QuadIron: A library for Number Theoretic Transform-based Erasure Codes*

Vianney Rancurel†
vianney.rancurel@scality.com

Lâm Pham-Sy‡
lam.pham-sy@scality.com

Sylvain Laperche‡
sylvain.laperche@scality.com

April 13, 2020

Abstract

Reed-Solomon codes can be seen as polynomial operations which can be implemented by number theoretic transforms and accelerated by different fast Fourier transform algorithms according to the characteristics of the fields. We present a library, QuadIron, which implements FFTs in binary, prime, and extension fields, as well as a set of optimizations to make them practical for a very large number of symbols. These FFT-based erasure codes could be useful for applications like decentralized storage over Internet, or real-time video high-quality streams.

Keywords— Number Theoretic Transforms, Discrete Fourier Transforms, Finite Fields, Erasure Codes, Digital Storage, Distributed storage, Decentralized Storage, Decentralized Cloud Storage, Real-time video, high-quality streaming

1 Introduction

A $\mathcal{C}(n, k)$ erasure code is defined by $n = k + m$, k being the number of data fragments, m being the number of desired erasure fragments. In an application it is required to transmit the n fragments. A Maximum Distance Separable (MDS) code guarantees that any k fragments can be used to decode a file. Erasure codes can be either systematic or non-systematic. Systematic codes generate $n - k$ erasure fragments and therefore maintain k data fragments. Non-systematic codes generate n erasure fragments. In the case of systematic codes, we try to retrieve primarily the k data fragments if possible because there is nothing to decode. A decoding is necessary only if one or more data fragments are missing. In the case of non-systematic codes, we need to decode k fragments. Erasure codes can also be compared by their sensitivity to the rate $r = k/n$, which may or may not impact the encoding and decoding speed. Another comparison criterion is the support of adaptive rate: does the erasure code allows to change k and m dynamically, without having to regenerate the whole set of erasure fragments. Another critical property is called the 'confidentiality' [1] which is determined if an attacker can partially decode the data if he obtains less than k fragments. Finally, we can also compare erasure code according to their repair bandwidth, i.e. the number of fragments required to repair a fragment. To sum up, here is a list of codes' properties that are of interest for us:

- MDS/non-MDS
- Systematic/Non-systematic
- Encoding/Decoding speed according to various n
- Encoding/Decoding speed predictivity and stability acc/ to n

*Manuscript received Januray 12, 2020.
†Scality in San Francisco, CA, 94104.
‡Scality France.

- Rate sensitivity
- Adaptive vs non-adaptive rate
- Confidentiality
- Repair bandwidth

Reed-Solomon (RS) codes are MDS codes constructed from Vandermonde or Cauchy matrices [2] that are systematic and support adaptive rates. The RS encoding process is traditionally performed by multiplying a $[n \times k]$ matrix by a k data vector in $\mathbb{F}_{2^w}$, leading to a high complexity of $\mathcal{O}(k \times n)$. The topic of optimizing those codes has been widely discussed. While such optimizations reduce the overall complexity, most of them rely solely on hardware optimizations, therefore their real complexity in "big O" notation has not been widely studied.

Low-Density-Parity-Check (LDPC) codes are also an important class of erasure codes and are constructed over sparse parity-check matrices. Although initially used in networking applications, some researchers recently showed that it is possible to use them in distributed storage scenarios [3]. Those codes, which even though require to store $n = k + m$ fragments (like MDS codes), need to retrieve $[f \times k]$ fragments to recover the data (instead of only k for MDS codes), f being called the overhead or the inefficiency. The study [3] shows 3 types of LDPC codes where f oscillates between 1.10 and 1.30 for $n \lessapprox 100$, and shows that you need a larger n, e.g. $n \gtrapprox 1000$ in order to have an f approaching 1.0. As shown by [3], those codes are more sensible to network latency because of the extra fragments, due to the overhead, that they need to retrieve, so in cases where latency can be important RS codes seems more interesting than LDPC codes. More recently hybrid-LDPC schemes such as [4] have reduced the overhead to $[k + f]$ with a very small f. Also, [5] have shown that it is possible to design LDPC codes which beat RS codes when taking into account the repair bandwidth, because RS codes always need to retrieve k fragments to be able to repair the data, while it is possible to design LDPC codes that require less than k fragments for the repair process. However:

- LDPC are not MDS: it is always possible to find a pattern (e.g. stopping sets) that cannot decode (e.g. having only k fragments out of n).
- You can always find/design an LDPC code optimized for few properties (i.e. tailored for a specific use case) that beats other codes on those few properties, but there is no silver bullet: it will be sub-optimal for the other properties (its a trade-off, e.g. good for large n and with an optimal repair bandwidth, but not good for small n and cannot support adaptive rate): these cannot be used in a generic library.
- Designing a good LDPC code is some kind of black art that requires a lot of fine tuning and experimentation. Ultimately an LDPC code optimal for all the interesting properties for a given use case could exist but would be very complex and/or would only be available in a commercial library.

Recently some other types of codes, called Locally-Repairable-Codes (LRC) [6] [7], have tackled the repair bandwidth issue of the RS codes. They combine multiple layers of RS: the local codes and the global codes. However those codes are not MDS and they require an higher storage overhead than MDS codes.

Fast Fourier transform (FFT) based RS codes remain relatively simple, and can be used to perform encoding on finite fields with clearer and lower announced complexities therefore having a good set of desirable properties:

- They are MDS
- Support both systematic or non-systematic codes
- Fast for a wide range of n
- Rate insensitive (for non-systematic codes).
- Confidential (for systematic codes).

We focus our research on optimizing their encoding and decoding speed.

Since the computational complexities of FFT operations mainly depend on the chosen finite field, we investigate two types of finite fields: prime finite fields and binary extension finite fields. For each type, there are different approaches to accelerate the FFT operations.

2 Reed-Solomon codes

A Reed-Solomon (RS) code is an MDS code $\mathcal{C}(n,k)$ defined over a finite field $\mathbb{F}_q$ where q is a prime power $q = p^w$. A code word composed of n symbols is generated from a message of length k. The message is recovered from a subset of at least k elements of the code word.

2.1 Encoding

As Reed and Solomon described in their foundational paper[2], a code word is encoded from a message using polynomial evaluation.

A message $\mathbf{m} = (m_0, \ldots, m_k) \in \mathbb{F}_q^k$ can be represented as a polynomial $p_{\mathbf{m}}(x)$ of degree $k-1$:

$$p_{\mathbf{m}}(x) = m_0 + m_1 x + m_2 x^2 + \cdots + m_{k-1} x^{k-1} \tag{1}$$

The code word $\mathbf{c} = (c_0, \ldots, c_{n-1}) \in \mathbb{F}_q^n$ of the message $\mathbf{m}$ is obtained by evaluating $p_{\mathbf{m}}(x)$ at a given but arbitrary set of n *different* points $S = \{s_0, \ldots, s_{n-1}\}$ of the field $\mathbb{F}_q$, called the set of evaluation points. Concretely,

$$\begin{aligned} c_0 &= m_0 + m_1 s_0 + m_2 s_0^2 + \cdots + m_{k-1} s_0^{k-1}, \\ c_1 &= m_0 + m_1 s_1 + m_2 s_1^2 + \cdots + m_{k-1} s_1^{k-1}, \\ &\cdots \\ c_{n-1} &= m_0 + m_1 s_{n-1} + m_2 s_{n-1}^2 + \cdots + m_{k-1} s_{n-1}^{k-1} \end{aligned} \tag{2}$$

Equation (2) can be represented as a multiplication of the message $\mathbf{m}$ and matrix G:

$$\mathbf{c} = \mathbf{m} \times G \tag{3}$$

where G is the generator matrix of the RS codes:

$$G = \begin{bmatrix} 1 & 1 & \ldots & 1 \\ s_0 & s_1 & \ldots & s_{n-1} \\ s_0^2 & s_1^2 & \ldots & s_{n-1}^2 \\ \vdots & \vdots & & \vdots \\ s_0^{k-1} & s_1^{k-1} & \ldots & s_{n-1}^{k-1} \end{bmatrix} \tag{4}$$

Since $s_i \neq s_j$ when $i \neq j$, G is actually a Vandermonde matrix [8].

Straightforwardly, the encoding process has a computational complexity of $\mathcal{O}(k \times n)$ for performing a multiplication of a vector $\mathbf{m}$ and a matrix G.

This complexity can be reduced by using a Discrete Fourier Transform (DFT) on a finite field that evaluates polynomial $p_{\mathbf{m}}(x)$ at the set S of evaluating points. Currently, two main Fast Fourier Transform (FFT) techniques apply for prime field $\mathbb{F}_p$ [9, 10, 11], and for binary extension field $\mathbb{F}_{2^w}$ [12, 13, 14, 15]. We summarize these techniques in Section 3. Briefly, thanks to the FFT techniques, the encoding complexity drops to $\mathcal{O}(n \times \log n)$.

