the lower half is XORed with some unnecessary values.

The upper half of the reduced result indicates if and where this happened. A third multiplication is used to correct this. The unnecessary XORs are undone by XORing with the lower half of the third multiplication.

For fields where $x^n + 1$ is irreducible, the algorithm simplifies to carry-less multiplication followed by XORing the upper and lower halves of the result. This is the case for $x^{63} + 1$, but is unfortunately not the case for $x^{64} + 1$ [5].

The justification for the algorithm may seem somewhat complex, but the algorithm itself is very short, simple, and efficient.

Algorithm 2 Carry-less Multiplication

```
function MULTIPLY(a, b)
    result ← CLMUL(a, b)
    result_partially_reduced ← CLMUL(upper(result), POLYNOMIAL)
    result_fully_reduced ← CLMUL(upper(result_partially_reduced), POLYNOMIAL)
    return lower(result) ⊕ lower(result_partially_reduced) ⊕ lower(result_fully_reduced)
end function
```

2.3 Extended Euclidean Algorithm

The polynomial extended Euclidean algorithm, given polynomials a and b, computes s and t such that $a \cdot s + b \cdot t = \mathrm{gcd}(a, b)$. When b is set to the irreducible polynomial, t is the multiplicative inverse of a. s does not need to be computed.

The algorithm uses repeated Euclidean division. Because the irreducible polynomial is of degree 64, the first Euclidean division iteration, in the first iteration of the Euclidean algorithm, is a special case. As a 65-bit polynomial cannot fit in the 64-bit variable b, the first iteration is done manually, outside the loop.

In the following pseudocode, leading_zeros(x) returns the number of leading zero bits in x. Modern CPUs have a dedicated instruction for counting leading zeros.

Algorithm 3 Extended Euclidean Algorithm

```
function EXTENDEDEUCLIDEAN(a)
    assert(a ≠ 0)
    if a = 1 then return 1 endif
    t ← 0
    new_t ← 1
    r ← POLYNOMIAL
    new_r ← a
    r ← r ⊕ (new_r ≪ (leading_zeros(new_r) + 1))
    quotient ← 1 ≪ (leading_zeros(new_r) + 1)
    while new_r ≠ 0 do
        while leading_zeros(new_r) >= leading_zeros(r) do
            degree_diff ← leading_zeros(new_r) − leading_zeros(r)
            r ← r ⊕ (new_r ≪ degree_diff)
            quotient ← quotient|(1 ≪ degree_diff)
        end while
        (r, new_r) ← (new_r, r)
        (t, new_t) ← (new_t, t ⊕ gf64_multiply(quotient, new_t))
        quotient ← 0
    end while
    return t
end function
```

Chapter 3

Novel Polynomial Basis

Standard algorithms for polynomial interpolation and evaluation, such as Newton interpolation and Horner's method, require $O(n^2)$ time. Efficient $O(n \log n)$ algorithms are used instead, based on FFT-like transforms introduced in [3].

3.1 Polynomial Basis

The polynomial basis $\mathbb{X} = \{X_0, \ldots, X_{2^{64}-1}\}$ admits transforms Ψ_h^l and $(\Psi_h^l)^{-1}$ which convert between values at h contiguous points with an arbitrary offset l and coefficients in $\mathbb{X}$, requiring h to be a power of 2. Both the forward and inverse transform can be computed in $O(n \log n)$ time.

Is it simple to oversample a polynomial using these transforms. To encode a $RS(n, k)$ code, where n and k are powers of two, the coefficients are obtained by applying the inverse transform to the input values with an offset of 0, then additional values are obtained by computing the forward transform n/k times at offsets $k, 2k, \ldots, n-k$.

The elements of the basis are obtained by multiplying the polynomials $\hat{W}_0, \ldots, \hat{W}_{63}$, corresponding to the bits set in the index i: $X_i = \prod_{j \in \text{bits}(i)} \hat{W}_j$.

