Performance evaluation of the Mojette erasure code for fault-tolerant distributed hot data storage

Dimitri Pertin, Didier Féron, Alexandre van Kempen, Benoît Parrein

Performance evaluation of the Mojette erasure code for fault-tolerant distributed hot data storage

Dimitri Pertin
Université de Nantes
IRCCyN UMR 6597
Rozo Systems

Didier Féron
Rozo Systems

Alexandre Van Kempen
Université de Nantes
IRCCyN UMR 6597

Benoît Parrein
Université de Nantes
IRCCyN UMR 6597

Abstract

Packet erasure codes are today a real alternative to replication in fault tolerant distributed storage systems. In this paper, we propose the Mojette erasure code based on the Mojette transform, a formerly tomographic tool. The performance of coding and decoding are compared to the Reed-Solomon code implementations of the two open-source reference libraries namely ISA-L and Jerasure 2.0. Results clearly show better performances for our discrete geometric code compared to the classical algebraic approaches. A gain factor up to 2 is measured in comparison with the ISA-L Intel . Those very good performances allow to deploy Mojette erasure code for hot data distributed storage and I/O intensive applications.

1 Introduction

Storage systems rely on redundancy to face ineluctable data unavailability and component failures. For its simplicity, data replication is the de facto standard to provide redundancy. For instance, three-way replication is the storage policy adopted by major file systems such as Hadoop Distributed File System [11] and Google File System [1]. While being straightforward to implement, plain replication typically incurs high storage overheads. It has now been acknowledged that erasure codes can significantly reduce the amount of redundancy compared to replication while offering the same data protection [13].

However, these storage savings come at a price in terms of additional complexity, as data must be encoded during write operations, and decoded during read operations. Very efficient coding operations are thus keys to maintain transparent operations for I/O intensive applications. Since data replication has higher storage costs but performs faster than erasure codes, storage systems tend to differentiate between *cold* data (i.e. not frequently accessed, such as in long term storage) and *hot* data, typically data that is frequently accessed. In practice, plain replication is used for I/O intensive applications due to fast data accesses while erasure codes are limited to long-term storage because of their extra complexity.

Reed-Solomon (RS) are the most popular codes as they provide deterministic general-purpose codes without limit on the parity level. They are mostly implemented in their systematic form, meaning that the information data is a part of the encoded data. In addition, they are known to be Maximum Distance Separable (MDS) thus providing the optimal reliability for a given storage overhead. The former definition of RS codes are based on Vandermonde matrices and expensive Galois field operations. Implementing such codes in an efficient manner is therefore challenging. One of the best known implementation is provided by Jerasure [9], an open-source library that relies on Cauchy generating bitmatrices to only perform XOR operations, thus avoiding the costly multiplications. Recently, Intel released ISA-L, a performance-oriented open-source library [4] that implements RS codes leveraging SIMD instructions. To the best of our knowledge, these two are the most efficient implementations publicly available.

In this paper, we propose to use the *Mojette* transform [2], a formerly tomographic tool, to implement a high-performance erasure code. The Mojette transform is a discrete and exact version of the Radon transform and relies on discrete geometry, contrary to the classic algebraic code definition. By nature, the Mojette transform provides a non-systematic erasure code. The geometric approach coupled with an optimized implementation help to perform very fast encoding and decoding operations, handling I/O intensive applications such as virtualization or databases, that access small blocks of data (4 KB or 8 KB) in a random pattern [8]. Those block sizes fit the general-purpose file systems requirements such as *ext4* or *Btrfs*.

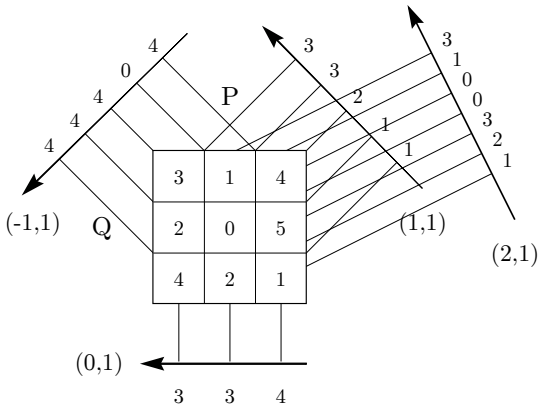

Figure 1: Mojette transform of a 3×3 image for directions (p,q) in the set $\{(-1,1),(0,1),(1,1),(2,1)\}$. Addition is done here modulo 6.