2.2 Decoding

A message $\mathbf{m}$ can be decoded from a code word $\mathbf{c}$ that could miss at most $n-k$ elements. Indeed, given a set of k coefficients of $\mathbf{c}$, the polynomial $p_{\mathbf{m}}(x)$ in Equation (1) can be regenerated using Lagrange interpolation because its degree is at most $k-1$. Let $\{\tilde{c}_0, \ldots, \tilde{c}_{k-1}\}$ denote the set of k known coefficients corresponding to k evaluation points $\{\tilde{s}_0, \ldots, \tilde{s}_{k-1}\}$. The polynomial $p_{\mathbf{m}}(x)$ is thus interpolated:

$$p_{\mathbf{m}}(x) = \sum_{i=0}^{k-1} \Big(\tilde{c}_i \times \sum_{0 \leq j \leq k-1, j \neq i} \frac{x - \tilde{s}_j}{\tilde{s}_i - \tilde{s}_j} \Big) \tag{5}$$

Coefficients of the message $\mathbf{m}$ are actually coefficients of $p_{\mathbf{m}}(x)$.

On prime field, equation 5 can be simplified as proposed in [10]. Note that degree of $p_{\mathbf{m}}(x)$ is at most $(k-1)$, the polynomial is derived as:

$$p_{\mathbf{m}}(x) = \Big(A(x) \times \sum_{i=0}^{k-1} \frac{n_i}{x - \tilde{s}_i}\Big) \bmod x^k \tag{6}$$

where

$$A(x) = \prod_{i=0}^{k-1}(x - \tilde{s}_i),$$
$$n_i = \frac{\tilde{c}_i}{A'(\tilde{s}_i)}$$

Using Taylor series, we have

$$\frac{1}{\tilde{s}_i - x} = \sum_j \frac{x^j}{\tilde{s}_i^{j+1}}$$

And by defining the following polynomial:

$$N(x) := \sum_{i=0}^{k-1} \frac{n_i}{\tilde{s}_i} \times x^{z_i}$$

where its exponents $\{z_i, i = 0, \ldots, k-1\}$ are determined by the evaluation set S and the n^{th}-root of unity r

$$\tilde{s}_i = r^{z_i},\ i = 0, \ldots, k-1$$

We obtain a simpler formula:

$$p_{\mathbf{m}}(x) = \Big(A(x) \times B(x)\Big) \bmod x^k \tag{7}$$

where $B(x) := \sum_{i=0}^{k-1} N(r^{-i})x^i$ whose coefficients are obtained by evaluating $N(x)$ at k points $\{r^{-i}\}$ that can be efficiently performed using the inverse FFT algorithm.

Thanks to the convolution theorem, we can use FFT techniques to accelerate the multiplication of two polynomials $A(x), B(x)$:

$$A(x) \times B(x) = iFFT_{2k}(FFT_{2k}(A) \cdot FFT_{2k}(B)) \tag{8}$$

where $FFT_{2k}, iFFT_{2k}$ are FFT and inverse FFT of length $2 \times k$.

Note that $A(x)$ depends only on evaluation points, hence $A(x), A'(x), FFT_{2k}(A)$ can be calculated once.

The decoding computation is composed of: (1) one inverse FFT of length n to compute $N(x)$; (2) one FFT and one inverse FFT of length $2k$ to compute $A(x) \times B(x)$. It leads to a decoding complexity of $\mathcal{O}(n \log n + 4k \log 2k)$.

Compared to [10], our computations are based on the fact that the degree of $p_{\mathbf{m}}(x)$ is smaller than k. Hence we use $2k$-length FFT instead of $2n$-length FFT.

3 How can FFT reduce Encoding Complexity?

In this section, we summarize two FFT techniques on the field $\mathbb{F}_q$ that can the reduce computational complexity of the encoding process. Their common methodology is to split an FFT operation on a vector $\mathbf{m}$ into two FFT operations on a smaller set $\mathbf{m}_1$ and $\mathbf{m}_2$. Let N denote the length of input vector $\mathbf{m}$ that is also represented as a polynomial $p_{\mathbf{m}}(x)$

$$p_{\mathbf{m}}(x) = m_0 + m_1 x + m_2 x^2 + \cdots + m_{N-1} x^{N-1} \tag{9}$$

Given a set $S = \{s_0, \ldots, s_{N-1}\}$ of different elements of $\mathbb{F}_q$, FFT transforms the input vector to an output vector $\mathbf{c}$ whose coefficients are values of $p_{\mathbf{m}}(x)$ evaluated at points of S.

$$\mathbf{c} = \big(p_{\mathbf{m}}(s_0), p_{\mathbf{m}}(s_1), \ldots, p_{\mathbf{m}}(s_{N-1})\big) \tag{10}$$

This is similar to the described RS encoding process, except that the input and output vectors are of the same length N.

The key factor that allows splitting FFTs into smaller transformations is choosing the set S. The two methods of constructing such a set are described here.

3.1 S as multiplicative group

The set S is constructed as a multiplicative group whose generator is the N^{th} root of unity of the field $\mathbb{F}_q$, i.e.

$$S = \left\{\alpha^0, \ldots, \alpha^{N-1}\right\} \tag{11}$$

where α is the N^{th} root of unity of the field $\mathbb{F}_q$. We call this technique *multiplicative FFT*.

The FFT technique applied for the constructed set S was first introduced in [9]. Supposing that $N = N_1 \times N_2$, the FFT operation on $\mathbf{m}$ is split into two FFT operations on two vectors $\mathbf{m}_1$ and $\mathbf{m}_2$ of length N_1 and N_2 respectively. This results in a computational complexity $\mathcal{O}(N(N_1+N_2))$. If N is highly composite, this technique reduces the computation time to $\mathcal{O}(N \log N)$.

To obtain such advantages, however, this technique requires the satisfaction of two conditions:

1. N is a divisor of $q-1$ (for the existing N^{th} root of unity of the field $\mathbb{F}_q$),
2. N is highly composite, e.g. $N = 2^v$ ideally

A simple solution to satisfying these conditions is to use the field $\mathbb{F}_{q=p^w}$ where p is prime and $p-1$ is highly composite. Fermat numbers, i.e. $F_i = 2^{2^i} + 1$ for $i \leq 4$, are perfectly suitable for that. This technique, also called Fermat Number Transform (FNT) based erasure codes, was introduced and analyzed in [10, 11].

This technique, however, does not efficiently apply to the binary extension field $\mathbb{F}_{2^w}$ because divisors of $(2^w - 1)$ are not highly composite. However, $\mathbb{F}_{2^w}$ is perfectly suitable for most practical applications because each element can be expressed by n bits. The following FFT technique focuses on this field.

3.2 S as additive group

This technique focuses on the binary extension field $\mathbb{F}_{2^w}$. Introduced in[12, 13], it was recently improved in [14] by using Taylor extension algorithms. A further improvement is shown in [15].

The FFT length is necessarily a power of 2, i.e. $N = 2^m$ with $m \leq w$. Let $\beta_0, \ldots, \beta_{m-1}$ be m linearly independent elements of $\mathbb{F}_{2^n}$. The set S is chosen as a subspace spanned by β_i over $\mathbb{F}_2$, i.e.

$$\begin{aligned} s_i = i_0\beta_0 + i_1\beta_1 + i_2\beta_2 + \cdots + i_{m-1}\beta_{m-1}, \\ \text{for } 0 \leq i \leq 2^m - 1 \end{aligned} \tag{12}$$

where $i = i_0 + i_1 2 + i_2 2^2 + \cdots + i_{m-1}2^{m-1}$ with $i_j \in \mathbb{F}_2$. The FFT operation on a vector $\mathbf{m}$ is re-expressed as two FFTs on two vectors of half length.

3.3 Computational complexity

Let $\mathcal{A}(N)$ denote the number of additions or subtractions in the field $\mathbb{F}_q$ needed for an FFT operation with an input vector of length N (assuming that an addition requires the same amount of effort to implement as a subtraction).

Let $\mathcal{M}(N)$ denote the number of multiplications required in the field $\mathbb{F}_q$ needed for an FFT operation with an input vector of length N.

The computational complexities of these algorithms are shown in Table 1, which shows that, with suitable parametrization of the FFT length and the field $\mathbb{F}_q$, multiplicative FFT is faster than all available additive FFTs.

Table 1: Computational complexity of FFT techniques

Name	Restriction	Addition $\mathcal{A}(N)$	Multiplication $\mathcal{M}(N)$
Multiplicative FFT	$N = 2^v$ and $(q-1)\%N = 0$	$N \log_2(N)$	$\frac{1}{2}N \log_2(N) - N + 1$
Additive FFT [14]	$N = 2^v$ and $q = 2^w$	$4N(\log_2(N))^2 + \frac{3}{4}N \log_2(N) - \frac{1}{2}N$	$2N \log_2(N) - 2N + 1$
Additive FFT [14]	$N = 2^{2^v}$ and $q = 2^w$	$N \log_2(N) + \frac{1}{2}N \log_2(N) \log_2 \log_2(N)$	$\frac{1}{2}N \log_2(N)$
Additive FFT [15]	$N = 2^v$ and $q = 2^w$	$N \log_2(N)$	$\frac{1}{2}N \log_2(N)$

4 The QuadIron library

The QuadIron library is a C++ library, written in C++14, that provides a streaming API to use the different flavors of NTT-based erasure codes.

The library focuses primarily on high fragmentation, i.e. $n \gg k$.

It includes general modular arithmetic routines, algorithms for manipulating rings of integers modulo n, finite fields (including binary, prime, and non-binary extension fields), polynomial operations, different flavors of discrete Fourier transforms, and forward error correction (FEC) algorithms.

