UNIVERSITY OF MINNESOTA-TWIN CITIES

Close-form and Matrix En/Decoding Evaluation on Different Erasure Codes

by

Zhe Zhang

A thesis submitted in partial fulfillment for the degree of Master of Science

in the

David Lilja
ECE Department

December 2013

Declaration of Authorship

I, Zhe Zhang, declare that this thesis titled, 'Close-form and Matrix En/Decoding Evaluation on Different Erasure Codes' and the work presented in it are my own and collaborated with NetApp Inc. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Abstract

Erasure code, often applied in disk-array distributed file system, is a technique to protect data in case there are some accidental failures in the system. Nowadays, as customers require higher reliability for their data, RAID6, which at most protect double failures in disk-array, is not sufficient any more. Triple failure protection scheme needs to proposed soon. RAID-DP(RDP) is one of most popular erasure codes on RAID6, and with close-form reconstruction, it can achieve optimal computational cost in terms of XORs. However, as the extension of RDP, close-form of RAID-TP(RTP) which provides triple failure protection can not achieve optimal computational cost. There is an alternatative en/decoding method based on matrix operations which seems promissing to replace close-form. Thus, we compared close-form en/decoding with matrix-based method, and we found that matrix en/decoding performs better only when the disk buffer is small. Additionally, calculating decoding matrix cost too much time for reconstruction. Furthermore, we choose several different types of erasure codes in addition to RTP, and evaluate en/decoding performance with matrix method. The results show that RTP needs fewer XORs but it suffers from calculating decoding matrix and updating small writes. Finally, we propose some potential techniques to address these issues that we may want to implement in future.

Acknowledgements

I would like to express the deepest appreciation to my committee chair Professor David Lilja, who has shown the attitude and the substance of a genius: he continually and persuasively conveyed a spirit of adventure in regard to research, and an excitement in discovering unkown issues. Without his supervision and constant help this dissertation would not have been possible.

I would like to thank my committee members. Professor David Du, as the director of CRIS(Center for Research in Intelligent Storage), set up concrete relationship with industries and broaden my view of literature and modern technology both in acedemia and industries. His work persistively invokes me to put more patience and efforts on my research. Professor Arya Mazumdar, whose work demonstrated to me that concern for error control codes not only apply to disk-array, but also to non-volatile devices and distributed file system, should always transcend academia and provide a quest for our times.

In addition, a thank you to Mr. Stan Skelton and Mr. Joe Moor, who introduced me to comprehensive concepts of data storage, where especially erasure codes apply to. Their passion for the innovative research ideas support me to pursue PhD degree in future. I thank the University of Minnesota for consent to include copyrighted pictures as a part of my dissertation. I also want to thank to NetApp Inc. for their financial support granted through my assistantship.

Contents

List of Figures

Chapter 1

Introduction

1.1 What is RAID

In 1987, three University of California, Berkeley, researchers – David Patterson, Garth A. Gibson, and Randy Katz – first defined the term RAID in a paper titled A Case for Redundant Arrays of Inexpensive Disks (RAID). They theorized that spreading data across multiple drives could improve system performance, lower costs and reduce power consumption while avoiding the potential reliability problems inherent in using inexpensive, and less reliable, disks. The paper also described the five original RAID levels.

Originally, the term RAID stood for "redundant array of inexpensive disks," but now it usually refers to a "redundant array of independent disks." While older storage devices used only one disk drive to store data, RAID storage uses multiple disks in order to provide fault tolerance, to improve overall performance, and to increase storage capacity in a system.

With RAID technology, data can be mirrored on one or more other disks in the same array, so that if one disk fails, the data is preserved. Thanks to a technique known as "striping," RAID also offers the option of reading or writing to more than one disk at the same time in order to improve performance. In this arrangement, sequential data is broken into segments which are sent to the various disks in the array, speeding up throughput. Also, because a RAID array uses multiple disks that appear to be a single device, it can often provide more storage capacity than a single disk.

RAID devices use many different architectures, depending on the desired balance between performance and fault tolerance. These architectures are called "levels." Standard RAID levels include the following: Level 0 (striped disk array without fault tolerance), Level 1 (mirroring and duplexing), Level 2 (error-correcting coding), Level 3 (bit-interleaved parity), Level 4 (dedicated parity drive), Level 5 (block interleaved distributed parity), Level 6 (independent data disks with double parity) and Level 10 (a stripe of mirrors). Some devices use more than one level in a hybrid or nested arrangement, and some vendors also offer non-standard proprietary RAID levels.

1.2 Erasure Codes on RAID

1.2.1 Reed-Solomon Code and RAID-DP on RAID

There has been a large amount of interest in recent years in erasure codes. The common class of erasure coding algorithms that people normally associate with the term includes algorithms that add a parameterized amount of computed redundancy to clear text data, such as Reed Solomon coding, and algorithms that scramble all data into a number of different chunks, none of which is clear text. In both cases, all data can be recovered if and only if m out of n of the distributed chunks can be recovered. It is worth noting that, strictly speaking, RAID4,5,6 and RAID-DP [1] are also erasure coding algorithms by definition. However, that is neither here nor there XOR parity-based schemes have different properties than the new algorithms being used in some systems that are what people are thinking of when they talk about erasure codes.

1.2.2 Use Cases for RS and RDP

Erasure codes are being used for deep stores, for distributed data stores and for very scalable stores. They are commonly used in RAIN systems, where the code covers both disk, node and connectivity failures, requiring data reconstruction after a node failure. These erasure codes are more computationally intensive both on encode (write) and decode (reconstruct) than xor parity-based schemes like RAID-DP. In fact, one of the big motivations for developing RAID-DP was that

the industry-standard Reed-Solomon code for dual-parity RAID-6, as well as the less widely used Information Dispersal algorithms to protect against more than one failure, are more computationally intensive than RAID-DP. RAID algorithms are also very suitable for sequential access in memory, as they can work with large word sizes. Many of the complex erasure codes are based on Galois Field arithmetic that works practically only on small (e.g. 4, 8 or 16 bit) quantities, although there are techniques for parallelizing in hardware, on GPUs, or using the SSE* [2] instructions on Intel architecture processors.

