Benchmarking Reed-Solomon and Cauchy Reed-Solomon Erasure Coding Algorithms

Peeyush Gupta
MS, Computer Engineering
Iowa State University
Ames, Iowa
peeyush@iastate.edu

I. Introduction

Erasure codes are a powerful tool for fault-tolerant data storage in distributed systems. They provide data redundancy by encoding the original data into a larger set of data fragments that are spread across multiple storage nodes. In the event of node failures, erasure codes can reconstruct the original data by using information from the remaining non-failed nodes.

There are several different types of erasure codes, including Reed-Solomon codes, Cauchy-Reed-Solomon codes, and Fountain codes. Among these, Reed-Solomon codes are widely used in practice, particularly in RAID (redundant array of independent disks) systems, while Fountain codes are used in peer-to-peer and content delivery networks.

At a high level, erasure codes work by dividing the original data into k data fragments and encoding them into n code fragments. These n fragments are then distributed across n+m storage nodes, where m is the number of erasures the system can tolerate. In the event of node failures, the erasure code can reconstruct the original data by using the remaining n-m non-failed nodes.

Erasure codes are especially useful in large-scale distributed systems, where node failures are common and expected. By providing fault tolerance and reliability, erasure codes allow these systems to continue functioning even in the presence of node failures. As a result, erasure codes are a critical component of many storage systems, such as cloud storage, distributed file systems, and content delivery networks.

II. Motivation

There are several kinds of Erasure Coding as discussed above like Reed-Soloman and Cauchy Reed-Soloman Erasure Coding. A lot of factors may effect the performance of the erasure codes. These effects may include number of data blocks required, number of parity blocks required etc. If these factors are not fine tuned it may decrease the performance of the whole system. The motivation of this paper to understand how these factors can effect erasure code's performance. At the end of the paper, the results are compared with other research in the similar category. This paper only deals with the encoding performance of Cautch Reed-Solomon erasure coding.

III. Reed-solomon code

Reed-Solomon [5] codes are an essential type of error-correcting codes, which are used in various digital communication and storage applications. These codes are used to ensure that data transmission and storage are error-free. The Reed-Solomon encoder adds extra bits of redundant data to the digital data block, while the Reed-Solomon decoder processes each block and attempts to correct errors and recover the original data.

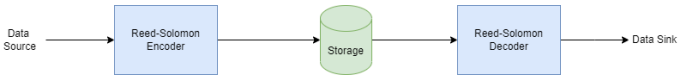

Fig. 1. Encoding and Decoding Reed-Solomon.

BCH codes are a subset of the linear block codes also known as Reed-Solomon codes. With s-bit symbols, the Reed-Solomon code is designated as RS(n,k). To construct an n-symbol codeword, the encoder takes k data symbols with s bits apiece and adds parity symbols. There are n, k, and s bits of parity symbols. Where 2t=n-k, the Reed-Solomon decoder can fix up to t symbols in a codeword that contain mistakes.

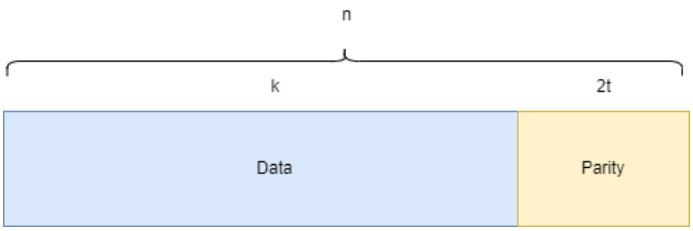

Fig. 2. Relation between n,k and t

Given a symbol size of s, the maximum codeword length for a Reed-Solomon code is $n = 2^s - 1$. For instance, a code using 8-bit symbols (s=8) can have a maximum length of 255 bytes. A number of data symbols can be made zero at the encoder, not transmitted, and then reinserted at the decoder to shorten Reed-Solomon codes. The amount of processing power required to encode and decode Reed-Solomon codes is related to the number of parity symbols per codeword. A large value of t means that a large number of errors can be corrected, but it requires more computational power than a small value of t. This will be verified in the further experiments. A single symbol error happens when either one or all of a symbol's bits are incorrect. For instance, 16 symbol mistakes can be fixed with RS(255,223). In the worst situation, 16 bit errors might happen, each in its own symbol (byte), requiring the decoder to fix them. The decoder corrects 16 x 8 bit errors when 16 whole byte errors occur, which is the best case scenario. When several bits in the codeword are received incorrectly, burst mistakes, Reed-Solomon codes are especially well suited to repair them.