2 The Mojette Erasure Code

This section presents how the Mojette transform is used to encode data, the uniqueness conditions of the reconstruction solution and its inverse algorithm enabling the decoding. Finally, the end of this section details how the Mojette transform is used as an erasure code in practical systems.

2.1 Forward Mojette Transform

The forward Mojette transform is a linear operation that computes a set of 1D projections at different angles, from a discrete image $f : (k,l) \mapsto \mathbb{N}$, composed of $P \times Q$ *pixels*. A projection direction is defined by a couple of co-prime integers (p,q). Projections are vectors of variable sizes whose elements are called *bins*. A bin in the Mojette transform of f is characterised by its position b in the projection which corresponds to a discrete line of equation $b = -kq + lp$. Its value is the sum of the centered pixels along the line:

$$(M_{(p,q)}f)(b) = \sum_{k=0}^{P-1} \sum_{l=0}^{Q-1} f(k,l)[b = -kq + lp], \qquad (1)$$

where, $[\cdot]$ is the Iverson bracket ($[P] = 1$ whenever P is true, 0 otherwise). The number of bins B of a projection depends on the projection direction (p,q) and the lattice size $P \times Q$:

$$B(p,q,P,Q) = |p|\,(Q-1) + |q|\,(P-1) + 1. \qquad (2)$$

Figure 1 gives an example of the forward Mojette transform for a 3×3 integer image. The process transforms the 2D image into a set of $I = 4$ projections along the directions of the following set: $\{(-1,1),(0,1),(1,1),(2,1)\}$. For the sake of this example, addition is arbitrarily done modulo-6 (but any addition works). The complexity $\mathcal{O}(PQI)$ is linear with the number of projections and the number of grid elements. Note that some border bins are the exact copy of some pixels. This remark will help to understand how starts the inverse transform algorithm.

2.2 Inverse Mojette Transform

In this section, we first expose the reconstruction criterion on the projection set which yields to a unique reconstructed image. Then, we detail how is implemented the reconstruction algorithm.

Reconstruction Criterion Katz has shown that for a $P \times Q$ lattice, the reconstruction is possible given a projection set S_I if one of the following criterion is verified [5]:

$$P \leq \sum_{i=0}^{I-1} |p_i| \text{ or } Q \leq \sum_{i=0}^{I-1} |q_i|, \qquad (3)$$

where I is the number of projections involved in the reconstruction process.

In the example of the Figure 1, we see that each subset of 3 projections $\{(p_0,q_0),\ldots,(p_2,q_2)\}$ is such that $\sum_{i=0}^{2} |q_i| = 3$. Thus, the 4 projections in Figure 1 depicts a redundant representation of the image, where any 3 projections among these 4 can be used for reconstruction.

Inverse Mojette Algorithm The reconstruction algorithm aims at finding a reconstructible bin and to write its value in the image by back-projection. Bins are reconstructible when they result from a unique pixel of the image. Once a bin is reconstructed, its contribution is subtracted from all the projections involved in the reconstruction, thus paving the way to reconstruct further bins. As the forward algorithm, the Mojette inverse is linear with the number of projections I and the number of elements $P \times Q$ in the grid.

Observing that the reconstruction propagates from the image corners to its center, Normand et al. [6] showed that given an image domain and a projection set, a dependency graph between the image pixels can be found. Within this graph, considering that a single projection is dedicated to the reconstruction of a single line of the image, a reconstruction path can be pre-determined. We refer the interested reader to [6] due to lack of space.