1.3 Matrix reconstruction and Closed-form reconstruction

Erasures codes, particularly those protecting against multiple failures in RAID disk arrays, provide a code specific means for reconstruction of lost (erased) data. In the RAID application this is modeled as loss of strips so that reconstruction algorithms are usually optimized to reconstruct entire strips; that is, they apply only to highly correlated sector failures, i.e., sequential sectors on a lost disk.

Matrix method for lost data reconstruction is developed by IBM researcher James Lee Hafner [3], and this method addresses two more general problems: (1) recovery of lost data due to scattered or uncorrelated erasures and (2) recovery of partial (but sequential) data from a single lost disk (in the presence of any number of failures). The latter case may arise in the context of host IO to a partial strip on a lost disk. The methodology we propose for both problems is completely general and can be applied to any erasure code, but is most suitable for XOR-based codes.

Closed form reconstrution is to express the data strips in terms of a finite number of certain polynomial functions. Often lost data can be represented as some variables $x_1, x_2, x_3, ..., x_n$, and problems are said to be tractable if they can be solved in textbfn equations.

The closed-form reconstruction varies from different erasure codes. In other words, every erasure code has its own closed-form reconstruction from mathematics point of view. Consequently, implementors need to put a lot efforts to design dedicated method for each erasure code, even though some type of easure code might have

more practical benefits than others. For example, if you carefully design reconstruction of RDP, most of XOR operations can be well sequentialized in data locations. Therefore, cache miss can be eliminated.

Chapter 2

Background Theory

2.1 Erasure Code with Generator Matrix

Erasure code can be constructed by generator matrix. [4] Figure 2.1 shows an example how this technique works. Assume there is a disk-array which contains 6 data disks and each of them has four symbols on it, and we want to provide double failure recovery to the array. Therefore, we create a generator matrix which includes 8×4 rows and 6×4 columns. First 6×4 rows represent a identity matrix because we want to maintain our data in the coded disk-array. The rest of 2×4 rows corresponds to construct parities. Different erasure codes vary these two group of rows and construct different parities.

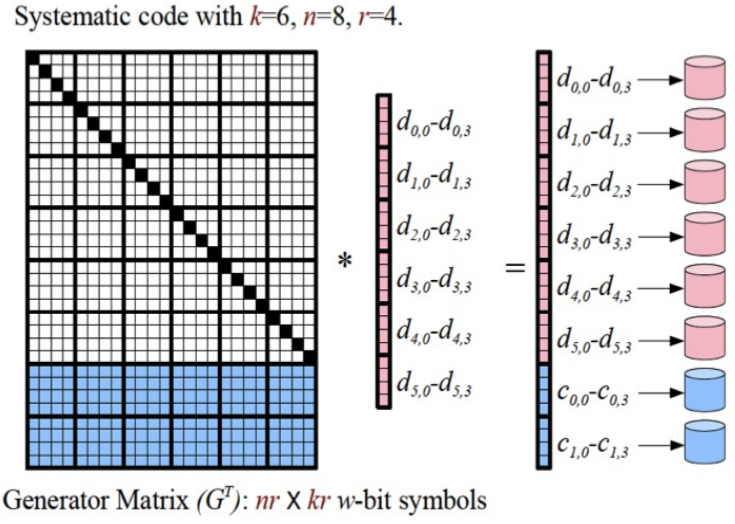

FIGURE 2.1: Generator Matrix on Erasure Code

Figure 2.2 shows how to multiply generator matrix with data disks to get code disks. We take the top symbol in first code disk for example. To construct $C_{0,0}$, we need to multiply first row of generator matrix with all the data symbols in data disks. The multiplication here implies dot product, and the number of columns in generator matrix equals the number of symbols in disk-array.

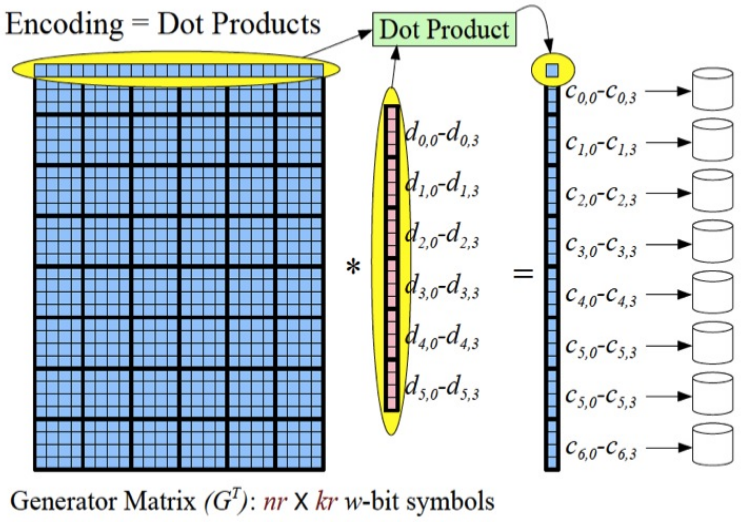

FIGURE 2.2: Multiplying Generator Matrix with Data Disks

If we follow the same procedure for all the rows in generator matrix and multiply with data symbols in array, then we can construct code disks (data+parities).

2.2 Decoding with Generator Matrix

The algorithm originally demonstrated in James Hafner's paper, and we simply the algorithm and briefly illustrate it as follows:

Assume there are two failures occurs as shown in Figure 2.3: the second disk and the fifth disk.

The first of decoding is to remove the rows corresponding to the lost data. In this example, as shown in Figure 2.4, the second and fifth groups of rows are removed/erased out.