Algebraic Reed-Solomon decoding techniques can fix mistakes and erasures. When the location of an incorrect symbol is known, an erasure happens. Up to t mistakes or up to 2t erasures can be fixed by a decoder. In a digital communication system, the demodulator frequently provides erasure information. The demodulator, for instance, marks received symbols that are probably in mistake. There are three possible results when a codeword is decoded. It is always possible to retrieve the original transmitted codeword if 2s + r $<$ 2t (s mistakes, r erasures). If not, either the decoder may mistakenly recover an incorrect codeword without warning or it will misdecode and be unable to recover the actual codeword. Each of the three alternatives has a different probability depending on the specific Reed-Solomon code as well as the quantity and distribution of faults.

The advantage of using Reed-Solomon codes is that the probability of an error remaining in the decoded data is (usually) much lower than the probability of an error if Reed-Solomon is not used. This is often described as coding gain. Reed-Solomon codes provide high coding gain due to their ability to correct multiple errors in a block of data. The coding gain increases as the number of errors corrected by the code increases. In practical communication systems, Reed-Solomon codes can provide coding gains of up to 10 dB [5] or more, which translates to a significant reduction in transmitter power or an increase in the range of the communication system.

In addition to their high coding gain, Reed-Solomon codes are also highly efficient in terms of their encoding and decoding complexity. This makes them a popular choice for various applications, including digital storage systems and digital communication systems.

A. Encoding and Decoding process

Reed-Solomon codes use a specialized area of mathematics known as Galois fields or finite fields, where arithmetic operations on field elements always produce a result in the field. Finite field arithmetic is used extensively in Reed-Solomon encoding and decoding. The arithmetic operations used in Reed-Solomon require special hardware or software functions to implement. The Reed-Solomon codeword is generated using a special polynomial called the generator polynomial. This generator polynomial is constructed in such a way that all valid codewords are exactly divisible by it. The general form of the generator polynomial is

$$g(x) = (x - a^i)(x - a^{i+1})...(x - a^{i+2t})$$

And code word is produced using :

$$c(x) = g(x).i(x)$$

IV. The Cauchy Reed-Solomon Code

As mentioned in paper [3], Jerasure Library uses The Cauchy Reed-Solomon code for encoding and decoding of data, this paper deals with Cauchy Reed-Solomon encoding.

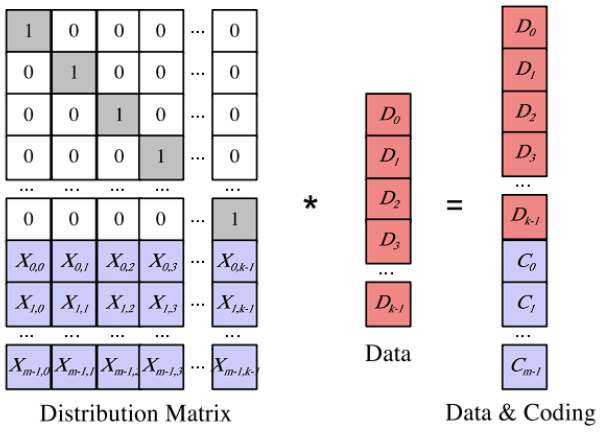

Fig. 3. Simple Matrix based encoding. Source : [3]

Assume we have w bits in each of the k data words, m coding words, and k data words. As demonstrated in Figure 3, a matrix-vector product can be utilised to characterise the state of a matrix-based coding system. This matrix, which has the dimensions (k + m)* k, is known as the distribution matrix. Its constituent integers, which range from 0 to 2w-1, employ Galois Field arithmetic, in which addition is identical to XOR and multiplication is accomplished in a variety of ways.

The remaining m rows of the distribution matrix make up the coding matrix, and the first k rows make up an identity matrix with the dimensions as k*k. We create a product vector by multiplying the distribution matrix by a vector that includes both data and coding words. We must do m dot products of the coding matrix with the data in order to encode.