The library also includes an abstraction for writing systematic and non-systematic codes, although for applications requiring high fragmentation, systematic codes are not especially useful.

4.1 Optimizations

The QuadIron library's code relies heavily on templates to generate specifically optimized code at compile time. We use this template to allow a selection of numeric types matching the size of the processor's registers to optimize performance.

4.2 Vectorization

We focus here on vectorizing multiplicative FFTs which are based on prime (non-binary) field $\mathbb{F}_q$ operations.

We observe that there are three operations that are costly in multiplicative FFT transformation: addition, subtraction and Hadamard multiplication. All of them are element-wise operations where the inputs are two vectors $\vec{a}, \vec{b}$ and the output is a vector $\vec{c}$. All of these vectors are of the same length n.

$$\begin{aligned} \vec{a} &= (a_1, a_2, \ldots, a_n) \\ \vec{b} &= (b_1, b_2, \ldots, b_n) \\ \vec{c} &= (c_1, c_2, \ldots, c_n) \end{aligned} \tag{13}$$

- Addition
$$c_i = (a_i + b_i)\%q, \text{ for } i = 1, \ldots, n$$
- Subtraction
$$c_i = (a_i - b_i)\%q, \text{ for } i = 1, \ldots, n$$
- Hadamard multiplication
$$c_i = (a_i * b_i)\%q, \text{ for } i = 1, \ldots, n$$

For simplicity, we express these operations by using a super-operator $\oplus$ as below:

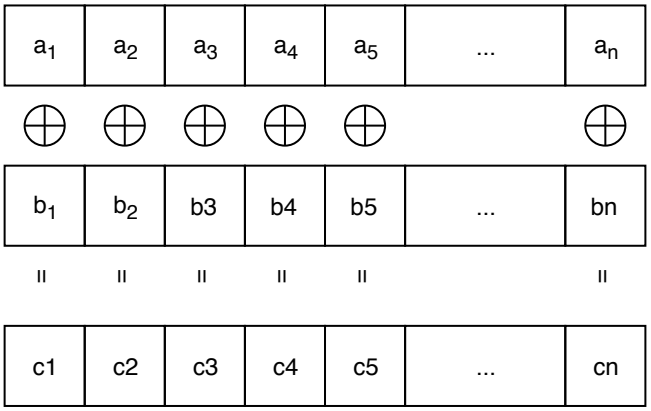

Figure 1: Element-wise operations $\vec{c} = \vec{a} \oplus \vec{b}$

We accelerate these operations in two ways: horizontal and vertical vectorizations. Note that the number of operations performed in parallel depends on bit-size of the elements and the bit-size of the vector registers. For example, if elements are 32-bit integers and registers are 128-bit wide, then there are $128/32 = 4$ operations that can be performed in parallel.

Without loss of generality, we assume that 4 element-wise operations can be performed in parallel.

4.2.1 Horizontal vectorization

Each vector is defined as an n-length array of integers. Four operations applied on four consecutive elements of $\vec{a}$ and $\vec{b}$ are performed in parallel. Therefore, operations on $\vec{a}$ and $\vec{b}$ corresponds to $\frac{n}{4}$ vector operations.

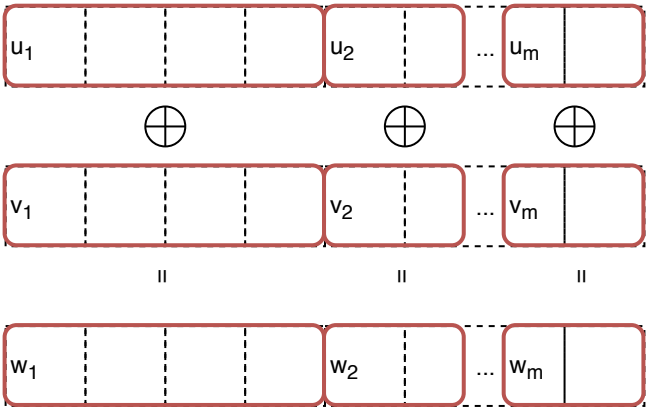

Figure 2: Horizontal vectorization

4.2.2 Vertical vectorization

Each vector is defined as an n-length array of buffers. Each element of these vectors is a buffer composed of integers, see Figure 3.

An operation $c_i = a_i \oplus b_i$ is equivalent to s operations performed on s elements each of a_i and b_i. We accelerate operations performed on each pair of buffers a_i and b_i, i.e. in vertical way as shown in Figure 4. Four operations applied on four consecutive elements of a_i and b_i are performed in parallel. Therefore, s operations on a_i and b_i corresponds to $\frac{s}{4}$ vector operations.

4.3 Vectorized modular operations in $\mathbb{F}_q$ where q is a Fermat number

In the previous section, we described the two methods to transform element-wise operations to packed vector operations. Each vector operation performs several element-wise operations in parallel. In this section, we

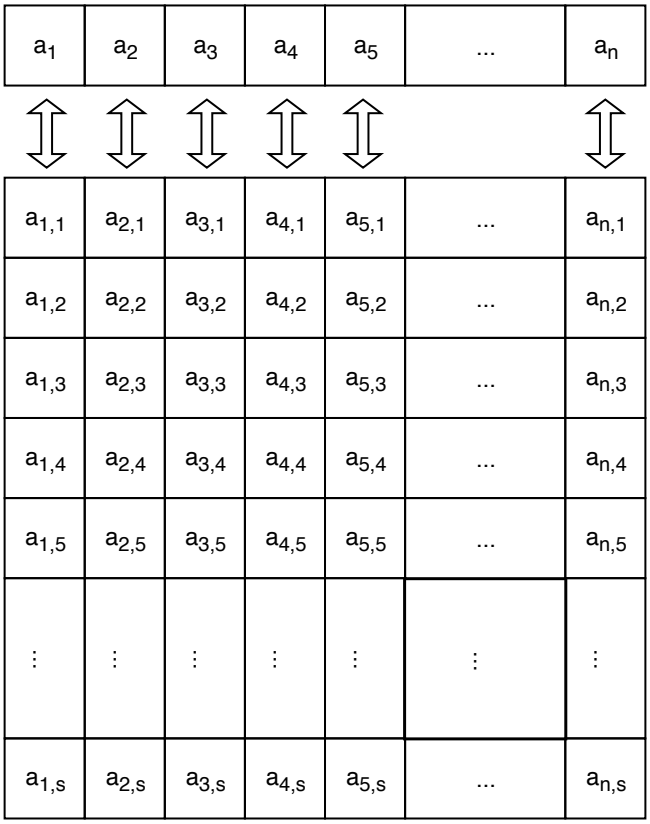

Figure 3: Vector of buffers

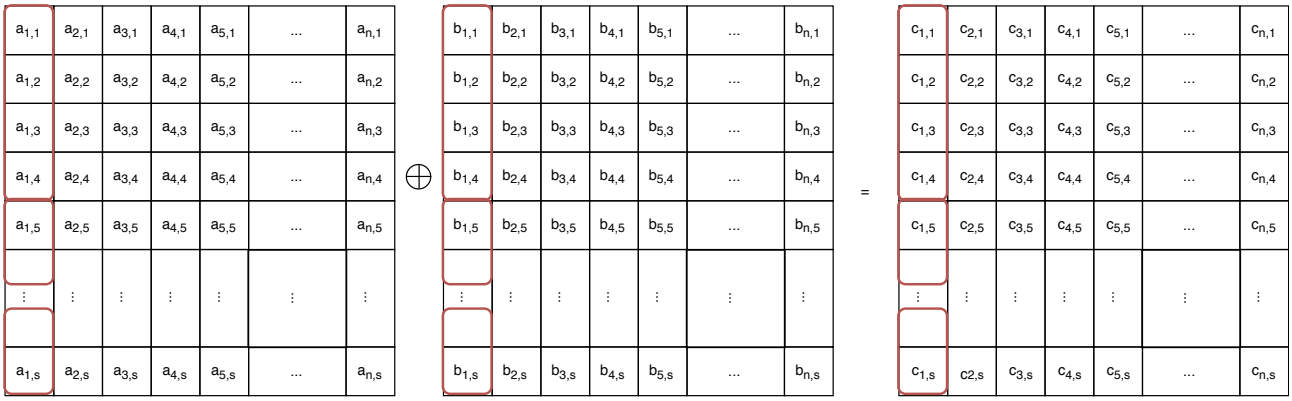

Figure 4: Vertical vectorization

describe how the parallel operations are performed on packed vectors.

Note that we focus on vectorizing modular arithmetic of finite field $\mathbb{F}_q$ where $q = 2^{2^w} + 1$, a Fermat number. This case is very interesting since it allows performing multiplicative FFT of low computational complexity (see the 1st row of Table 1) [10]

Let $\vec{q}$ denote a packed vector whose elements are q. Let $\vec{h}$ denote a packed vector whose elements are $q - 1$. Let $\vec{u}, \vec{v}$ denote two input packed vectors.

Let $vector\_operation > (\vec{u}, \vec{v})$ denote an instruction that applies an operation on two input packed vectors $\vec{u}, \vec{v}$.

