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

This project aims to implement file corruption repair using Reed-Solomon error-correcting codes. The code used is an erasure code over the field GF(2^{64}), implemented using Newton interpolation.

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

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

The file is split into blocks, and an arbitrary number of parity blocks can be generated. Repair requires as many intact blocks as there were data blocks originally. Block corruption is detected using a hash.

The program processes multiple codes at once. This is necessary, as each code is spread across the file, which would cause a naive implementation to perform a system call for each symbol.

Multithreading is used to speed up the polynomial interpolation and evaluation.

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 Newton interpolation.

The fundamental principle of the code is to interpret the data as 64-bit values of a polynomial, use interpolation to obtain the coefficients of the polynomial, and then evaluate the polynomial at different x-values to obtain the encoded data.

The redundancy of the code comes from the fact that there is only one polynomial of degree at most $k - 1$ that passes through k points. Any combination of at least k the original and redundant values can be interpolated once again to obtain the same polynomial, and then evaluate it at the x-values of corrupted data to recover it. Any amount of redundancy can be added, up to the limit of 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. Other codes do not include the original data, and 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]

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: $a \cdot (b + c) = a \cdot b + a \cdot c$

Although the real numbers form a field, standard machine arithmetic as supported by most CPUs does not form a field. This is because multiplication is not always invertible. For example $0 \cdot 2 = 0$, but also $2^{n-1} \cdot 2 = 0$ (because of overflow), where n is the number of bits in the integer.

The theorems of the polynomial mathematics used in Reed-Solomon codes require a field, so standard arithmetic cannot be used.

Fortunately, it is possible to define fields over integers of a fixed size, although the arithmetic is more complex. The size of the field must be either a prime number or a power of a prime number. In the case of machine arithmetic, the number of integers is a power of 2.

In a finite fields GF(p^m), where p is a prime number and m is a positive integer, the elements are themselves polynomials of degree $m - 1$, with coefficients in GF(p). So, 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)). For the purposes of this project, n is always 64, so the field is GF(2^{64}).

It is important to note that these polynomials are not the same as the ones used in Reed-Solomon codes. Finite field polynomials are simply machine integers with more complex arithmetic. They are polynomials over GF(2), with 64 coefficients. Reed-Solomon polynomials can be arbitrarily long. They are polynomials over GF(2^{64}), with an arbitrary number of coefficients.

Addition is defined as polymial addition. As the coefficients are bits, this is equivalent to XOR. Additive inverse is a no-op, because XOR is its own inverse.

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

The choice of irreducible polynomial is arbitrary. There are public tables of irreducible polynomials available online, and the choice of polynomial does not affect correctness, as long as the encoder and decoder use the same polynomial. The chosen polynomial for this project is $x^{64} + x^4 + x^3 + x + 1$.

Division is defined as multiplication by the multiplicative inverse.

A simple way to compute the inverse is to raise the element to the power of $2^{64} - 2$ 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

The constant POLYNOMIAL referred to in pseudocode is the irreducible polynomial $x^{64} + x^4 + x^3 + x + 1$, with the coefficient x^{64} omitted, as it does not fit in a 64-bit integer. [5]

As previously mentioned, Reed-Solomon codes require the use of non-standard arithmetic - arithmetic over finite fields.

In modular arithmetic, a modular inverse exists only for elements that are coprime with the modulus, i.e. only multiplication by odd numbers is invertible modulo 2^n. The field axioms require that all elements have a multiplicative inverse.

Using GF(2^{64}) arithmetic is necessary to resolve this issue, but complicates the arithmetic. Addition is extremely simple, as it is equivalent to XOR. Multiplication, however, is less efficient than standard multiplication, and divison even less so.

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. Still, this algorithm is necessary as a fallback, for CPUs which don't support 128-bit carry-less multiplication.

2.2 Multiplication - Carry-less Multiplication

GF(2^{64}) multiplication can be performed using only 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 fields where $x^{n-1} + 1$ is irreducible, two multiplications would suffice, but this is not the case for GF(2^{64}).

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 and should stop. However, this is not how carry-less multiplication works. 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 if and where this happened. A third multiplication will correct this. The unnecessary XORs are undone by XORing with the lower half of the third multiplication.

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

The functions upper(x) and lower(x) return the upper and lower 64 bits of the 128-bit integer x, respectively.

Algorithm 2 Carry-less Multiplication

function MULTIPLY(a, b)
    result $\leftarrow$ CLMUL(a, b)
    result_partially_reduced $\leftarrow$ CLMUL(upper(result), POLYNOMIAL)
    result_fully_reduced $\leftarrow$ CLMUL(upper(result_partially_reduced), POLYNOMIAL)
    return lower(result) $\oplus$ lower(result_partially_reduced) $\oplus$ 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 $a \cdot s + b \cdot t = \gcd(a, b)$. 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. On modern CPUs, this function can be implemented using the `LZCNT` instruction.