$\hat{W}_i = W_i / W_i(2^i)$ is a normalized vanishing polynomial of degree 2^i, which vanishes (i.e. evaluates to zero) at the points $\omega_0, \omega_1, \ldots, \omega_{2^i-1}$, and evaluates to 1 at ω_{2^i}. W_i is the non-normalized polynomial obtained by multiplying the factors $(X - \omega_j)$ for $j = 0, \ldots, 2^i - 1$.

W_i has degree 2^i, as it is the product of 2^i degree one factors. Therefore, X_i has degree i, since it the product of W_i corresponding to the bits set in i. Since all X_i in $\mathbb{X}$ have different degrees, they are linearly independent, so they form a valid basis which can represent all 2^{64} polynomials of degree at most 63 in GF(2^{64}).

All W_i are linearized polynomials, which means that they only have coefficients at powers of 2, i.e. $W_i = c_0 + c_1X + c_2X^2 + c_3X^4 + \ldots + c_iX^{2^{i-1}}$, and they are additive: $W_i(x + y) = W_i(x) + W_i(y)$.

Note that the standard monomial basis $\{1, x, x^2, \ldots, x^{2^{64}-1}\}$ could also be defined in a similar way, with $\hat{W}_i = X^{2^i}$, but $O(n \log n)$ transforms require the more complex basis $\mathbb{X}$.

The structure of the basis allows for FFT-like recursive computation of the transforms.

3.2 Forward and Inverse Transforms

Let D_h be the data polynomial with h coefficients $d_0, d_1, \ldots, d_{h-1}$.

It D_h can be written as a recursive function $\Delta_i^m(x)$:

$$\Delta_i^m(x) = \begin{cases} \Delta_{i+1}^m(x) + \hat{W}_i(x)\Delta_{i+1}^{m+2^i}(x) & 0 \leq i \leq \log_2(h) \\ d_m & i = \log_2(h) \end{cases}$$

At each step, the polynomial is split into coefficients whose index has the i-th bit set and those which don't. The final steps select the coefficient corresponding to the selected index m.

Because of the properties of the normalized vanishing polynomials, the vector of evaluations of Δ_0^0 can be computed from two vectors of size $h/2$: the evaluations of Δ_1^0 and Δ_1^1 at even points, which can further be split into smaller vectors.

Let $\Phi(i, m, l) = [\Delta_i^m(\omega_c + \omega_l)$ for c in $[0, 2^i, \ldots, h - 2^i]]$ be the vector of $n/2^i$ evaluations of Δ_i^m at all points $\omega_c + \omega_l$ where c has the i most significant bits unset, with l an arbitrary offset.

$\Phi(i, m, l)$ can be computed in $O(n)$ time from $\Phi(i + 1, m, l)$ and $\Phi(i + 1, m + 2^i, l)$.

From each pair of values at indices i from the two smaller vectors, the values at indices $2i$ and $2i + 1$ in the larger vector can be computed. The values will be denoted as i, j, i', j' for clarity.

i' is straightforwardly computed as:

$$i' = \Delta_i^m(\omega_c + \omega_l) = \Delta_{i+1}^m(\omega_c + \omega_l) + \hat{W}_i(\omega_c + \omega_l)\Delta_{i+1}^{m+2^i}(\omega_c + \omega_l) = i + \hat{W}_i(\omega_c + \omega_l)j$$

The calculation of j' relies on the properties of the vanishing polynomials:

$$j' = \Delta_i^m(\omega_c + \omega_l + \omega_{2^i}) = \Delta_{i+1}^m(\omega_c + \omega_l + \omega_{2^i}) + \hat{W}_i(\omega_c + \omega_l + \omega_{2^i})\Delta_{i+1}^{m+2^i}(\omega_c + \omega_l + \omega_{2^i})$$

The term ω_{2^i} vanishes in both Δ_{i+1}^m and $\Delta_{i+1}^{m+2^i}$, since both contain only vanishing polynomials W_j with $j \geq i+1$.