As the second step illustrates, we simplify the generator matrix with deteled rows as B-matrix. Consequently, we can represent Figure 2.4 as a multiplication between partial generator matrix and original data, and then we get survivors. Figure 2.5

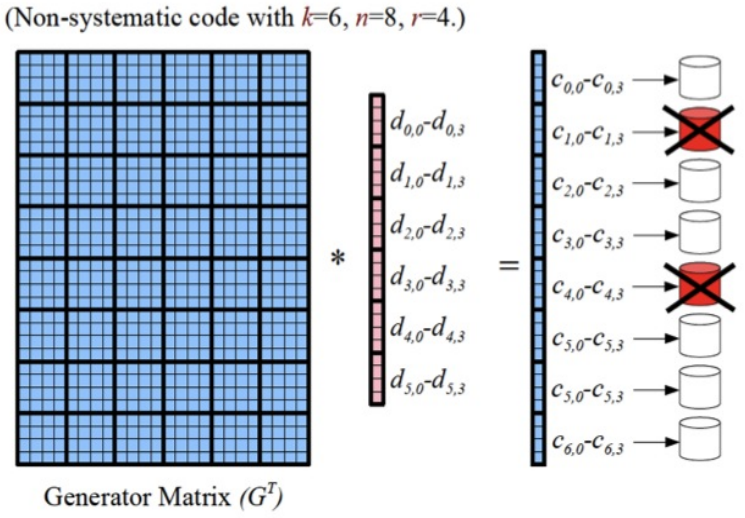

FIGURE 2.3: Two Failures occurs in Disk-array

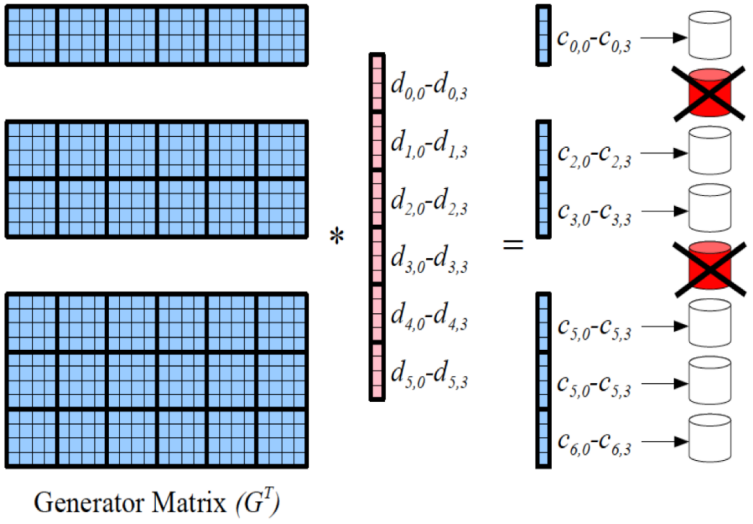

FIGURE 2.4: Removing Rows corresponding to Lost Data

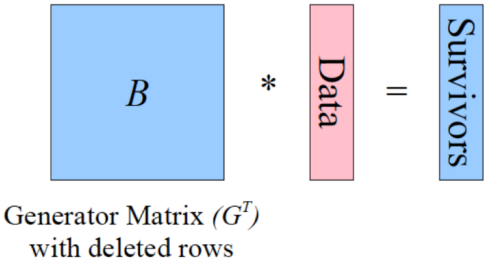

FIGURE 2.5: Lost Data

We multiply B^{-1} which is the inverse matrix of B to both sides of the equation as Figure 2.6 shows. After this simple mathematical step, B^{-1} cancels out with B and therefore we can reconstruct original data. as shown in Figure 2.7

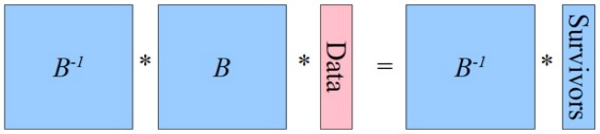

FIGURE 2.6: Multiplying Decoding Matrix to both sides

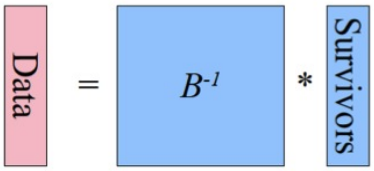

FIGURE 2.7: Reconstruct Data

2.3 RDP/RTP

RAID-DP is developed by NetApp Inc. and it can tolerate at most two failures in disk-array. We can take a look at how to construct RAID-DP from generator matrix. Figure 2.8 shows the steps to construct RDP in disk-array, and Figure 2.9 shows the equivalent XORs between corresponding symbols. We can see that $p_0 - p_3$'s rows of generator matrix are actually representing XORs between symbols in different rwos.

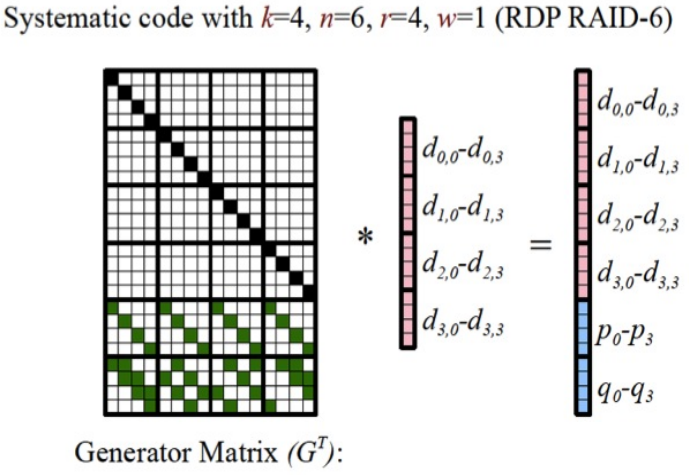

FIGURE 2.8: RDP with Generator Matrix

Figure 2.10 shows how to construct diagonal parity in RDP. Row $q_0 - q_2$ indicates corresponding symbols needed to construct the diagonal parity. For example, q_0 is the XOR result between 7 symbols in data disks marked as blue. 4 symbols

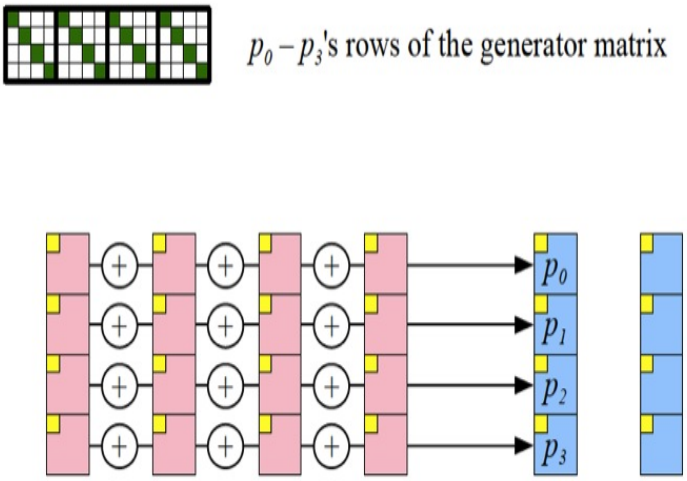

FIGURE 2.9: Row Parity for RDP

out of 7 symbols locates in the second row of data disk and their XOR equals p_1. Consequently, diagonal parity can be constructed by XORing 3 blue diagonal symbols with 1 blue symbol in row parity as shown in Figure 2.11. If we follow the same step for other diagonals, we can construct $q_0 - q_3$ on diagonal parity as shown in Figure 2.12. This is the way how RDP encode the data, and if we follow the similar way to reconstruct the failures without decoding generator matrix, this is called close-form reconstruction. Detailed implementions are illustrated in Chapter 3.