2.3 Properties of the Mojette Erasure Code

The Mojette erasure code extends the application of the Mojette transform, originally designed for images, to any

type of data. As the Mojette transform creates a redundant representation, it appears to be an appealing candidate to provide failure tolerance in storage systems. In classic coding theory words, we consider the k lines of the Mojette array as the input data packets, and we compute n projections as the set of encoded packets. The Mojette erasure code is therefore non-systematic here. Note that a systematic version of the Mojette is currently under development. Since the size of projection varies with the parameters (p,q) we consider for each projection that $q_i = 1$ to limit the bin overhead (as proposed in [7]). Then the Katz criterion proves that if we get any k out of the n projections, it is possible to exactly reconstruct the array. This way, the storage system is able to face the unavailability of up to any $n-k$ storage nodes.

In practice, we can observe that some bins are never used during reconstruction whatever the projection set used for the process [12]. Removing these bins from the encoding process, particularly when p increases, significantly limits the projection size variation and therefore yields to a negligible storage overhead relative to the MDS case.

3 Erasure Code Micro-benchmark

In this section, we evaluate the performance of our new erasure code compared to the Jerasure and ISA-L libraries. Firstly, we describe our Mojette implementation and then present our two competitors. Secondly, we detail the experiment setup to finally depict the results and analysis in the last section.

Mojette We implemented a non-systematic version of the Mojette erasure code in C. In practice, pixel size should fit a computer word to improve performance based on XOR operations. Since x86 architectures provide Streaming SIMD Extensions (SSE) instruction set, pixel and bin sizes are set to 128 bits to benefit from high-performance XOR computations. The Mojette encoding requires at most $k-1$ XORs per computed bin (and zero XOR for bins at projection edges). Similarly for decoding, at most $k-1$ XORs are required per reconstructible pixel. The progressive reconstruction from left to right of connected pixels (as proposed in [6]), coupled with a drastic reduction of updates, guarantees spatial memory locality, thus high-performance computation.

Jerasure The first competitor is the systematic Vandermonde implementation of RS codes from the open-source Jerasure 2.0 library [9]. We choose their Vandermonde implementation since it performs better than their Cauchy-based implementation for such small packets size. The Galois field size is set to $w = 8$ to fit our erasure code configuration (n,k).

Intel ISA-L The second competitor is the RS implementation provided in Intel ISA-L open-source library [4]. It is one of the fastest systematic erasure code implementation since it makes intensive usage of the x86 architecture features such as *xmm* registers and SSE instructions.

3.1 Experiment Setup

We conducted all experiments on small data blocks of *BlockSize* equals to 4 KB and 8 KB (that fit block-based file-system like *ext4*). For encoding, we consider a single data block filled with random data. For decoding, we record the performance as we increase the number of erasures up to the failure tolerance. With systematic codes (implementations of ISA-L and Jerasure), erasures only concern data packets (no decoding otherwise).

Two erasure code configurations are considered for the benchmark: (n,k) equals $(6,4)$ and $(12,8)$, preventing from 2 and 4 failures respectively. All the computations are performed in memory, with no disk I/O operation. Furthermore, we do not take into account pre-computations such as the matrix inversion or deterministic reconstruction path respectively for the RS and Mojette implementations. Since we measure optimized encoding and decoding functions that are mostly computation-bounded, with the data entirely located in L1 and L2 cache, we use the *RDTSC* instruction that returns the *time stamp counter* (TSC) which is incremented on every CPU clock cycle [3]. For all tested implementations, the standard deviation is too negligible to be represented (less than 1%).

All the experiments are done on a single processor running Linux 3.2 and Debian Wheezy over an x86-64 architecture. It embeds a 1.80 GHz Intel Xeon processor, with 16 GB of RAM and cache sizes of 32, 256 and 10240 KB for respectively L1, L2 and L3 cache levels.