- `vector_add`$(\vec{u}, \vec{v})$ does element-wise addition
- `vector_sub`$(\vec{u}, \vec{v})$ does element-wise subtraction

- `vector_mullo`$(\vec{u}, \vec{v})$ does element-wise multiplication
- `vector_or`$(\vec{u}, \vec{v})$ compute bitwise OR operation
- `vector_and`$(\vec{u}, \vec{v})$ compute bitwise AND operation
- `vector_andnot`$(\vec{u}, \vec{v})$ compute bitwise NOT of $\vec{u}$ then AND with $\vec{v}$
- `vector_cmpgt`$(\vec{u}, \vec{v})$ returns $((u_i > v_i)?1 : 0, i = 1, \ldots, n)$
- `vector_cmpeq`$(\vec{u}, \vec{v})$ returns $((u_i == v_i)?1 : 0, i = 1, \ldots, n)$
- `vector_rshift`$(\vec{u})$ returns $(x[i] \gg m, i = 1, \ldots, n)$

4.3.1 Addition

The pseudocode of addition is as below. This algorithm works as $0 \leq u_i + v_i < 2q$, hence

$$u_i + v_i = \begin{cases} u_i + v_i, & \text{if } u_i + v_i < q \\ u_i + v_i - q, & \text{otherwise} \end{cases} \tag{14}$$

Algorithm 1: Addition

Data: two vectors $\vec{u}, \vec{v}$
Result: Element-wise addition modulo q of $\vec{u}, \vec{v}$
`/* compute element-wise addition */`
1 $\vec{r} :=$ `vector_add`$(\vec{u}, \vec{v})$;
`/* subtract to` q `*/`
2 $\vec{m} :=$ `vector_sub`$(\vec{r}, \vec{q})$;
`/* If` $r_i \geq q$`, result is` m_i`. Otherwise, i.e.` $r_i < q$`, result is` r_i `*/`
3 **return** *vector_min*$(\vec{r}, \vec{m})$

4.3.2 Subtraction

The pseudocode of subtraction is as below. This algorithm works as

$$u_i - v_i = \begin{cases} u_i - v_i, & \text{if } u_i \geq v_i \\ q + u_i - v_i, & \text{otherwise} \end{cases} \tag{15}$$

Algorithm 2: Subtraction

Data: two vectors $\vec{u}, \vec{v}$
Result: Element-wise subtraction modulo q of $\vec{u}, \vec{v}$
`/* compute element-wise subtraction */`
1 $\vec{r} :=$ `vector_sub`$(\vec{u}, \vec{v})$;
`/* add by` q `*/`
2 $\vec{m} :=$ `vector_add`$(\vec{r}, \vec{q})$;
`/* If` $r_i \geq 0$`, result is` r_i`. Otherwise, i.e.` $r_i < 0$`, result is` m_i `*/`
3 **return** *vector_min*$(\vec{r}, \vec{m})$

4.3.3 Hadamard multiplication

Note, we need to check and to perform extra operations for the case both elements u_i, v_i are equal to $(q-1)$. However, this case occurs with a low probability. Hence, we implement two multiplication operations:

- Simple multiplication of u, v whose pair elements u_i, v_i are not both equal to $(q-1)$.
- General multiplication of u, v where the special case will be handled.

Obviously the simple multiplication is faster than the general multiplication. And it?s useful in performing FFT techniques in which a term of multiplications (twiddle factors) is generally known at compile-time.

The general multiplication consists in two steps:

- Step 1: compute element-wise multiplication. `vector_mullo`$(\vec{u}, \vec{v})$ does element-wise multiplication. We should handle the overflow case where
  - $u_i = v_i = (q-1)$
  - bit-size of elements smaller than 2^{w+1}

  Indeed, $u_i * v_i = 2^{2^{w+1}}$ is out-of-range of defined integer for elements. Note that in the overflow case the product result is one as $(u_i * v_i)\%q = (q-1)^2\%q = 1$.
- Step 2: perform a modulo on the multiplication result.

Let $r := (u[i] * v[i])$. Let r_{hi} and r_{lo} denote the high and low 2^w-bit values of r. We express r in the following way thanks to $q = 2^{2^w}+1$

$$\begin{aligned} r &= r_{\text{hi}} * (q-1) + r_{\text{lo}} \\ &= r_{\text{hi}} * q + (r_{\text{lo}} - r_{\text{hi}}) \\ \Rightarrow r\%q &= (r_{\text{lo}} - r_{\text{hi}})\%q \end{aligned} \tag{16}$$

Algorithm 3: Simple multiplication

Data: two vectors $\vec{u}, \vec{v}$ whose elements u_i, v_i are not both equal to $(q-1)$
Result: `simp_mul`$(\vec{u}, \vec{v})$: simple multiplication modulo q of $\vec{u}, \vec{v}$

```
/* compute element-wise addition                                          */
```
1 $\vec{s}$:= `vector_mullo`$(\vec{u}, \vec{v})$;
```
/* get low part of s                                                      */
```
2 $\vec{l}$:= `vector_blend`$(0, \vec{s}, 0x55)$;
```
/* get high part of s                                                     */
```
3 $\vec{h}$:= `vector_blend`$(0,$ `vector_rshift`$(\vec{s}, 2), 0x55)$;
```
/* result is returned by subtracting the low part by the high part        */
```
4 **return** *subtraction*$(\vec{l}, \vec{h})$

4.4 Testing

The QuadIron library contains a test suite (to ensure correctness) and a benchmark suite (to measure performance). These suites are integrated in our continuous integration system and run regularly, allowing early detection of performance regressions and bugs.

4.5 Documentation

The library comes with a Doxygen-generated documentation that associates the implementation with corresponding formulas and references.

Algorithm 4: General multiplication

Data: two vectors $\vec{u}, \vec{v}$
Result: General multiplication modulo q of $\vec{u}, \vec{v}$
/* perform a simple multiplication */
1 $\vec{s} :=$ `simp_mul`$(\vec{u}, \vec{v})$;
/* create a mask checking whether both of elements u_i, v_i are equal to $q-1$ */
2 $\vec{m} :=$ `vector_and`(`vector_cmpeq`(u, h), `vector_cmpeq`(v, h));
/* return simple multiplication if no pair of elements u_i, v_i are equal to $q-1$ */
3 **if** *is_zero*$(\vec{m})$ **then return** $\vec{s}$;
/* create a vector whose element is 1 according to the mask, otherwise zero */
4 $\vec{d} :=$ `vector_and`$(1, \vec{m})$;
/* set specified element $s_i = 1$ for $u_i = v_i = q-1$ and return it */
5 **return** *vector_or*$(\vec{s}, \vec{d})$

4.6 Contribution process

The implementation is open source (licensed under the 3-Clause BSD License) and open to external contributions. The code can be found on Github, on the scality/quadiron repository.

5 Performance

Encoding performance is expressed by the encoding speed metric that is the ratio of encoded data size to encoding time.

In our measurements, for systematic codes whose encoding performs a matrix multiplication, the encoding speed is given by:

$$\text{Encoding speed} = \frac{n-k}{k} \times \frac{\text{file size}}{\text{encoding time}} \tag{17}$$

For other codes, i.e. FFT-based codes, the encoding speed is given by:

$$\text{Encoding speed} = \frac{n}{k} \times \frac{\text{file size}}{\text{encoding time}} \tag{18}$$

We use the two different formulas that expresses the computational cost of the corresponding encoder.

Decoding performance is expressed by the decoding speed metric that is the ratio of information data size to decoding time, i.e.

$$\text{Decoding speed} = \frac{\text{file size}}{\text{decoding time}} \tag{19}$$

Simulated files are composed of k fragments each of a fixed 50-KiB size. Encoding speed is therefore linear to $\frac{n}{\text{encoding time}}$. With our optimization resulting in an encoding complexity of $\mathcal{O}(N \log k)$, the encoding speed is thus linear to $\frac{1}{\log k}$:

$$\text{Encoding speed} \sim \frac{1}{\log k} \tag{20}$$

We measured performance of RS codes $\mathcal{C}(n, k)$ over $\mathbb{F}_{2^{2^4}+1}$ on a Intel Core i5 processor rated at 2.7 GHz, running Mac OS X v.10.12.5 with 64-bit compilation, L2 cache (per core) of 256 KB and L3 cache of 3 MB. All tests were on a single core.

5.1 Vertical vs. horizontal vectorizations

We have two methods of vectorizations as described in section 4.2. Theoretically, the two optimization methods result in similar improvements. However, in practice the vertical vectorization enhances significantly the performance as it benefits from the cache memory.

We measure encoding performance of RS($k = 8, n = 128$) and RS($k = 8, n = 1024$) by using the two methods. With horizontal vectorization using 128-bit vector, encoding speed of the two codes are roughly 90 MiB/s and 99 MiB/s. These encoding speeds can be improved significantly by using vertical vectorization using also a 128-bit vector. The packet size s, i.e. number of elements in each buffers, impact on the encoding speed is shown in Figure 5. By choosing a "good" packet size, e.g. $s = 1024$, these RS codes can be encoded at very high speeds, such as 1424 MiB/s and 1294 MiB/s.

In both curves of figure 5 there are three parts. The first part that increases quickly for s from 16 to 256 represents the advantages of L2 cache on small size. The second part on larger packet sizes corresponds to the use of the L3 cache. Finally, when the packet size is too large to fit into the L3 cache, the performance decreases dramatically.