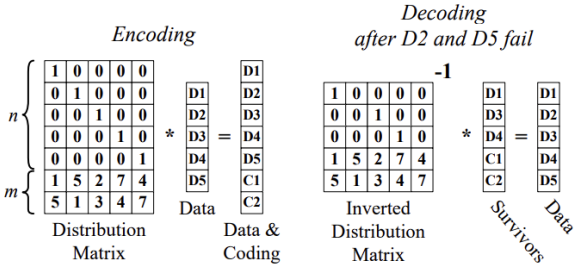

Fig. 4. Encoding and Decoding. Source : [2]

In decoding, it is observed that each word in the system corresponds to a row of the distribution matrix. In the event of device failure, we create a decoding matrix using k rows of the distribution matrix corresponding to non-failed devices. It's important to note that this matrix multiplied by the original data equals the k survivors whose rows we selected. By inverting this matrix and multiplying it by both sides of the equation, we obtain a decoding equation, which states that the inverted matrix multiplied by the survivors equals the original data.

In order to decode the system, we notice that each word matches a row of the distribution matrix. When a device fails, we use the k rows of the distribution matrix related to working devices to generate a decoding matrix. It's crucial to remember that this matrix divided by the starting set of information equals the k survivors whose rows we chose. We generate a decoding equation, which asserts that the inverted matrix multiplied by the survivors equals the original data, by inverting this matrix and multiplying it by both sides of the equation. The resulting matrix, which we refer to as a binary distribution matrix, is w(k + m) wk (BDM). We multiply this matrix by a vector of wk elements, each of which contains w bits from a different data device. A w(k + m) element vector with w bits from each data and coding device is the result of this multiplication. Figure 5 serves as an example of this procedure.

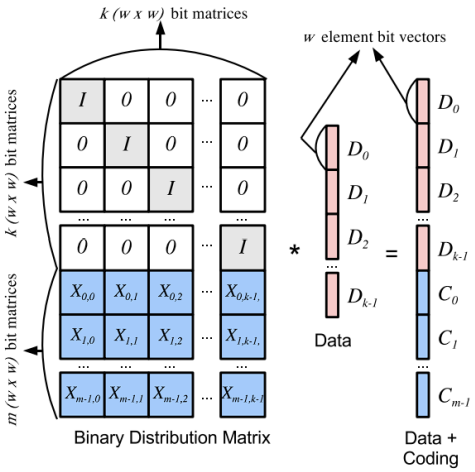

Fig. 5. Bit Matrix. Source : [3]

Each data bit whose corresponding element in the row of the matrix is one can be used to compute the dot product of each row of the product with

the data bits in the BDM. Unlike the dot product using additions and multiplications, this dot product is computed using only XOR operations because all of the elements are bits. Also, the quantity of ones in a row has a direct impact on how well this dot product performs. Thus, using matrices with fewer elements is preferable. Bit-matrix decoding is similar to GF(2w) decoding, with the exception that each device corresponds to w rows of the matrix rather than simply one.

Instead of starting with a typical distribution matrix in GF(2w), which is the conventional method for building bit-matrices, it is also possible to build bit-matrices that are independent of Galois Field arithmetic and nevertheless possess desired coding and decoding properties.

V. Results

For encoding and decoding, the github library [1] is being used and implementation is done in Go. It provides methods, New, encode and decode to create the Encoding matrix and using that matrix to encode and decode the data. All the experiments are performed on Ryzen 5900HS CPU with 8 cores, 16 logical processors and base clock of 3.30 GHz, 16GBs of RAM. Operating system used is Windows 10 Home v22H2. Various file sizes with random data ([A-Z], [a-z], [0-9]) is encoded and decoded with different data blocks and parity blocks.

A. Reed-Solomon Encoding

Encoding Results: Different files are encoded with Reed-Solomon encoding and the results are provided in table I below.

Data Block	Parity Blocks	File Size(MB)	Speed(MBpS)
8	4	10	9633.91
		20	6262.13
		40	7089.43
		60	6158.96
32	4	10	19704.43
		20	39408.87
		40	14495.38
		60	12231.92
32	8	10	19805.90
		20	19359.21
		40	10883.17
		60	5005.63
32	12	10	6387.33
		20	13089.86
		40	8269.59
		60	3991.64

TABLE I
Reed-Solomon Encoding performance with various data blocks and parity blocks.