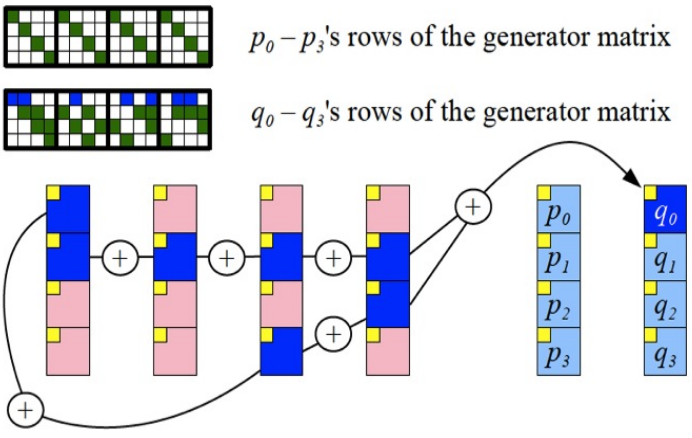

FIGURE 2.10: Diagonal Construction with G-Matrix for RDP

RTP, extended from RDP, provide triple failure recovery [5]. The row parity and diagonal parity are constructed in the same way as RDP, and there is one more anti-diagonal parity in RTP. This parity is constructed by XORing symbols

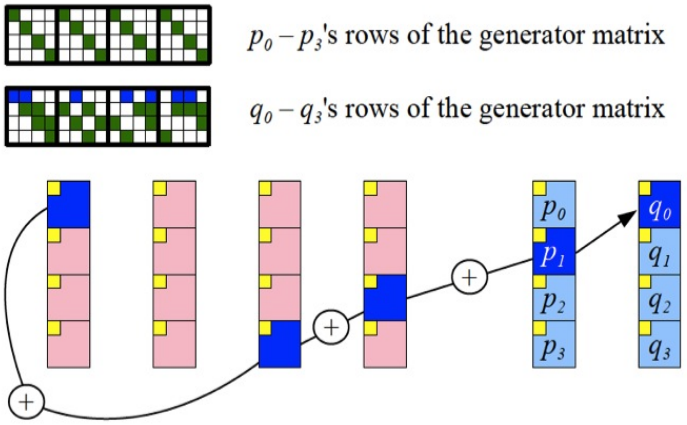

FIGURE 2.11: Diagonal Construction for RDP

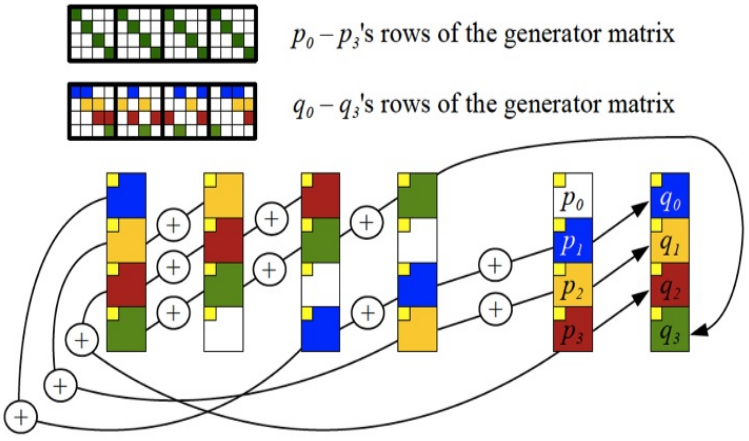

FIGURE 2.12: Diagonal Construction for RDP

in anti-diagonal directions. Detailed encoding and decoding implementations are demonstrated in Chapter 3.

2.4 EVENODD/STAR

EVENODD code is another erausre code which provide double parities and maximumly tolerate two failures in the disk-array. [6] The construction of first parity is XORing all the row symbols, which is the same as RDP. However, the second parity is constructed a bit different from RDP although it is still XORing the symbols in diagonal direction. It does not need to XOR the symbols on row parity to construct the second parity. Instead, it computes sydrome symbol first, and then XORs each diagonal symbols with this sydrome and construct diagonal parity.

STAR [7] is the triple parity version of EVENODD, and the third parity is contructed by XORing symbols in anti-diagonal direction.

2.5 Generalized Blaum-Roth Code

Blaum-Roth code is another erasure code [8], and it can be generalized to tolerate any number of failures. But, different from RDP and EVENODD, the third parity is constructed by XORing symbols on diagonal direction with slope of 2. If you want to construct fourth parity, simply increase the slope and XOR the symbols in diagonal direction with slope of 3.

2.6 Performance Evaluation of Erausre Codes

Erausre codes becomes popular in current academia, and some researchers already evaluate the performance of different types of erasure codes. James Plank evaluated the performance for different types of erasure codes such as Reed-Solomon, Cauchy Reed-Solomon, EVENODD and RDP, all of which provide double parities and maximumly support double failure reconstruction. Figure 2.13, 2.14 and 2.15 show the performance evaluation [9], and we can see XOR-based erasure codes such as EVENODD, RDP have better performance than RS-based code. Additionally, Cauchy RS code seems better than normal RS code, and their encoding and decoding speed drops faster than XOR-based codes as strip depth becomes larger. This is due to computation cost in Garlois-Field, and as the dimension of GF becomes larger, computations takes more time.