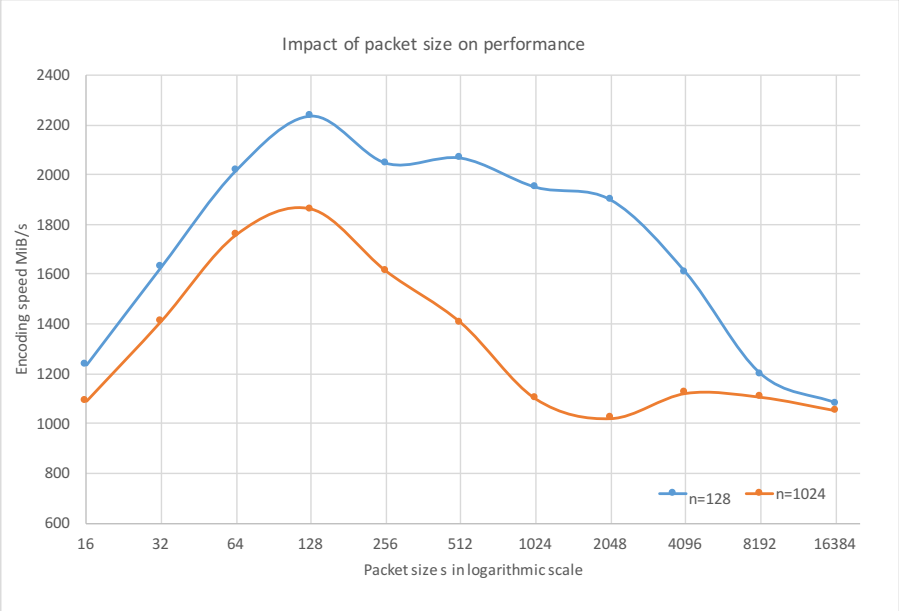

Figure 5: Vertical vectorization: impact of packet size s on encoding speed of RS codes over $\mathbb{F}_{2^{2^4}+1}$: RS($k = 8, n = 128$) and RS($k = 8, n = 1024$).

5.2 Impact of data length k

Mindful of hyper-fragmentation, we measured the following parameters:

- $n \in \{256, 1024, 2048\}$
- $k \in \{8, 16, 32\}$

Figure 6 shows the encoding speed for the three cases of code length n over different number of data fragments k. Each curve represents the encoding speed for a given n. We observe that for each code length n the speed is linear to $\frac{1}{\log k}$. The gap between the curves shows that the advantages of cache memory is more important for short code length than long ones. For short code length, encoding process can benefit from the different cache memories L2 and L3, while long code lengths can only exploit the L3 cache.

Figure 7 shows the decoding speed.

6 Applications

6.1 Decentralized Storage over the Internet

As drive density continues to be driven higher in keeping with Moore's Law, the price of storage continues to fall. This makes extra data copies cheaper. For example, generating and spreading hundreds of fragments from

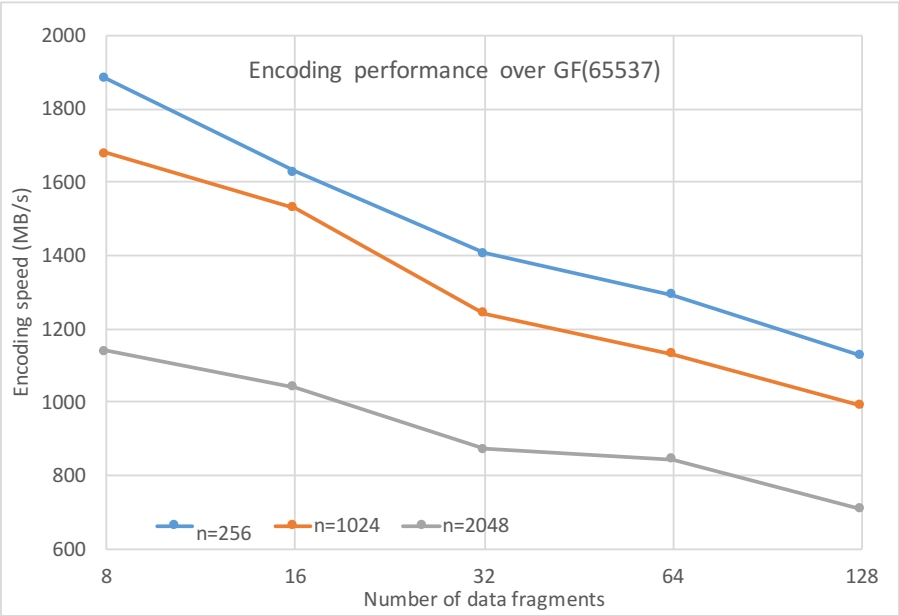

Figure 6: Encoding speed of RS codes over $\mathbb{F}_{2^{2^4}+1}$. Each curve represents a fixed n.

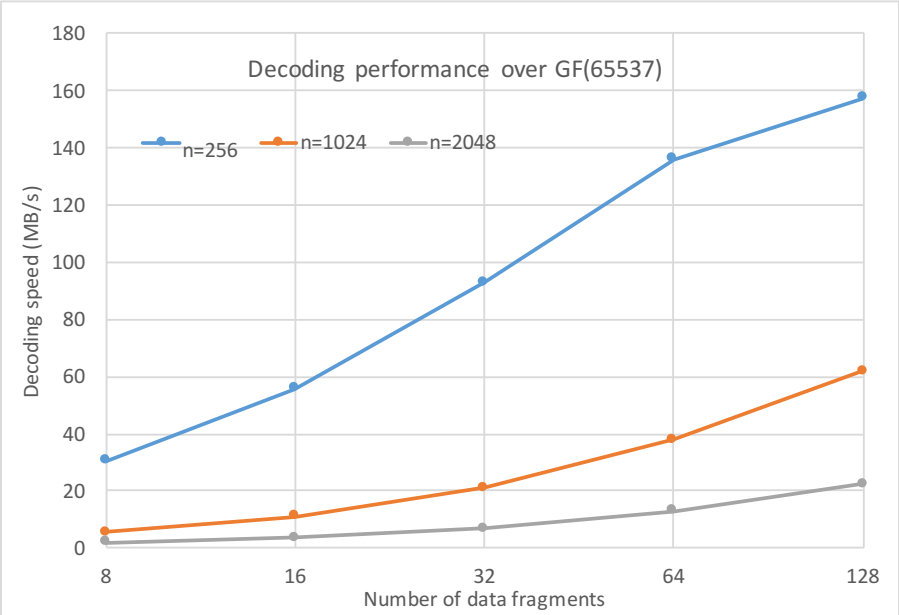

Figure 7: Decoding speed of RS codes over $\mathbb{F}_{2^{2^4}+1}$. Each curve represents a fixed n.

a file makes it possible to reconstruct the data while having only a fraction of the total data available. In this context, it is possible and interesting to develop a reliable decentralized cloud storage system that allows to exploit unused storage on the heterogeneous edge nodes (smartphones, laptops, servers, ...) of the network (see figure 8).

We have to compensate the unreliability of the underlying devices (which can go off-line anytime for any duration of time) if we want to have a reliable and resilient storage. Erasure coding is an excellent solution to increase durability while keeping the storage overhead low. Existing practical systems, such as Storj, are going toward encoding schemes like $\mathcal{C}(30, 10)$, $\mathcal{C}(60, 20)$ or $\mathcal{C}(120, 40)$.

The systematic property is interesting but not absolutely necessary in a decentralized storage application: Indeed a decentralized storage software may choose to place the plain text data fragments on the peers having

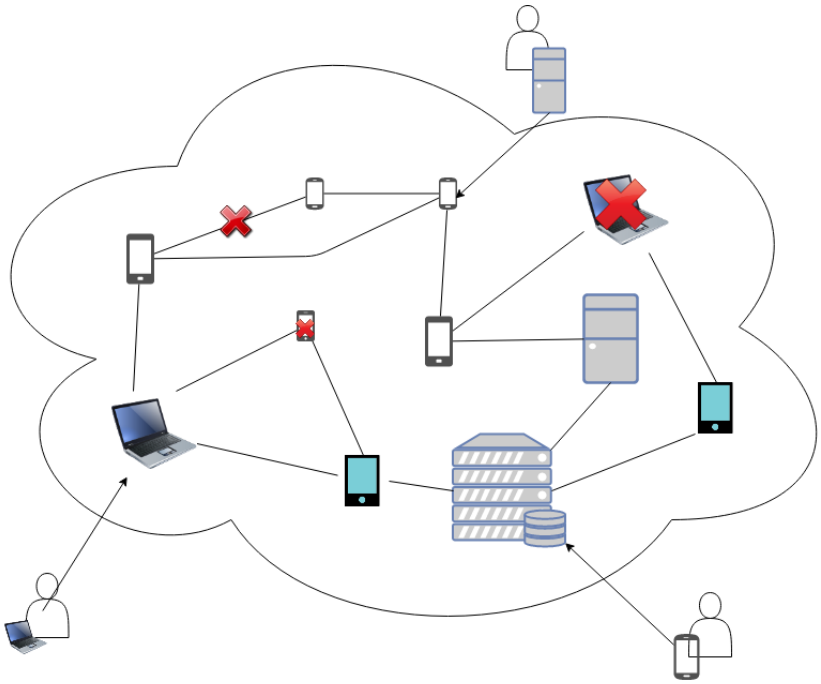

Figure 8: A decentralized storage system based on heterogeneous nodes

the best quality of service. But in an environment like the Internet where it is difficult to evaluate the quality of service of peers, or when the quality of service is very variable, the probability of reaching a peer that contains a plain text data fragment is getting lower as k gets bigger (see figure 9). Therefore the systematic property becomes useless. Moreover non-systematic code are rate insensitive.