From the experiments it was observed that on increasing the number of data block size keeping parity blocks constant to 4 and data block constant to 8, the performance increases initially then decreases. It can be seen in the graph 7 and table III below.

Data Block Size(Kbytes)	Time	Speed (MB/S)
125	0.00052	1919.017
625	0.00106	4736.19
1250	0.0016	6255.86
2500	0.00312	6418.28
5000	0.00584	6855.07
7500	0.0740	6114.76

TABLE II
Performance with increasing data blocks and same parity block(4). We observe that encoding speed increases.

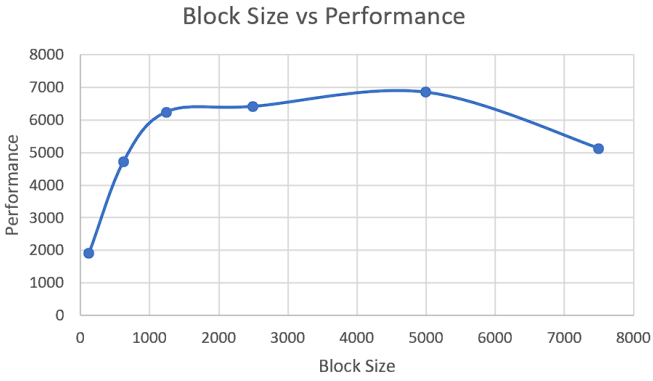

Fig. 6. Data block size vs Performance for Reed-Solomon Encoding.

Next, keeping the data blocks constant to 8 and increasing parity blocks, decreases the performance. This can be observed from the graph below.

Parity Blocks	Time	Speed (MB/S)
4	0.1228	8139.69
5	0.1346	7430.46
6	0.1520	6576.47
7	0.43067	2321.94
8	0.4806	2080.64
9	0.5456	1832.66
10	0.6016	1662.24

TABLE III
Performance with increasing parity blocks and same data block(8). We observe that speed decreases.

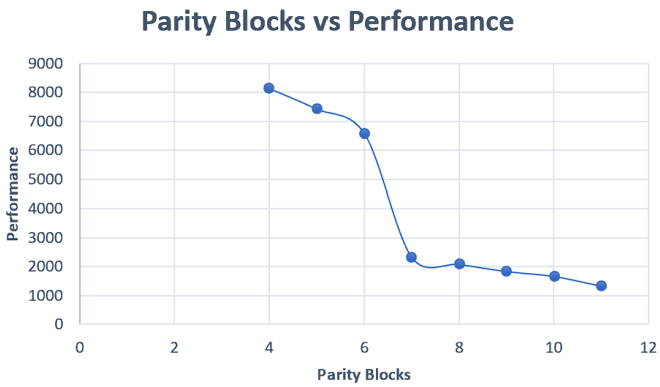

Fig. 7. Parity blocks vs Performance for Reed-Solomon Encoding.

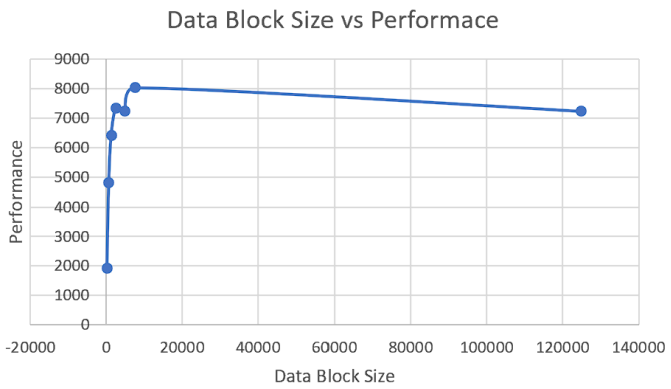

Fig. 8. Data block size vs Performance.

After experiments it was found that for optimal performance data block size should be 5000 kbytes and parity block count should be 4.

B. Cauchy Reed-Solomon Encoding

From the initial experiments, several random files of different sizes are generated, then the Cauchy Reed-Solomon encoding is run on these files. The results are provided in the table below.