3.2 Results

We now present the results of encoding and decoding throughput for various *BlockSize* and code parameters. For the sake of comparison, we plot the optimal performance recorded by the **memcpy()**. More precisely, the optimal encoding is given by the **memcpy()** of n packets of $\frac{BlockSize}{k}$ bytes while the optimal decoding is the **memcpy()** of only k packets among the n encoded. Once again, note that the Mojette is implemented as a non-systematic code, thus increasing the overall computation compared to the two other systematic codes.

Encoding Figure 2 shows the encoding performance recorded for 4 KB (top) and 8 KB (bottom) data blocks for the $(6,4)$ and $(12,8)$ codes. The first observation is that the Mojette erasure code outperforms the two other

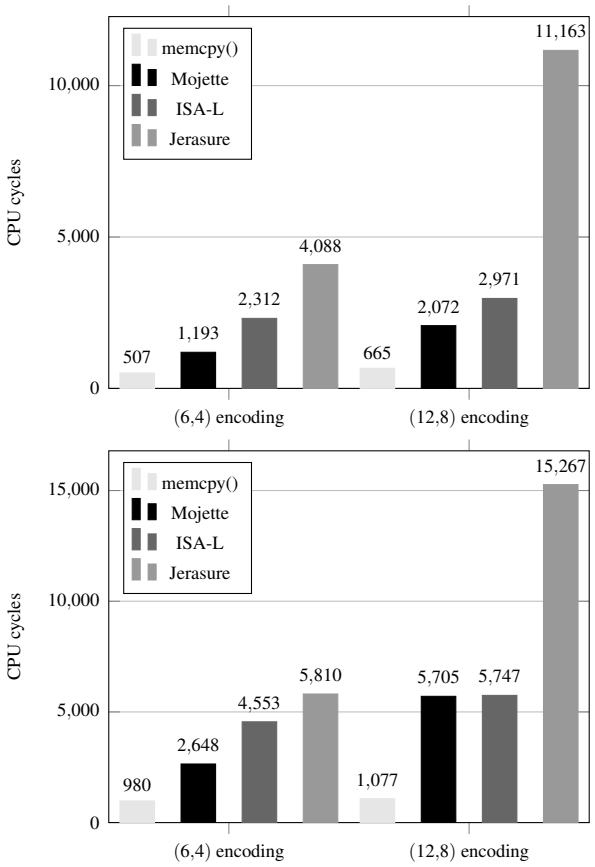

Figure 2: Encoding performance for an input data block of 4 KB (**top**) and 8 KB (**bottom**) depending on the code parameters $(n = 6, k = 4)$ or $(n = 12, k = 8)$.

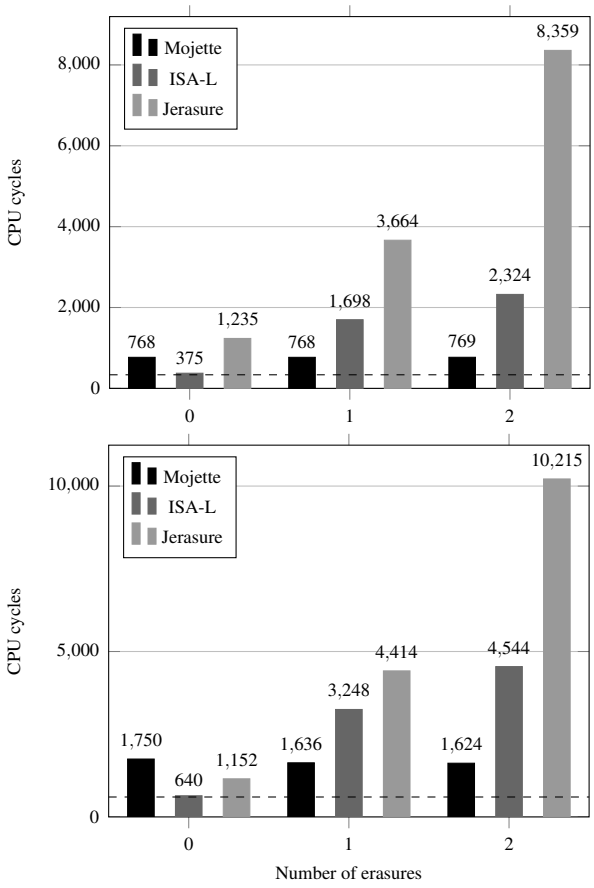

Figure 3: Decoding performance of a $(n = 6, k = 4)$ code for an input data block of 4 KB (**top**) and 8 KB (**bottom**) depending on the number of failures. The dashed line depicts the optimal value of the **memcpy()** respectively measured at 336 (4 KB) and 603 (8 KB).