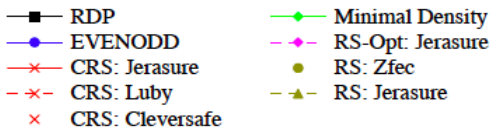

FIGURE 2.13: Erasure Codes

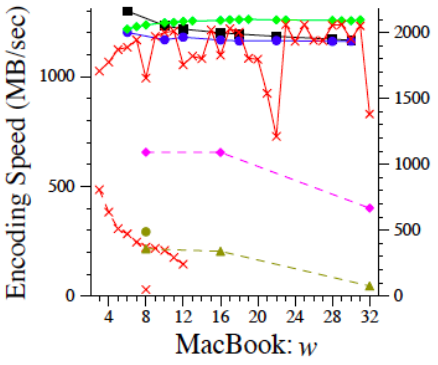

FIGURE 2.14: Encoding

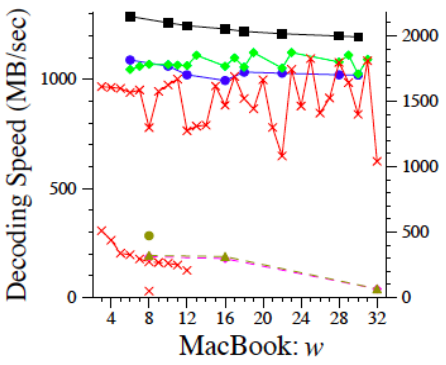

FIGURE 2.15: Decoding

Chapter 3

Implementation

3.1 Jerasure Simulator

Jerasure is developed by Dr.James Plank, and it can evaluate the performance of different types of erausre codes such as Reed-Solomon, Cauchy Reed-Solomon and Blaum-Roth [10]. It compares the coding and decoding performance. You could get the results of the memory bandwidth of codec process, and coding and decoding matrix at any point in the procedure. This is the good simulator tool to evaluate the accuracy as well as the codec complexity and memory bandwdith cost of different erausre codes.

3.2 Closed-from RDP/RTP Construction

To simulate the construction of RTP, we store data in the memory and group certain size of data in a buffer. All the operations are based on those data buffer and they have the same size.

To calculate the first parity, simply XOR all the data buffer in a row direction. The operation, shown in Figure 3.1, is simple for this operation, we XOR the first two memory buffers together and store the partial result in a tempory buffer. Similarly, we XOR this temporay buffer with left buffers each a time, and replace the tempory buffer with new partial result each time. Finally, all the buffer are XORed and the tempory buffer obtain the final result. The benefit of this trick is

to save immediate memory cost, and the tempory buffer alone might restricted in the lower level cache, which could accelerate the wright process.

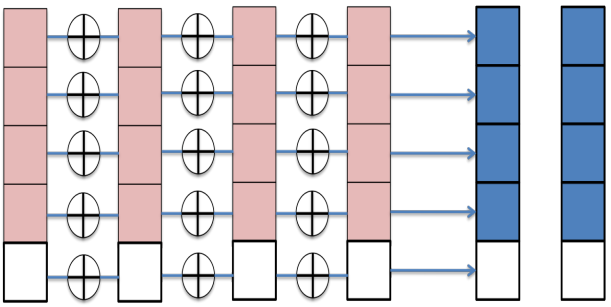

FIGURE 3.1: Row Parity Construction

There is a dummy symbol added at the end of each buffer, and they are treated as 0 when the buffers run XORs. These dummy symbols can help buffers align with each other and simply the encoding and decoding process.

For the diagonal parity, we apply the same trick create a tempory buffer for diagonal parity. Additionally, we have to create a function to manipulate shifting operation. As we see in the construction algorithm of RDP/RTP, the symbols XORed in adjacent buffers are shifted by one position. This step is shown in Figure 3.2. We do not need to load every symbol from different locations of buffers. After shifting adjacent buffer down by one symbol, we can wrap around the symbols, as shown in Figure 3.3, and move back outbound symbols back to the top of the buffers. In this way, we can still XOR data buffers instead of dispersed symbols each a time. The results are eventually stored in diagonal buffer.

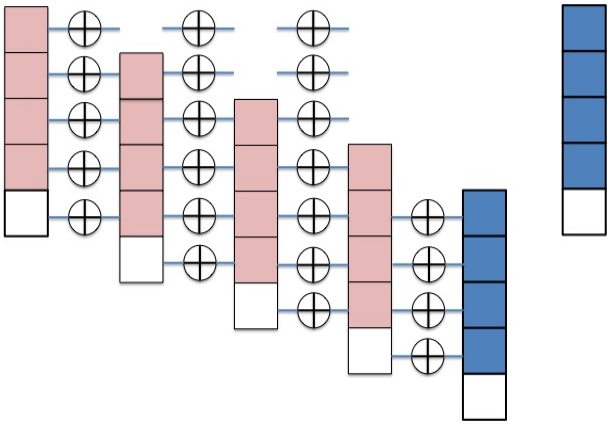

FIGURE 3.2: Shift Symbol

Anti-diagonal in RTP is constructed in the same way as diagonal buffer. The only difference is that we need to shift buffer up by one symbol each time and wrap around the outside symbols back to the end of the buffers.

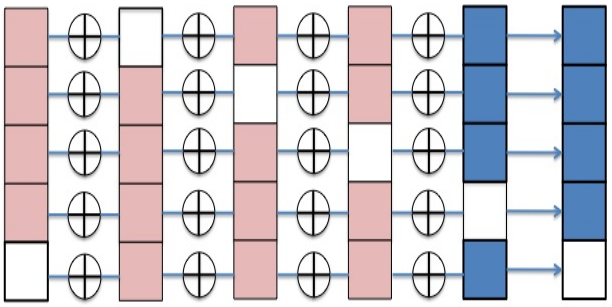

FIGURE 3.3: Wrap around Symbols

3.3 RDP Reconstruction

Reconstruction works with a specific algorithm. We start with RDP reconstruction. Figure 3.4 shows an example of the diagonal construction of RDP. Disk pool contains four stripes in a 6 disk RDP array ($p = 5$). The number in each block indicates the diagonal parity set the block belongs to. Each row parity block contains the even parity of the data blocks in that row, not including the diagonal parity block. Each diagonal parity block contains the even parity of the data and row parity blocks in the same diagonal. Note that there are $p = 5$ diagonals, but that we only store the parity of $p - 1 = 4$ of the diagonals. The selection of which diagonals to store parity fo is completely arbitrary. We refer to the diagonal for which we do not store parity as the "missing" diagonal.