As $\hat{W}_i$ is normalized, $\hat{W}_i(\omega_c + \omega_l + \omega_{2^i}) = \hat{W}_i(\omega_c + \omega_l) + \hat{W}_i(\omega_{2^i}) = \hat{W}_i(\omega_c + \omega_l) + 1$.

Therefore, j' is computed as:

$$j' = i + (\hat{W}_i(\omega_c + \omega_l) + 1)j = i + \hat{W}_i(\omega_c + \omega_l)j + j = i' + j$$

The reverse calculation is also straightforward, and does not require division:

$$j = j' + i' = (i' + j) + i' = j$$

$$i = i' + \hat{W}_i(\omega_c + \omega_l)j = (i + \hat{W}_i(\omega_c + \omega_l)j) + \hat{W}_i(\omega_c + \omega_l)j = i$$

The vectors can be stored interleaved in a single array, initialized to $[d_0, d_1, \ldots, d_{h-1}]$ (h single-element vectors), and then updated in-place in $log_2(h)$ steps, each step requiring $O(n)$ time.

See the butterfly diagram in [3] for a visual representation of the algorithm.

In total, $n-1$ unique factors are needed, requiring evaluations of $W_0, \ldots, W_{log2(n)-1}$. The coefficients of each W_i can be computed in $O(2^i)$ time, so the factors can be computed in $O(n \log n)$ time.

The algorithms can be implemented iteratively as follows:

Algorithm 4 Transform Algorithms

```
function PRECOMPUTEFACTORS(len, offset)
    pow ← log_2(len)
    factors ← new array of size len - 1
    factor_idx ← 0
    for step ← 0 to pow − 1 do
        groups ← 2^(pow−step−1)
        for group ← 0 to groups − 1 do
            factors[factor_idx] ← Ŵ_step(ω_(group·2^(step+1)) + ω_offset)
            factor_idx ← factor_idx + 1
        end for
    end for
    return factors
end function
function INVERSETRANSFORM(data, factors)
    pow ← log_2(len(data))
    factors_idx ← 0
    for step ← 0 to pow − 1 do
        group_len ← 2^(step+1)
        for group ← 0 to 2^(pow−step−1) − 1 do
            for x ← 0 to group_len/2 − 1 do
                i ← group · group_len + x
                j ← i + group_len/2
                data[j] ← data[j] + data[i]
                data[i] ← data[i] + data[j] · factors[factors_idx]
            end for
        end for
        factors_idx ← factors_idx + 1
    end for
end function
function FORWARDTRANSFORM(data, factors)
    pow ← log_2(len(data))
    factors_idx ← len(factors) - 1
    for step ← 0 to pow − 1 do
        group_len ← 2^(step+1)
        for group ← 0 to 2^(pow−step−1) − 1 do
            for x ← 0 to group_len/2 − 1 do
                i ← group · group_len + x
                j ← i + group_len/2
                data[i] ← data[i] + factors[factors_idx] · data[j]
                data[j] ← data[j] + data[i]
            end for
        end for
        factors_idx ← factors_idx − 1
    end for
end function
```

Chapter 4

File Format

Currently, parity data is stored in a separate file, specified by the user. Support for multiple input files, and single-file archives is planned, but not yet implemented.

The parity file contains necessary metadata and hashes for error detection.

4.1 Metadata

The parity file header contains the expected size of the data file, the number of data and parity blocks, and the size of a block. It also contains a hash of the file metadata (excluding the hash itself), used to detect metadata corruption.

It is necessary to store the size of the data file, even though the block size and number of data blocks are also stored, because the last block is allowed to be incomplete, and is implicitly padded with zeros for the Reed-Solomon encoding. The size of the data file cannot be inferred from the block size and number of blocks.

Currently, metadata repair is not supported. It could be implemented by creating meta-parity blocks and interleaving them with the normal parity blocks. These blocks would require a header string and hash embedded in them, to allow locating them to recover the metadata.

After the file header and metadata hash, the hashes and first 8 bytes of each block are stored.