Data Block	Parity Blocks	File Size(KB)	Speed(MBpS)
8	4	10	9633.91
		20	9217.01
		40	8182.97
		60	8398.30
32	4	10	19704.43
		20	9606.61
		40	7866.43
		60	14447.04
32	8	10	19805.90
		20	9540.62
		40	4410.53
		60	11422.69
32	12	10	6387.33
		20	7401.38
		40	3556.19
		60	8535.34

TABLE IV
RESULTS FOR DIFFERENT DATA AND PARITY BLOCKS

To check the performance of the Cauchy Reed-Solomon Erasure Code, data blocks, parity blocks and file sizes are varied and their performance is calculated.

From the experiments, it was observed that when we the file sizes are increased by keeping the parity block same, the performance increases for Cauchy Reed-Solomon Encoding as well. The Fig 8 projects the same.

Data Block	Time	Speed (MB/S)
5	0.015619	32.78059
10	0.017302	29.59127
15	0.016613	30.81961
20	0.016335	31.34297
25	0.017584	29.11804
30	0.020537	24.93037
35	0.016985	30.14424

TABLE V
PERFORMANCE WITH INCREASING DATA BLOCKS AND SAME PARITY BLOCK(4). WE OBSERVE THAT SPEED ALMOST REMAINS SAME.

But when we try to increase the number of parity blocks, keeping the Data blocks same, the performance continuously decreases. Fig 11 projects the same.

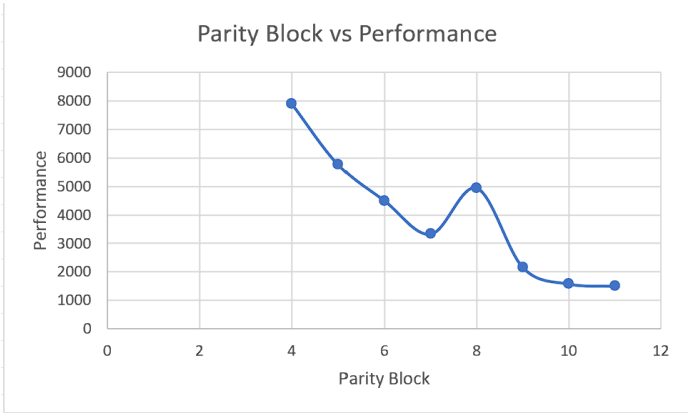

Fig. 9. Parity Block vs Performance.

Parity Block	Time(in Sec)	Speed (MB/S)
4	0.1263	7914.66
5	0.17303	5779.45
6	0.2221	4501.35
7	0.30043	3328.51
8	0.2021	4947.65
9	0.4640	2155.10
10	0.6344	1576.14

TABLE VI
PERFORMANCE WITH INCREASING PARITY BLOCKS AND SAME DATA BLOCKS(32). WE OBSERVE THAT PERFORMANCE DECREASES WITH MORE PARITY BLOCKS.

For optimal performance in Cauchy Reed-Solomon Encoding it was found out that data block size should be 7500 Kbytes and parity block count should be 4.

C. Reed-Solomon Decoding

With the the optimal performance values (block size 5000 KB and parity blocks count 4), the encoded blocks(quantity 2) are randomly selected and deleted, then the reconstruction time is calculated. The results are as follows:

Data Block	Time(in Sec)	Speed (MB/S)
1	0.0010749	930.3190995
1	0.0016071	3111.194076
2	0.0022267	4490.95
4	0.0031978	6254.29
8	0.0037286	10727.89
12	0.0059377	10104.93
200	0.0826032	12106.07

TABLE VII
RECONSTRUCTION SPEED INCREASES WITH DATA BLOCKS INCREASE IN REED-SOLOMON

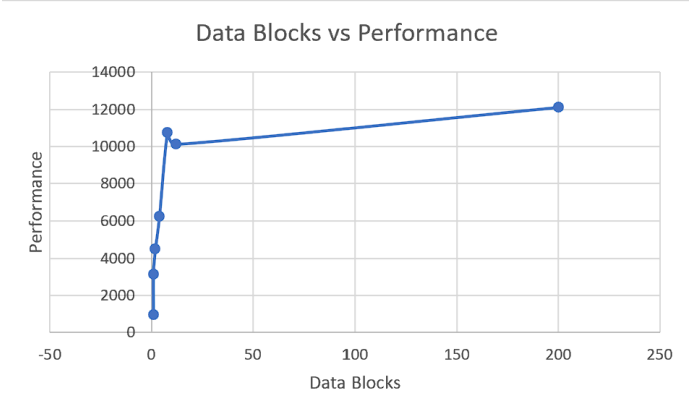

Fig. 10. Data Block vs Reconstruction Performance.