Assume that data disks 1 and 3 have failed in the array of Figure 3.4. It is necessary to reconstruct from the remaining data and parity disks. Clearly, row parity is useless in the first step, since we have lost two members of each row parity set. However, since each diagonal misses one disk, and all diagonals miss a different disk, then there are two diagonal parity sets that are only missing one block. At least one of these two diagonal parity sets has a stored parity block. In our example, we are missing only one block from each of the diagonal parity sets 0 and 2. This allows us to reconstruct those two missing blocks.

Having reconstructed those blocks, we can now use row parity to reconstruct two more missing blocks in the two rows where we reconstructed the two diagonal blocks: the block in diagonal 4 in data disk 3 and the block in diagonal 3 in data disk 1. Those blocks in turn are on two other diagonals: diagonals 4 and 3. We cannot use diagonal 4 for reconstruction, since we did not compute or store parity for diagonal 4. However, using diagonal 3, we can reconstruct the block in diagonal 3 in data disk 3. The next step is to reconstruct the block in diagonal 1 in data

Data Disk 0	Data Disk 1	Data Disk 2	Data Disk 3	Row Parity	Diag. Parity
0	1	2	3	4	0
1	2	3	4	0	1
2	3	4	0	1	2
3	4	0	1	2	3

FIGURE 3.4: RDP Example of Diagonal Parity

disk 1 using row parity, then the block in diagonal 1 in data disk 3, then finally the block in diagonal 4 in data disk 1, using row parity.

The important observation is that even though we did not compute parity for diagonal 4, we did not require the parity of diagonal 4 to complete the reconstruction of all the missing blocks. This turns out to be true for all pairs of failed disks: we never need to use the parity of the missing diagonal to complete reconstruction. Therefore, we can safely ignore one diagonal during parity construction.

3.4 RTP Reconstruction

3.4.1 Single and Double Disk Reconstruction

Recovery from single disk failures can be accomplished either by using row parity or by computing the diagonal or anti-diagonal parity disk. Since RTP extends RDP, double disk reconstruction can be performed by using the RDP reconstruction algorithm. This is because the two failed disks belong to at least one of the diagonal or anti-diagonal RDP parity sets. If both diagonal and anti-diagonal parity disks fail, they can be independently reconstructed from the data and horizontal parity drives, using the RTP parity construction algorithm.

3.4.2 Triple Disk Reconstruction

Triple disk failure cases in an RTP array can be classified into three categories, depending upon the number of data and parity disks failed. For simplicity and ease of understanding, the row parity disk and data disks are collectively referred

to as RAID 4 disks, since they are symmetric with respect to the diagonal and anti-diagonal parity sets.

- One of RAID 4, diagonal and anti-diagonal disk failed: This case is trivial since the missing RAID 4 disk can be recovered using row parity. The RTP parity computation algorithm can then be applied to recover the missing diagonal and anti-diagonal parity disks.
- Two RAID 4 and one diagonal (or anti-diagonal) disks failed: For this case, reconstruction proceeds by first applying the RDP double reconstruction algorithm using the good anti-diagonal (or diagonal) parity disk. After the failed RAID 4 disks are recovered, the missing diagonal (or anti-diagonal) can be recovered using the RTP parity computation algorithm.
- Three RAID 4 disks failed: The primary step in the process of reconstructing three RAID 4 disks involves computing p 4-tuple XOR sums on one of the missing disks. Each 4-tuple sum (equation 7, section 3.2.2) is computed as the XOR sum of 4 blocks on one of the failed disks. The set of linear equations corresponding to these sums can then be solved in various ways to recover that missing disk. Subsequently, the remaining two disks can then be recovered using the RDP double reconstruction algorithm.

For this process to work, parity for all diagonal and anti-diagonals must be available. Hence, the triple disk reconstruction steps can be broadly classified as:

1. Compute the dropped parity block for both the diagonal and anti-diagonal parity sets.
2. Compute a set of 4-tuple XOR sums on one of the failed disks
3. Recover one of the failed disks.
4. Use RDP double reconstruction to recover the remaining two disks

Compute Diagonal And Anti-Diagonal Parity In a set of $p1$ stripes forming a complete row, diagonal and anti-diagonal parity set, the parity for the missing/dropped diagonal in an RTP (or even RDP) array can be computed as the XOR sum of

the $p1$ blocks on the diagonal parity disk. Similarly, the missing anti-diagonal can be computed as the XOR sum of blocks on the anti-diagonal disk.

Proof. Since each diagonal spans both data and row parity disks, and one diagonal is dropped owing to insufficient space, the XOR sum of the blocks stored on the diagonal parity disk can be computed as:

$$\sum Diag[..] = (\sum A[..] + \sum A_{droppeddiagonal} + \sum R[..] + \sum R_{droppeddiagonal})$$

Substituting $\sum R[..]$ by $\sum A[..]$ since blocks on the row parity disk are themselves the XOR sums of data blocks, we get

$$\sum Diag[..] = (\sum A_{droppeddiagonal} + \sum R_{droppeddiagonal}) = \sum AR_{droppeddiag}$$

Another way to think of this is that the sum of all the RAID 4 blocks equals the sum of all the diagonal parity blocks including the dropped diagonal parity block. Therefore, subtracting the sum of the stored diagonal parity blocks from the sum of all RAID 4 blocks gives the sum of the RAID 4 blocks on the dropped diagonal. Similarly,

$$\sum Anti - diag[..] = \sum AR_{droppedanti-diag}$$

After computing the dropped diagonal and anti-diagonal parity, parity for all diagonals and anti-diagonals is available. For each row, diagonal and anti-diagonal, the XOR sum of missing blocks on the three failed disks can now be computed by summing the surviving blocks on the corresponding row, diagonal and anti-diagonal respectively. Let $XOR_r(k)$, $XOR_d(d)$, and $XOR_a(a)$ represent these values corresponding to row k, diagonal d, and anti-diagonal a respectively. Since with even parity the sum of all blocks in one parity stripe or diagonal is zero, then the sum of any subset of items in the set must equal the sum of the remaining items not in the subset. The sum of the surviving items in a parity set is often called a parity syndrome in the literature.