The purpose of the 8-byte prefixes is to allow reassembly of the data and parity files if somehow the blocks become scrambled. This should not happen as a result of normal corruption, which would edit bytes but not insert or delete, but it could theoretically happen as a result of a bug in some network transfer or filesystem operation. While it's extremely unlikely such a thing would happen, it costs very little space to include the prefixes, and without them, deletion or insertion of a single byte would completely defeat the error correction scheme.

Such errors can be simulated by inserting or deleting characters in Notepad or a hex editor, and by cutting and pasting large sections of the file around. Reassembly should have no issue recovering from these errors, with only a few blocks (the ones cut in half) being lost.

Note that without metadata repair, any errors that hit the metadata will still be fatal, but the metadata should be a small part of the file.

4.2 Blocks

The input file is split into data blocks, and the generated parity file contains parity blocks. Blocks are not, as might be expected, individual Reed-Solomon codes. If they were, damage to a block could not repaired using other blocks, as they would be completely independent.

Let b be the number of blocks, and n the number of 64-bit symbols in a block.

It might be expected that there are b Reed-Solomon codes, each with n symbols, but it is instead the opposite.

There are n Reed-Solomon codes, each with b symbols. A code is made up of all symbols at a given index in each block. If a block is lost, this results in losing one symbol from each code.

This scheme is necessary for several reasons:

- Due to the $\mathcal{O}(n^2)$ time complexity of the encoding and repair algorithms, attempting to treat a file as one big code would be far too slow.
- The interpolation algorithm would produce a polynomial of the size as the file. A terabyte file would produce a terabyte polynomial, which would need to be kept on disk until all parity blocks are generated.
- Repair would always require processing the entire file, even if only a single block is lost.

The downside is that codes are not contiguous on disk, requiring reading and writing to many different locations. Naively processing one code at a time would require one system call per symbol. To mitigate this, many codes are read at once, depending on the available memory, and processed in parallel on all available cores.

Chapter 5

Technology

The project is implemented with Rust, and uses external libraries for OS interaction, multi-threading, progress reporting, and `blake3` hashing.

5.1 MultiThreaded Encoding and Decoding

The same core code is used for encoding and repairing, as the same fundamental interpolation and evaluation process is used in both cases.

When generating parity data, symbols are read strictly from the data file and written strictly to the parity file. When repairing data, in general, symbols are read from both files, and written to the damaged files, which could be either or both of the data and parity files.

The core code is given the indices of good and corrupt blocks in both files, the input and output x values to use for encoding, handles of the input and output files, and memory maps of the same files.

In the case of encoding, it is told to use all blocks in the data file, and consider every parity block corrupt (as none exist yet). For repair, the verification code is used to determine which blocks are corrupt using the hashes.

Reading is done using the `positioned-io` library, which allows convenient random access to files. Attempting to use memory maps for reading seemed to cause the file to be read into memory and remain there, with old pages not being removed from memory.

For writing, the memory maps, created with the `memmap2` library, are used instead.

Communication between threads is primarily done using `crossbeam-channel`, a library which provides multi-producer, multi-consumer channels.

The multithreaded pipeline consists of a reader thread, an adapter thread, many processor threads, and a writer thread.

Five channels are used for the following purposes:

- Sending filled input buffers from the reader to the adapter.
- Sending filled input buffers, with some additional data and reference counting added, from the adapter to the processors.
- Sending filled output buffers from the processors to the writer.
- Returning input buffers to the reader after they have been processed by the processors.
- Returning output buffers to the processors after their data has been written by the writer.

Since multiple codes are read into an input buffer at once, reference counting and read-write locks are used to manage the sharing of input buffers between multiple processor threads. To return input buffers to the reader, an atomic integer is bundled with the buffer, and decremented by a processor when it finished interpolating a code from the buffer, so that the last processor to work on a buffer knows to return it to the reader.

The adapter thread is responsible for sending many references to the same input buffer to the processors, wrapped in a structure that includes information about which code from the buffer to process, and a reference to the atomic integer used to count the number of tasks remaining for the buffer.