D. Cauchy Reed-Solomon Decoding

With the the optimal performance values (block size 7500 KB and parity blocks count 4), the encoded blocks(quantity 2) are randomly selected and deleted, then the reconstruction time is calculated. The results are as follows:

Data Block	Time(in Sec)	Speed (MB/S)
1	0.0010437	958.1297308
1	0.0010394	4810.467577
2	0.0026576	3762.793498
3	0.0044051	4540.19205
5	0.0041076	9738.046548
8	0.0062515	9597.696553
134	0.0725885	13776.28688

TABLE VIII
RECONSTRUCTION SPEED INCREASES WITH DATA BLOCKS INCREASE IN REED-SOLOMON

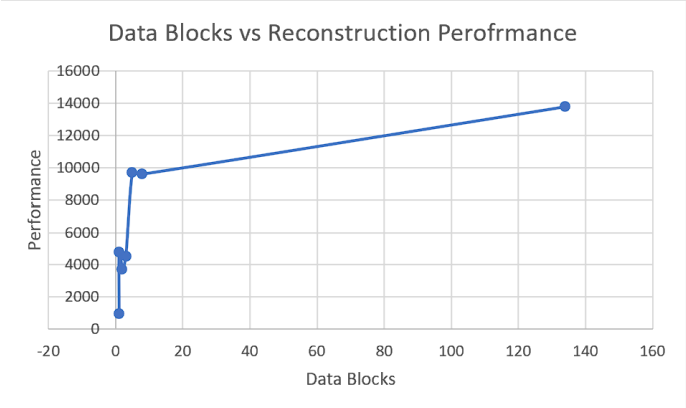

Fig. 11. Data Block vs Reconstruction Performance.

VI. COMPARISON

The results of this paper are compared with [4]. Following are the comparison criteria:

A. On basis of Packet Size

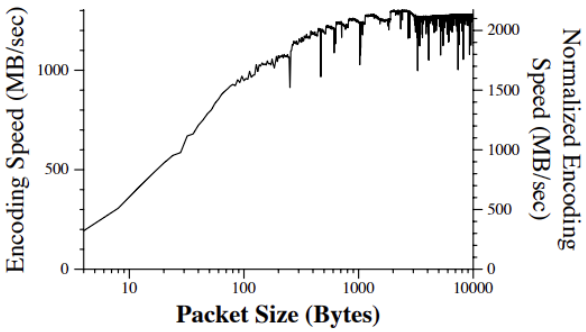

Figure 7: The effect of modifying the packet size on RDP coding, $k = 6$, $m = 2$, $w = 6$ on the Macbook.

Fig. 12. Src : From paper [4].

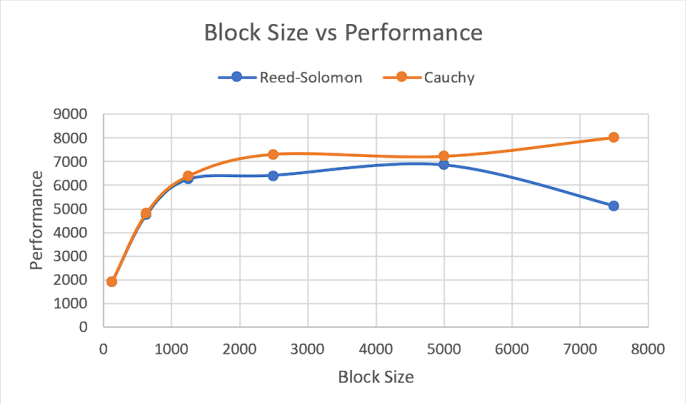

Fig. 13. Packet size comparison between Cauchy and Reed-Solomon. Both follow the same trend.

The author of the paper [4], mentions that the higher packet size performs better than lower packet sizes. This is typically true for both the Reed-Solomon Coding and Cauchy Reed-Solomon Coding. But there is a sweet spot where the L1 cache performs best. Increasing the packet size too much may lead to frequent update of data in cache thus hampering the performance. From figure 13 it can be seen that the trend matches with the figure 12.