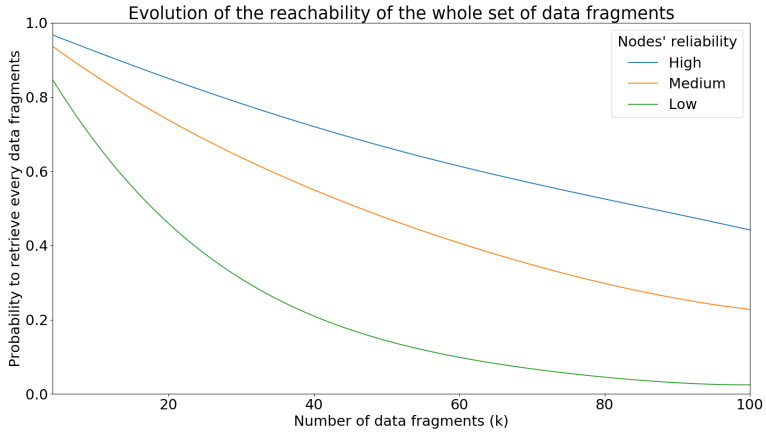

Figure 9: Evolution of the reachability of the whole set of data fragments in function of k

The adaptive rate property may also be useful in a decentralized storage application because it is not practically possible to delete and rewrite the erasure code fragments which are already in place. However if the initial durability chosen is already very high, the need for re-dimensioning the code is not critical.

The repair bandwidth is indeed a problem for RS codes but it can be mitigated in multiple ways in a decentralized storage scenario:

- Generally we cut a file in sub-blocks of fixed size called stripes. Since the stripes are spread in a very high number of peers, the loss of a peer will affect only one stripe with a very high probability.

- It is possible to detect a missing fragment at the time of the read and repair it immediately after (read-repair).
- Because of the chosen very high durability, a pro-active repair mechanism (called scrubbing) is not absolutely critical and can be scheduled rarely.

Finally one can argue that both the systematic property and the optimization of the repair bandwidth at the level of a stripe is still advantageous in order to support get_range() operations: the ability to read a portion of a stripe without decoding or by minimizing the decoding. The get_range() properties might indeed be of interest in the case of a local distributed file system or a local big data cluster, but not in a decentralized storage system which targets primarily the cloud storage use case (for archival purpose) where people generally wants the files stored and retrieved in their totality.

We do compare our erasure codes with other libraries (ISA-L, Leopard, Wirehair) for k up to 1024. The ISA-L library [16] is a performance-oriented open-source library developed by Intel that implements a systematic RS code defined over $\mathbb{F}_{2^8}$ (which limits the code length to at most 255). Leopard [17] is a performance-oriented RS codes using additive FFT techniques. Wirehair [4] is a performance-oriented hybrid LDPC library that implements a systematic code defined over $\mathbb{F}_{2^8}$ that also leverage SIMD instructions.

Figures 10, 11, 12, 13 show encoding speeds of these codes for different coding rates 1/4, 1/3, 1/2 and 2/3 respectively. We observe that for small k, ISA-L codes are very good. But for $k > 24$, our codes outperform ISA-L codes thanks to FFT operations.

LDPC encoding speed worsen as k grows. Note the graphs shows the pure decoding speed and does not show the additional time needed to retrieve the additional fragments due to the overhead (see [3] for a detailed study on this topic).

Wirehair is also less good for higher code rates.

Figures 14, 15, 16, 17 show decoding speeds of these codes for different coding rates 1/4, 1/3, 1/2 and 2/3 respectively. We observe that average decoding time of FNT codes is about only 3.5 times longer than encoding time, compared to 8 times mentioned in [10]. However our FNT decode speed is still slow compared to others. We plan to implement the approach defined in [11].

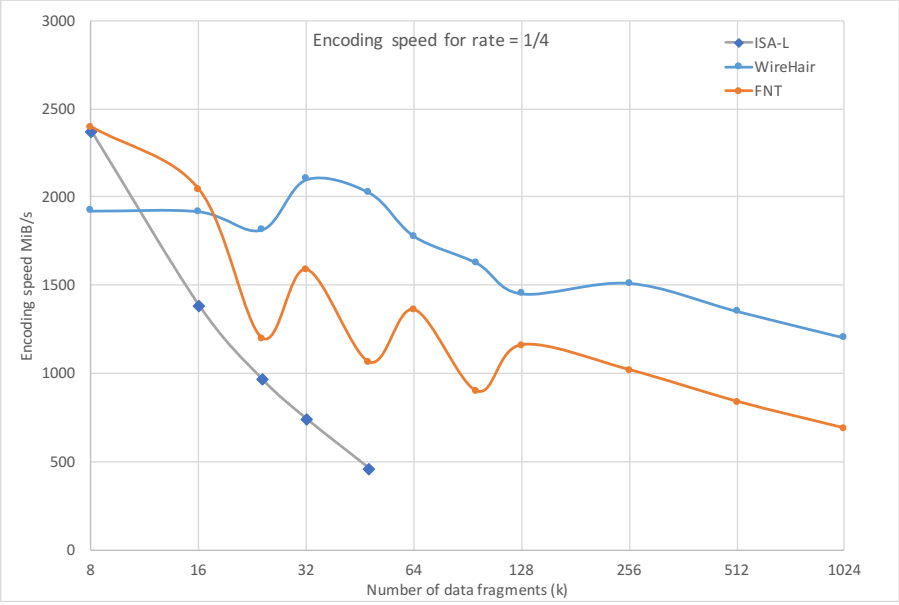

Figure 10: Encoding speed of RS codes of rate 1/4: comparison our FNT codes vs. ISA-L vs. Leopard vs. WireHair

As wee see in practical decentralized storage applications like Storj the typical code rate is 1/3 or 1/4 which means relatively high durability and relatively low number of fragments to retrieve which is good for latency. LDPC such as Wirehair show to be very good, although non MDS, we show that FNT has a relatively good performance but is still inferior to LDPC, we think we can improve FNT perf in improving the FFT method

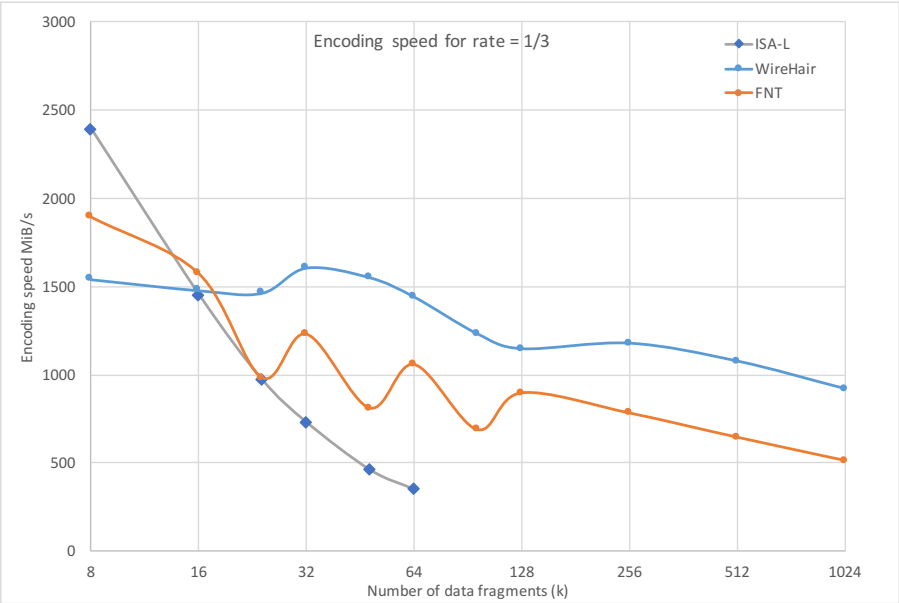

Figure 11: Encoding speed of RS codes of rate 1/3: comparison our FNT codes vs. ISA-L vs. Leopard vs. WireHair

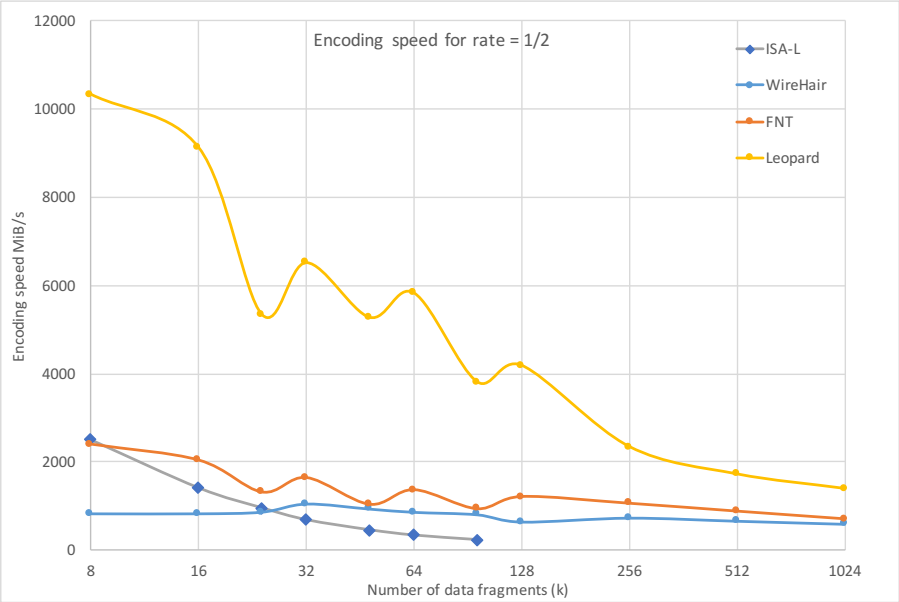

Figure 12: Encoding speed of RS codes of rate 1/2: comparison our FNT codes vs. ISA-L vs. Leopard vs. WireHair

(we currently implement a bit-reverse method that requires copying data and we could replace by other more efficient methods). We also think that accelerating additive FFT such as what has been done in Leopard [17] is very promising.]