Algorithm 3 Extended Euclidean Algorithm

function EXTENDEDEUCLIDEAN(a)
  assert($a \neq 0$)
  if $a = 1$ **then return** 1 **endif**
  $t \leftarrow 0$
  new_t $\leftarrow 1$
  $r \leftarrow$ POLYNOMIAL
  new_r $\leftarrow a$
  $r \leftarrow r \oplus$ (new_r $\ll$ (leading_zeros(new_r) $+ 1$))
  quotient $\leftarrow 1 \ll$ (leading_zeros(new_r) $+ 1$)
  while new_r $\neq 0$ **do**
    while leading_zeros(new_r) $>=$ leading_zeros(r) **do**
      degree_diff $\leftarrow$ leading_zeros(new_r) $-$ leading_zeros(r)
      r $\leftarrow r \oplus$ (new_r $\ll$ degree_diff)
      quotient $\leftarrow$ quotient$|(1 \ll$ degree_diff)
    end while
    $(r,$ new_r$) \leftarrow$ (new_r, r)
    $(t,$ new_t$) \leftarrow$ (new_t, $t \oplus$ gf64_multiply(quotient, new_t))
    $quotient \leftarrow 0$
  end while
  return t
end function

Chapter 3

Polynomial Interpolation and Evaluation

3.1 Horner's Rule

Horner's rule is a method for evaluating polynomials in $\mathcal{O}(n)$ time.

A polynomial $a_0 + a_1x + \ldots + a_{n-1}x^{n-1}$ can be written as $a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-1})\ldots))$.

It can then be evaluated from the inside out, starting with the highest degree coefficient a_{n-1}, and repeatedly multiplying by x and adding the next coefficient.

3.2 Newton Interpolation

Polynomial interpolation is used to find a polynomial of degree at most $k-1$ that passes through k points, where the x coordinates of the points are distinct. It is known that this polynomial is unique.

Newton interpolation is the polynomial interpolation algorithm used in this project to interpolate data in $\mathcal{O}(n^2)$ time and $\mathcal{O}(n)$ space, where n is the number of points.

The Newton interpolation polynomial is a linear combination of basis polynomials.

$$N(x) = \sum_{k=0}^{n} [y_0, \ldots, y_k] \cdot \prod_{i=0}^{k-1}(x - x_i) \tag{3.1}$$

The notation $[y_k, \ldots, y_{k+j}]$ denotes the divided difference, which is recursively defined as

follows:

$$[y_k] = y_k \tag{3.2}$$

$$[y_k, \ldots, y_{k+j}] = \frac{[y_{k+1}, \ldots, y_{k+j}] - [y_k, \ldots, y_{k+j-1}]}{x_{k+j} - x_k} \tag{3.3}$$

Naive computation of divided differences requires $\mathcal{O}(n^2)$ time and space, but the space requirement can be reduced to $\mathcal{O}(n)$ by reusing values from previous iterations.

At the initial iteration, the basis polynomial is initialized to $n_0(x) = 1$ and the list of divided differences is initialized to $[y_0]$.

At any iteration $k + 1$, divided differences $[y_0, \ldots, y_k], [y_1, \ldots, y_k], \ldots, [y_{k-1}, y_k], [y_k]$ are known from previous iterations.

The value $[y_{k+1}]$ is added to the list of divided differences, and the rest of the values are updated to $[y_0, \ldots, y_{k+1}], [y_1, \ldots, y_{k+1}], \ldots, [y_{k-1}, y_{k+1}], [y_k, y_{k+1}]$, from right to left. At every step of the update, the next divided difference is computed using the current value and the value to its right (which was updated in the previous step).

The basis polynomial is n_{k+1} is computed by multiplying the previous basis polynomial by $x - x_k$.

The interpolation polynomial is updated by adding the basis polynomial multiplied by the divided difference to it.

At each iteration, we perform multiplication by $x - x_k$, update the divided differences, and update the interpolation polynomial. Since these steps require $\mathcal{O}(k)$ time, and we perform n iterations, the total time complexity is $\mathcal{O}(n^2)$.

No general polynomial multiplication is required, nor is any polynomial division (only GF(2^{64}) division in the divided differences).

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, multithreading, 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 $\mathcal{O}(n^2)$ Newton interpolation algorithm is a significant issue for large files, as it prevents the use of small blocks for more granular error correction. Implementing a $\mathcal{O}(n \log n)$ algorithm is one main area for future improvement. [3]

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.

[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).