implementations in every tested settings. For example, to encode a 4 KB block with a $(6,4)$ code, the Mojette implementation divides the number of CPU cycles by a factor of 1.94 and 3.42 when respectively compared to ISA-L and Jerasure. While the Mojette implementation still provides the closest performance from the optimal value of the **memcpy()**, the improvements are mitigated for the code $(12,8)$, and especially versus ISA-L for a block of 8 KB. This is mainly due to our non-systematic design. Indeed, to encode a 8 KB block with a $(12,8)$ code, the encoder splits 8 KB into 8 packets of one kilobyte and produces 4 encoded packets for systematic codes, while non-systematic codes have to compute 12 encoded packets thus performing 3 times more computations. Finally, we notice that, as expected, the CPU cycles number linearly increases with the *BlockSize*, as well as with the number of blocks to be encoded thus experimentally confirming the linear complexity of the Mojette transform.

Decoding We respectively plot in Figures 3 and 4 the number of CPU cycles required to decode the data for the same codes as before, depending on the number of failures. We still emphasize here the differences between systematic and non-systematic implementations. Since RS codes are systematic, **when no failure occurs**, they should achieve optimal performance (equivalent to a **memcpy()**) as the decoding process boils down to the copy of k data blocks in memory. For example, we see that ISA-L delivers the optimal performance for every 0-erasure settings. Note that our ongoing implementation of the Mojette in systematic-form would also provide the same results.

We now focus on the results in the presence of failures, when decoding operations are therefore involved (i.e. we do not just retrieve the data packets in memory). Results for the code $(n = 6, k = 4)$ on a 4 KB block, depicted on top of the Figure 3 show that the number of CPU cycles is divided by a factor of 2.2 and 4.8 when respectively compared to ISA-L and Jerasure for a single failure. These factors are even higher when two erasures occurred. In fact, due to its non-systematic form, the

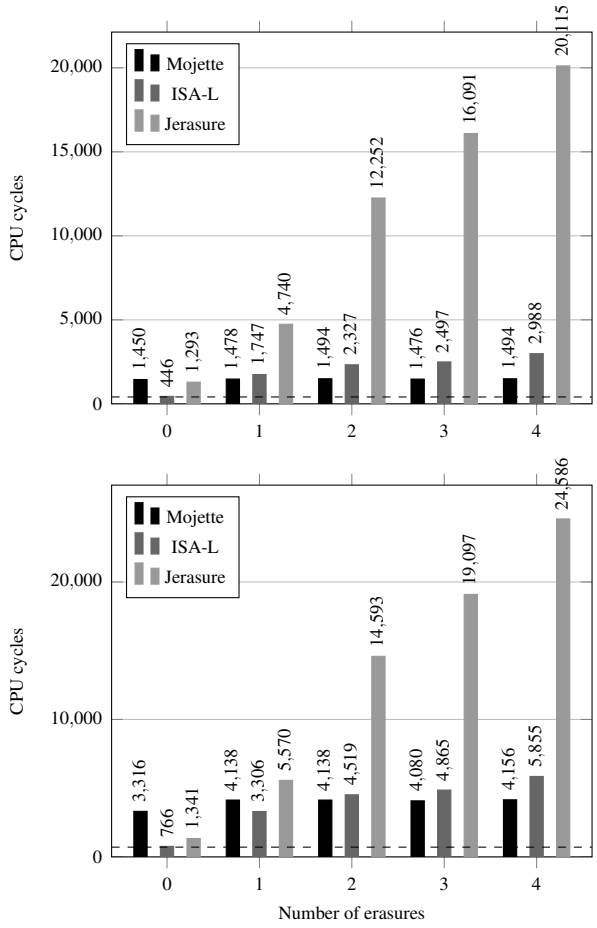

Figure 4: Decoding performance of a $(n = 12, k = 8)$ code for an input data block of 4 KB (**top**) and 8 KB (**bottom**) depending on the number of failures. The dashed line depicts the optimal value of the **memcpy()** respectively measured at 411 (4 KB) and 711 (8 KB).