Another interesting property for this application is the confidentiality of the erasure code, even though the client decides to encrypt or not the files beforehand, using a non-systematic FNT augments the security.

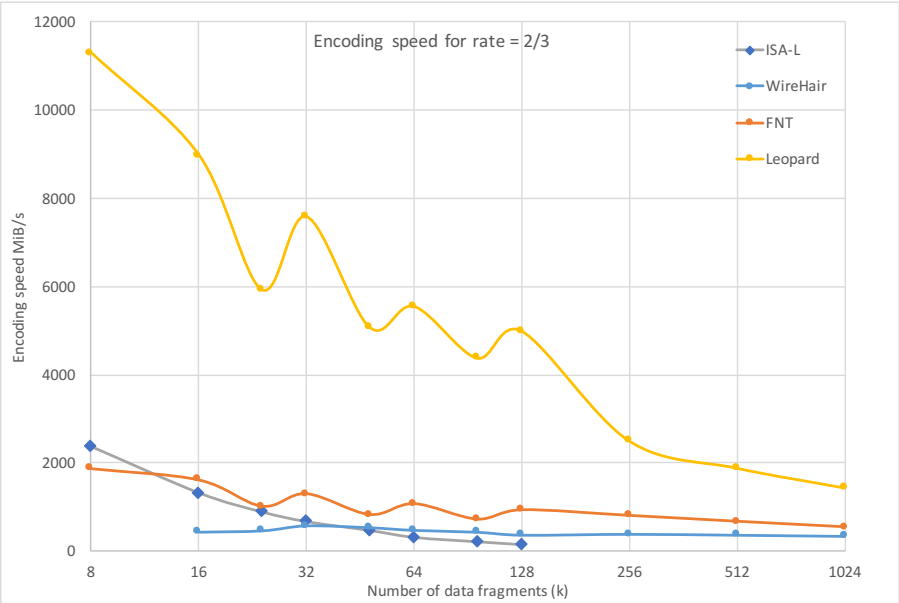

Figure 13: Encoding speed of RS codes of rate 2/3: comparison our FNT codes vs. ISA-L vs. Leopard vs. WireHair

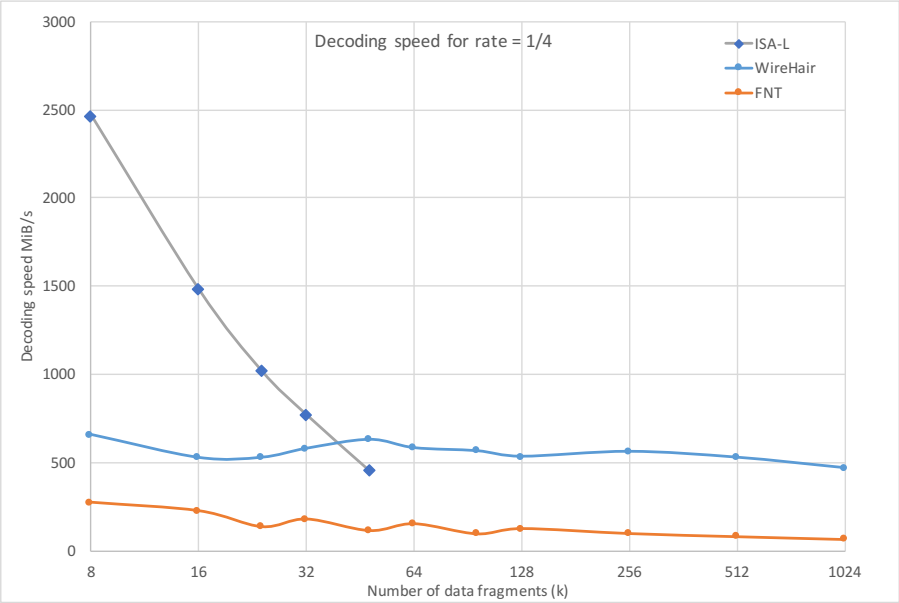

Figure 14: Decoding speed of RS codes of rate 1/4: comparison our FNT codes vs. ISA-L vs. Leopard vs. WireHair

6.2 Real-Time Video Encoding High-Quality Streaming

There are many types of video streams. For high quality ingress of a broadcast system, the master stream is typically broken up into pieces and encoded to lower quality with a few seconds of delay.

Those real-time video high quality streams with 30% interleaving (code rate 2/3) have many pieces, and are generally encoded using $\mathcal{O}(N^2)$ streaming convolutional codes e.g. like CauchyCaterpillar [18] which operates on $\mathbb{F}_{2^8}$ and is limited to 2MB/s. To go faster the technique generally used is interleaving block codes (which is how video FEC is usually done [19]). The overhead from using something fast like a XOR based LDPC on $\mathbb{F}_{2^2}$ is highly inefficient. Our MDS NTT approach is much better because we have no overhead and we can keep up

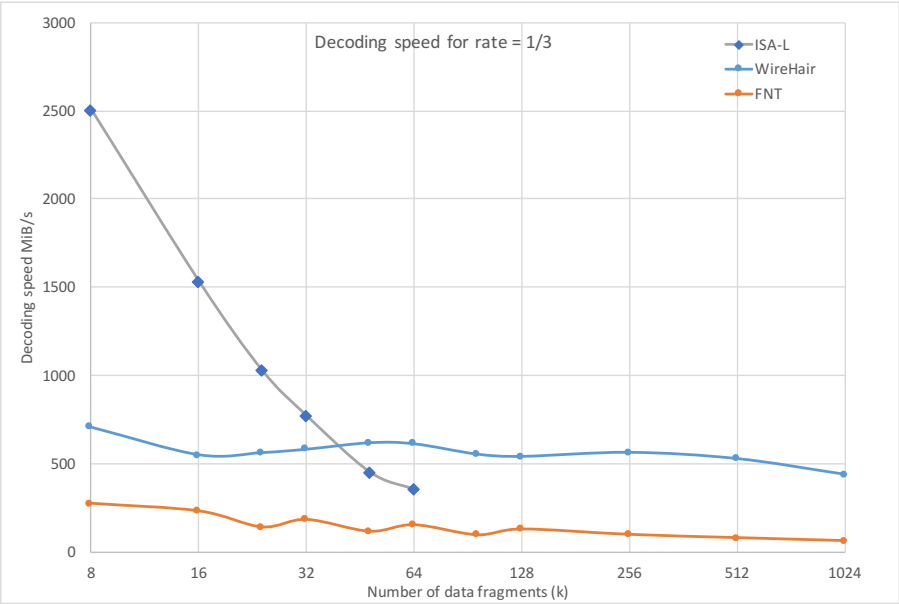

Figure 15: Decoding speed of RS codes of rate 1/3: comparison our FNT codes vs. ISA-L vs. Leopard vs. WireHair

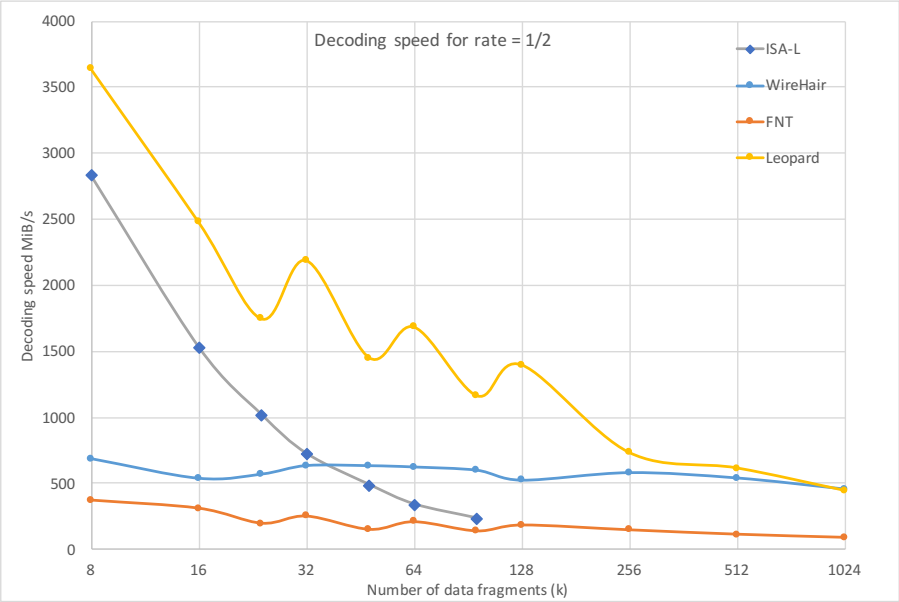

Figure 16: Decoding speed of RS codes of rate 1/2: comparison our FNT codes vs. ISA-L vs. Leopard vs. WireHair

with the speed.