Unlike input buffers, output buffers contain a single code. The writer relies on the operating system memory mapping system to efficiently write the data to disk, coalescing writes when possible. While it would be possible to gather output buffers into larger buffers and use `positioned-io` for writing as well as reading, the operating system appears to handle the write-only maps efficiently, and using memory maps for writing was simpler to implement.

The amount of memory and number of threads used is automatically determined, using the libraries `num_cpus` and `sysinfo` to query the OS for the available resources.

5.2 User Interface

The program has a basic CLI interface, implemented without any libraries. It supports the following commands:

- `encode` - Generates parity data.
- `verify` - Checks for corruption in the data and parity files. Code shared with the repair command for finding corrupt block locations.
- `repair` - Repairs corruption in the data or parity files, if there is enough redundancy.
- `reassemble` - Attempts to find misplaced blocks in the data and parity files, and copies them to new files in the correct locations.
- `test` - Runs an end-to-end test of the encoding, verification, and repair commands. (default)

Progress reporting is done using the terminal progress bar library `indicatif`.

5.3 Testing

The finite field and polynomial arithmetic code is testing with random data (generated using `fastrand`) using Rust's built-in testing framework.

The encoding, verification, and repair code is tested using the aforementioned end-to-end test, which randomly corrupts a test file and attempts to repair it.

Automated testing has not yet been implemented for the `reassemble` command. Manual testing was successful.

Chapter 6

Conclusions

6.1 Summary

The implementation can successfully generate parity data and repair file corruption using Reed-Solomon codes.

The file metadata is currently not protected from corruption. This can be addressed by adding meta-parity blocks among the parity blocks. Although the metadata is generally small, this is a significant flaw in the current implementation.

Only a single data file and parity file are supported. Multiple input files and single-file archives would be a useful feature to add. This would require significal re-architecting of the program's I/O in order to read and write blocks to arbitrary files in arbitrary folder structures. The file header of the parity file would need to be extended to include relative paths to the data files. Single-file archives would be less difficult to add.

6.2 Figures

As a final note, and to visually demonstrate some limitations of this error correction scheme, the following figures show the use of Reed-Solomon codes to repair bitmap images.

Figure B is impossible to repair, yet figure A has more errors. This is because the errors in figure A are contiguous - they are burst errors - so they affect less blocks.

Figure C is the recovered image from figure A.

See the script `generate_figures.py` for how these images were generated.

(a) A bitmap image with repairable errors.

(b) A bitmap image with irreparable errors.

(c) Figure A, repaired.

Figure 6.1: Image source: `scipy.datasets.face`.

Bibliography

[1] Shay Gueron and Michael E. Kounavis. *Intel® Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode*. 2014. URL: https://web.archive.org/web/20190806061845/https://software.intel.com/sites/default/files/managed/72/cc/clmul-wp-rev-2.02-2014-04-20.pdf (visited on 01/31/2025).

[2] Rudolf Lidl and Harald Niederreiter. *Finite Fields*. 1997. URL: https://www.cambridge.org/core/books/finite-fields/75BDAA74ABAE713196E718392B9E5E72 (visited on 01/31/2025).

[3] Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang S. Han. "Novel Polynomial Basis and Its Application to Reed-Solomon Erasure Codes". In: *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*. 2014, pp. 316–325. DOI: 10.1109/FOCS.2014.41. URL: https://arxiv.org/pdf/1404.3458 (visited on 05/28/2025).

[4] F. Jessie MacWilliams and N. J. A. Sloane. "The Theory of Error-Correcting Codes". In: 1977. URL: https://api.semanticscholar.org/CorpusID:118260868 (visited on 01/31/2025).

[5] Gadiel Seroussi. *Table of Low-Weght Binary Irreducible Polynomials*. 1998. URL: https://shiftleft.com/mirrors/www.hpl.hp.com/techreports/98/HPL-98-135.pdf (visited on 01/31/2025).