B. *On basis of Performance*

The author in paper [4], derives to the conclusion that Cauchy-Reed Solomon Encoding performs better than Reed-Solomon Encoding. This can be seen from the figure 14. Figure 15 represents the performance of Cauchy and Reed-Solomon encoding. The trend follows but the difference between the values seem to greater.

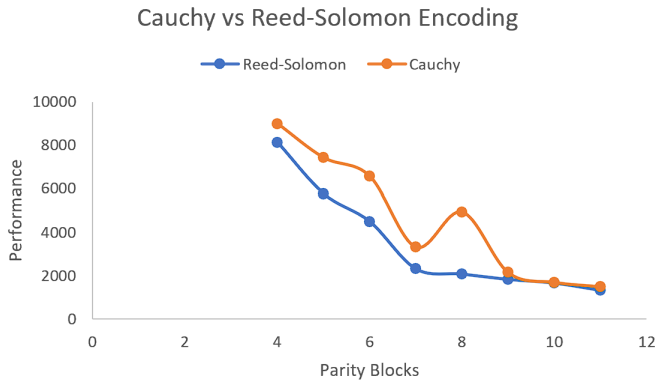

Fig. 14. Comparison of Performance with Parity Blocks

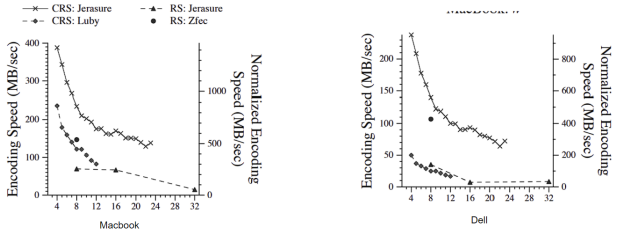

Fig. 15. Src : From paper [4]. Comparison of Reed-Solomon Encoding with Cauchy Reed-Solomon Encoding. The comparison is paper is done for two machines. Left is on Macbook and right is on Dell.

C. *Decoding Speeds Comparison*

Final comparison can be done between the decoding speeds of Cauchy and Reed-Solomon Algorithms. From the figure 16, it can be clearly seen that Cauchy tends to have higher decoding speeds as compared to Reed-Solomon encoding.

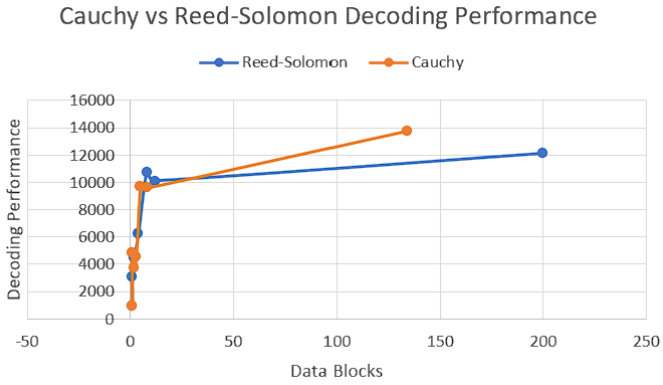

Fig. 16. Decoding speed vs Data Block comparison. Cauchy Reed-Solomon tends to have higher decoding speeds.

VII. CONCLUSION

From the above paper, it can be concluded that the encoding and decoding performance of different algorithms varies greatly with block size, data block length and parity blocks length. When the algorithm is tuned for performance, we observed that Cauchy Reed-Solomon performs better than Reed-Solomon Encoding algorithm. While using erasure-coding algorithms, rather than using default values, the algorithm should be tuned for better performance.

REFERENCES

[1] "GitHub - klauspost/reedsolomon: Reed-Solomon Erasure Coding in Go". In: (). URL: https://github.com/klauspost/reedsolomon.

[2] James S Plank. "Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Storage Applications". In: (2005). URL: http://www.cs.utk.edu/%CB%9Cplank/plank/papers/CS-05-659.html.

[3] James S Plank, Scott Simmerman, and Catherine D Schuman. "Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications Version 2.0". In: (). URL: http://www.cs.utk.edu/~plank/plank/papers/CS-08-627.html.

[4] James S Plank et al. "A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries For Storage". In: ().

[5] "reed-solomon codes". In: (). URL: https://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html.