set of projections used has no influence on the decoding performances of the Mojette. On the contrary, the performances of Jerasure and ISA-L progressively decrease with the number of erasures. Although this performance gap is reduced for the code $(n = 12, k = 8)$, especially versus ISA-L, results presented in Figure 4 still confirm the above observations.

4 Conclusion

Erasure codes are well known to incur a high computational penalty due to their inherent coding operations, thus preventing them from being deployed in I/O intensive applications. In this paper, we advocated that the *Mojette transform* is a particularly suitable tool to design high-performance erasure code. We implemented and evaluated our new erasure code compared to the best-known implementations, namely ISA-L and Jerasure. Results show that this paradigm shift towards a geometric approach enables the *Mojette*-based implementation to significantly improve the throughput of coding and decoding operations. As non-systematic, the proposed code can still bring better throughputs in a foreseeable future. A Mojette erasure code implementation is currently deployed in an open-source project RozoFS [10]. We believe that this new code paves the way to the use of erasure codes in I/O intensive applications.

5 Acknowledgements

This material is based upon work supported by the Agence Nationale de la Recherche (ANR) through the project FEC4Cloud (ANR-12-EMMA-0031-01).

References

[1] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[2] GUÉDON, J. P., AND NORMAND, N. The Mojette transform: The first ten years. In *Discrete Geometry for Computer Imagery*, E. Andres, G. Damiand, and P. Lienhardt, Eds., vol. 3429 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, pp. 79–91.

[3] INTEL. Using the RDTSC instruction for performance monitoring. Tech. rep., Intel Corporation, 1997.

[4] ISA-L. https://01.org/intel%C2%AE-storage-acceleration-library-open-source-version.

[5] KATZ, M. *Questions of uniqueness and resolution in reconstruction from projections*. Springer-Verlag Berlin ; New York, 1978.

[6] NORMAND, N., KINGSTON, A., AND ÉVENOU, P. A geometry driven reconstruction algorithm for the Mojette transform. In *DGCI*, vol. 4245 of *LNCS*. Springer Berlin Heidelberg, 2006, pp. 122–133.

[7] PARREIN, B., NORMAND, N., AND GUÉDON, J. P. Multiple description coding using exact discrete Radon transform. In *Proceedings of the Data Compression Conference* (Washington, DC, USA, 2001), DCC '01, IEEE Computer Society, pp. 508–.

[8] PERTIN, D., DAVID, S., ÉVENOU, P., PARREIN, B., AND NICOLAS, N. Distributed file system based on erasure coding for I/O-intensive application. *The 4th International Conference on Cloud Computing and Services Science, CLOSER 2014*, 13-16 (Apr. 2014).

[9] PLANK, J. S., AND GREENAN, K. M. Jerasure: A library in C facilitating erasure coding for storage applications–version 2.0. Tech. rep., Technical Report UT-EECS-14-721, University of Tennessee, 2014.

[10] ROZOFS. https://github.com/rozofs/rozofs.

[11] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on* (May 2010), pp. 1–10.

[12] VERBERT, P., RICORDEL, V., AND GUÉDON, J. P. Analysis of Mojette transform projections for an efficient coding. In *Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS)* (Lisboa, Portugal, Apr 2004), pp. –.

[13] WEATHERSPOON, H., AND KUBIATOWICZ, J. Erasure coding vs. replication: A quantitative comparison. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems* (London, UK, 2002), IPTPS '01, Springer-Verlag, pp. 328–338.