Compute 4-tuple sums on one of the failed disks

We can take a look at an example in Figure 3.5. This two tables show how RTP constructs diagonal parity and anti-diagonal parity. The remaining sets of steps are described in the context of this 9-disk pool.

Figure 1: Diagonal Parity Sets

D0	D1	D2	D3	D4	D5	R	Diag	A-Diag
0	1	2	3	4	5	6	0	
1	2	3	4	5	6	0	1	
2	3	4	5	6	0	1	2	
3	4	5	6	0	1	2	3	
4	5	6	0	1	2	3	4	
5	6	0	1	2	3	4	5	

Figure 2: Anti-diagonal Parity Sets

D0	D1	D2	D3	D4	D5	R	Diag	A-Diag
6	0	1	2	3	4	5		6
5	6	0	1	2	3	4		5
4	5	6	0	1	2	3		4
3	4	5	6	0	1	2		3
2	3	4	5	6	0	1		2
1	2	3	4	5	6	0		1

FIGURE 3.5: RTP Example of Diagonal and Anti-diagonal Parities

Figure 3.6 and Figure 3.7 represents and equivalent diagonal and anti-diagonal layout. The seventh row contains dummy symbols as we mentioned before, and these will simplify the reconstruction. We assume disks $X = 0$, $Y = 1$ and $Z = 4$, represent the 3 failed drives within the array. As RTP algorithm demonstrateds [5], To reconstruct these 4-tuple sums, we need to a crossing which diagonal and anti-diagonal parities overlap their corner symbols with two row parities. In Figure 3.6 and Figure 3.7, if we XOR all the survived symbols on 4-th diagonal with th 4-th symbol on diagonal parity, we can get the XOR result of 4-th diagonal symbols on failed disks. The same situation applies in anti-diagonal disk. 6-th anti-diagonal marked in Figure 3.7 overlapped with 4-th diagonal in row 0 and row 4. Therefore, if we XOR all the 1-st and 4-th symbols on survived disks with the corresponding symbols in row parity, we can get XOR result of 1-st row and 4-th row in failed disks. If we XOR 1-st and 4-th rows of three lost disks with 4-th diagonal symbols of lost disks as well 6-th anti-diagonal symbols of lost disks together. There are four symbols in disk X and disk Z cancelled with each other, and the result only contains four symbols [0,1,3,4] on the middle disk Y.

As illustrated above, this only demonstrates one crossing, which spreads between 1-st row and 4-th row. Similarly, we can find a crossing between 2-nd row and 5-th row, then we can get XOR between another four symbols [1,2,4,5]. If we iterates all the crossings in disk array, we can get all 4-tuple sums as shown in Figure 3.8.

X=0	Y=1	d2	d3	Z=4	d5	R
0	1Φ	2	3	4	5	6
1	2Φ	3	4	5	6	0
2	3	4	5	6	0	1
3	4Φ	5	6	0	1	2
4	5Φ	6	0	1	2	3
5	6	0	1	2	3	4
6	0	1	2	3	4	5

FIGURE 3.6: 4-Tunple Sum on Diagonal Parity

X=0	Y=1	d2	d3	Z=4	d5	R
6	0Φ	1	2	3	4	5
5	6Φ	0	1	2	3	4
4	5	6	0	1	2	3
3	4Φ	5	6	0	1	2
2	3Φ	4	5	6	0	1
1	2	3	4	5	6	0
0	1	2	3	4	5	6

FIGURE 3.7: 4-Tunple Sum on Anti-diagonal Parity

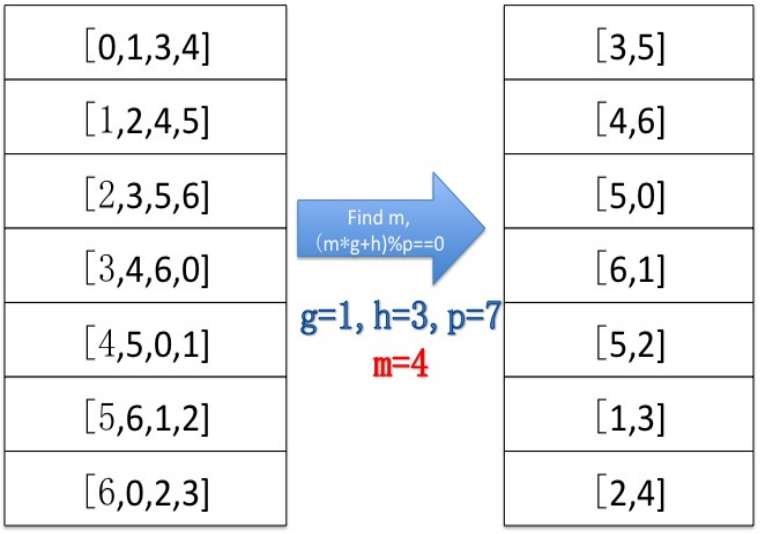

FIGURE 3.8: 4-Tunple Sum to Pair Sum

Reduce to pairwise sums

The main idea behind 4-tuple sums is that the collection of 4-tuple sums represents a set of p linear equations over p variables. Each variable represents a block on the middle disk and contributes to exactly four equations. One of these variables, the

block on the imaginary p_th row, is known to have the value zero. This set of linear equations can be solved in various ways to recover the middle disk. To illustrate the process of recovering the middle disk, this section describes one approach where a subset of 4-tuples is selected and reduced to a pairwise sum. Repeating this process $p1$ times by starting with a different tuple helps generate $p1$ pairwise sums which are separated by a constant distance. At least one such pair, however, has only one unknown block and hence can be recovered. The set of other pairwise sums can then be used to recover the remaining blocks on the middle disk. This is the reason why this is called close-form reconstruction because you can always solve p variables with p linear equations.

As the algorithm shown in RTP paper, with the distances between three failures (which is g=1, h=3 in our example) we can always find an m, and choose a offset which results in cancelling common blocks in 4-tuple sums. For example, in Figure 3.8, we can find $m = 4$ and apply this offset to every block in 4-tuple sums. Let's take block 0 and block 4 in 4-tuple sums in Figure 3.8, there are three common symbols 0,1 and 4 within [0,1,3,4] and [4,5,0,1]. Thus, we can get pairwise sum between symbol 3 and 5 if we XORs block 0 and block 4 together. Consequently, we can get pairwise sums with different symbols combined together as shown in Figure 3.8. Finally, Figure 3.9 shows 7 equations which totally contain 7 symbols, and note that symbol 6 in the middle disk is the dummy symbol and it is 0. With these equations, we can reconstruct the middle disk now.