In this type of application there is a need for a systematic code. As wee see in [17], [15] and [10] we can implement systematic NTT based codes with lesser but similar performance.

7 Conclusion

NTT codes offer superior encoding performance compared to matrix-based RS erasure codes for applications requiring $n \gtrapprox 24$ symbols, and are simpler than LDPC codes, while supporting all the desirable properties: Fast,

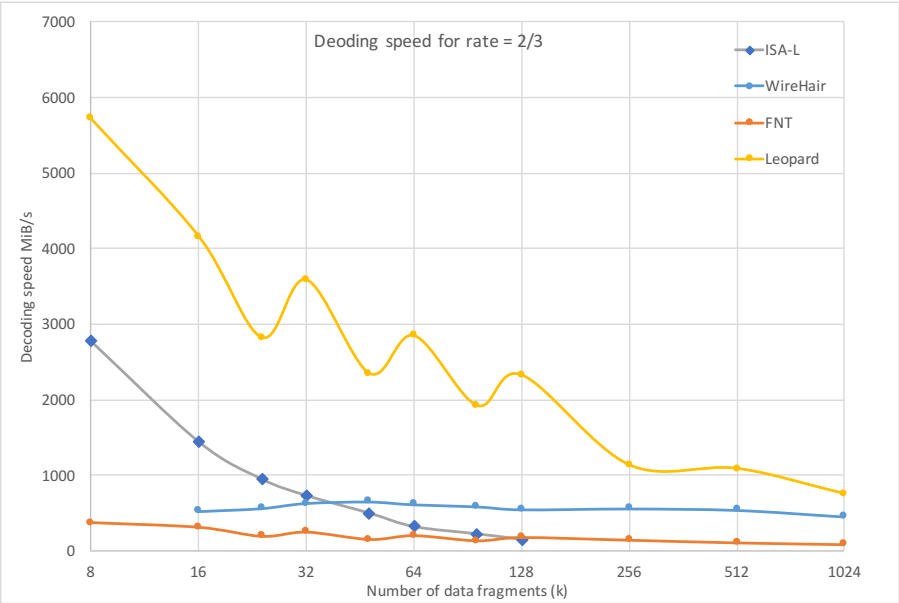

Figure 17: Decoding speed of RS codes of rate 2/3: comparison our FNT codes vs. ISA-L vs. Leopard vs. WireHair

MDS, systematic or non-systematic, confidential (for systematic codes). As we have seen the optimization of repair bandwidth as offered by LRC codes is not critical for a decentralized cloud storage application. The most important property for us remains the MDS property as a rock solid contract: being sure than if k fragments are available then the data is recoverable. Also, as we seen, FNT aka multiplicative FFT codes offer not necessarily always faster but more predictive performance than additive FFT codes for various n.

Those codes may have other potential applications such as high-quality video streaming but a lot of experiments and analysis have yet to be conducted.

8 Future Work

The next steps for the QuadIron library are in order of priority:

- Optimize the multiplicative FFT decoding, which is for now relatively slow ([11] and [20] has shown a way to optimize). It can be possible for special values of k and m, e.g. $k\%m = 0$ or $m\%k = 0$.
- Optimize additive FFTs (for now we don't have any HW acceleration).
- Implement additive FFT based systematic codes.
- Design and implement NTT based adaptive codes for multiplicative FFTs.
- Design and implement NTT based adaptive codes for additive FFTs.
- Implement Frobenius multiplicative FFT [21].
- Implement Frobenius additive FFT [22].

References

[1] Mingqiang Li. On the confidentiality of information dispersal algorithms and their erasure codes. *CoRR*, abs/1206.4123, 2012.

[2] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[3] J. S. Plank and M. G. Thomason. On the practical use of ldpc erasure codes for distributed storage applications. Technical Report CS-03-510, University of Tennessee, September 2003.

[4] Christopher Taylor. Fast and Portable Fountain Codes in C. https://github.com/catid/wirehair, 2018. [Online; accessed 1-July-2018].

[5] Hyegyeong Park, Dongwon Lee, and Jaekyun Moon. LDPC code design for distributed storage: Balancing repair bandwidth, reliability and storage overhead. *CoRR*, abs/1710.05615, 2017.

[6] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris S. Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. Xoring elephants: Novel erasure codes for big data. *CoRR*, abs/1301.3791, 2013.

[7] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *USENIX ATC 2012 (Winner of the Best Paper Award)*. USENIX, June 2012.

[8] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes*. North-holland Publishing Company, 2nd edition, 1978.

[9] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[10] A. Soro and J. Lacan. Fnt-based reed-solomon erasure codes. In *2010 7th IEEE Consumer Communications and Networking Conference*, pages 1–5, Jan 2010.

[11] S. J. Lin and W. H. Chung. An efficient (n, k) information dispersal algorithm based on fermat number transforms. *IEEE Transactions on Information Forensics and Security*, 8(8):1371–1383, Aug 2013.

[12] Y. Wang and X. Zhu. A fast algorithm for the fourier transform over finite fields and its vlsi implementation. *IEEE Journal on Selected Areas in Communications*, 6(3):572–577, Apr 1988.

[13] David G Cantor. On arithmetical algorithms over finite fields. *Journal of Combinatorial Theory, Series A*, 50(2):285 – 300, 1989.

[14] S. Gao and T. Mateer. Additive fast fourier transforms over finite fields. *IEEE Transactions on Information Theory*, 56(12):6265–6272, Dec 2010.

[15] S. J. Lin, T. Y. Al-Naffouri, Y. S. Han, and W. H. Chung. Novel polynomial basis with fast fourier transform and its application to reed-solomon erasure codes. *IEEE Transactions on Information Theory*, 62(11):6284–6299, Nov 2016.

[16] Intel. Intel intelligent storage acceleration library (intel isa-l) open source version. 01.org/intel-storage-acceleration-library-open-source-version. Online; accessed 30 April 2018.

[17] Christopher Taylor. MDS Reed-Solomon Erasure Correction Codes for Large Data in C. https://github.com/catid/leopard, 2017. [Online; accessed 1-July-2018].

[18] Christopher Taylor. Short-Window Streaming Erasure Codes. https://github.com/catid/CauchyCaterpillar, 2018. [Online; accessed 1-July-2018].

[19] Martin Ellis, Dimitrios P. Pezaros, and Colin Perkins. Performance analysis of AL-FEC for rtp-based streaming video traffic to residential users. In *PV*, pages 1–6. IEEE, 2012.

[20] Shuhong Gao. A new algorithm for decoding reed-solomon codes. In *in Communications, Information and Network Security, V.Bhargava, H.V.Poor, V.Tarokh, and S.Yoon*, pages 55–68. Kluwer, 2002.

[21] Joris van der Hoeven and Robin Larrieu. The frobenius fft. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 437–444. ACM, 2017.

[22] Wen-Ding Li, Ming-Shing Chen, Po-Chun Kuo, Chen-Mou Cheng, and Bo-Yin Yang. Frobenius additive fast fourier transform. *CoRR*, abs/1802.03932, 2018.

Authors

Vianney Rancurel is a computer professional with more than 20 years of experience designing and developing computer products. He currently leads the Scality research team which is responsible for feasibility studies, prototypes and patents/publications. Previously he worked in the messaging and telco industries on both software and hardware technologies at Bizanga, Borea Technologies, and Freescale. He was also a professor at the EPITA engineering school and was responsible for its System Research Lab. His specialties are system software programming, distributed systems, embedded systems and security. Vianney has a degree of Systems & Networks Engineering from EPITA.

Lam Pham-Sy is a research engineer working on information theory and computer science. His main research focuses on different families of forward erasure correcting codes such as Reed-Solomon codes, Low-Density Parity-Check codes, Locally Repairable codes etc. Their application covers from digital communication to data storage. He did his PhD program in a collaboration between CEA-Leti and Eutelsat S.A. on the subject of forward erasure codes for satellite communications. Afterwards, he continued his researches at ETIS laboratory and at Orange Labs. Currently he works at Scality S.A. as a research engineer whose research topics include application of erasure codes in distributed storage systems, finite field arithmetics.

Sylvain Laperche is a code craftsman. With a background in biotech engineering,he learnt how to hack bacteria before learning how to hack a computer. That changed when it studied bioinformatics, and since then he honed and applied its skill on a wide set of problematics: genome sequencing, complex embedded systems, climate modelling at European scale, mass-scale geolocation for telco industries. Its steps led him to work on distributed storage systems and he currently works as an R&D engineer at Scality. Sylvain Laperche has an Engineer's degree in Hardware, Circuit Design and Embedded Systems from ISIMA.