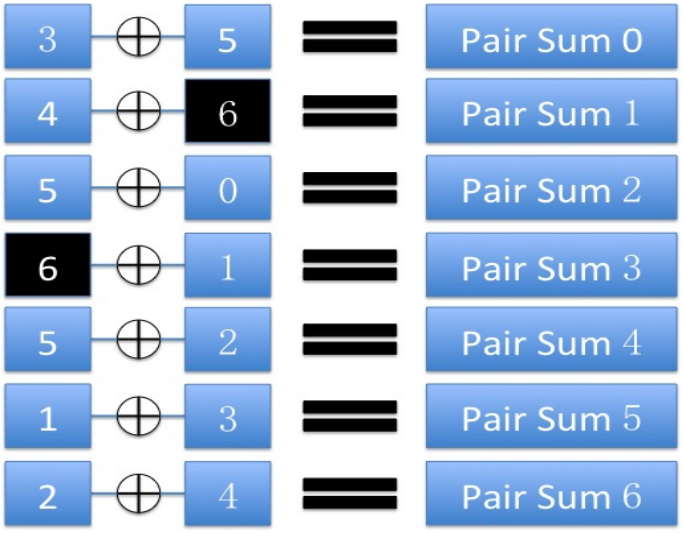

FIGURE 3.9: Reconstruct disk by pairwise sums

Use RDP double reconstruction to recover remaining two disks Having recovered the middle disk, RDP double reconstruction can be used to recover the remaining two disks, thus completing the process of reconstructing three RAID 4 disks.

3.5 Matrix Reconstruction

3.5.1 Generator Matrix

Jerausre has already implemented the matrix reconstruction algorithm. Given a generator matrix, the symbols corresponding to 1's in G-matrix are multiplied with each other. G-matrix multiplies data buffers and stores three parities in three dedicated buffers.

The input of Jerasure is G-matrix. Different erasure codes apply different G-matrices. Therefore, we need to construct G-matrix for RTP based on the RTP algorithm. Figure 3.10 shows the generator matrix of RTP.

FIGURE 3.10: Generator Matrix of RTP

Original Jerasure can only create generator matricies for double parity. It creates the two groups of rows, and each group is responsible to generate one parity. To construct generator matrix, in liberation.c file, there is a function called "*blaum_roth_coding_matrix*". It creates first two groups of rows and accords to the row parity and Blaum-Roth diagonal parity. In order to create our own erasure codes, we need to modify this part based on different algorithms. If we want to construct the triple parities, we can add one more group of rows in the generator matrix. Figure 3.11 shows the generator matrix for triple parity based on Blaum-Roth algorithm. Figure 3.12 shows the generator matrix for STAR.

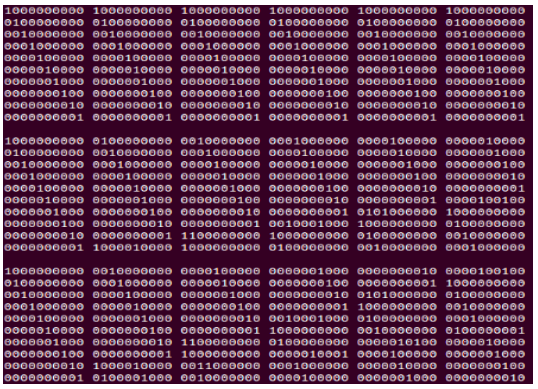

FIGURE 3.11: Generator Matrix of Triple Blaum-Roth

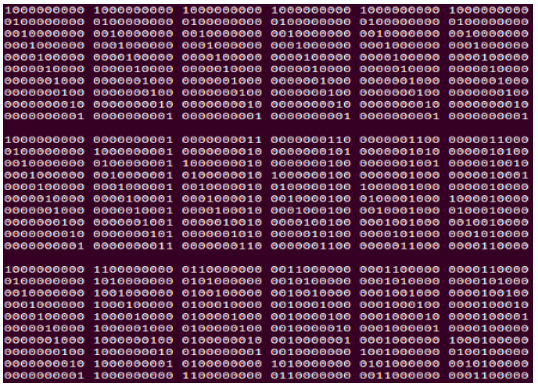

FIGURE 3.12: Generator Matrix of STAR

3.5.2 Reconstruction Matrix

To calculate reconstruct matrix, Jerasure follows the algorithm presented in James Hafner's paper. Basically, G-matrix removes the columns locally corresponds to erased disks. By pseudo-inverse algorithm, this partial G-matrix is transformed into reconstruct matrix.

Chapter 4

Experiment Result

4.1 Reed-Solomon Code and RDP/RTP

Matrix reconstruction is suitable for XOR-based erasure code such as EVENODD, RAID-DP and Blaum-Roth because matrix only contains 0's and 1's, and the row and column computations are simple. Reed-Solomon Code, on the other hand, needs to rebuild the decoding matrix in Garlois field. It has apply some special calculations such as multiplying a scalar in Garlois field, which needs much more efforts in normal CPU than XOR-calculation. Therefore, the memory throughtput suffers a lot when we implement Reed-Solomon Code in disk-array. Figure 4.1 demonstrates the memory throughtputs of Reed-Solomon Code and RDP/RTP. We simulate 16 data disks and each of disks contains 16 symobls and each symbol is 1KB. We compare Reed-Solomon Code with 2 parities to RDP and 3 parities with RTP, and both encoding and decoding are measured.

The reconstruction in Jerasure includes two steps. According to failure locations, we need to first calculate pseudo-inverse matrix of partial coding matrix, and get the decoding matrix. The second step is to multiply decoding matrix with survived data and parities, and this step will reconstruct the failures. There are two observations from Figure 4.1. First observation, in RDP/RTP, is the huge gap between encoding and decoding. This gap implies reconstruction procedure spends most of the time to calculate the decoding matrix and this is the bottleneck. This gap is not extremely obvious for Reed-Solomon Code, although the decoding matrix still occupies some portion of reconstruction process. Additionally, we also observe that memory throughput of Reed-Solomon code is far less than RDP/RTP.

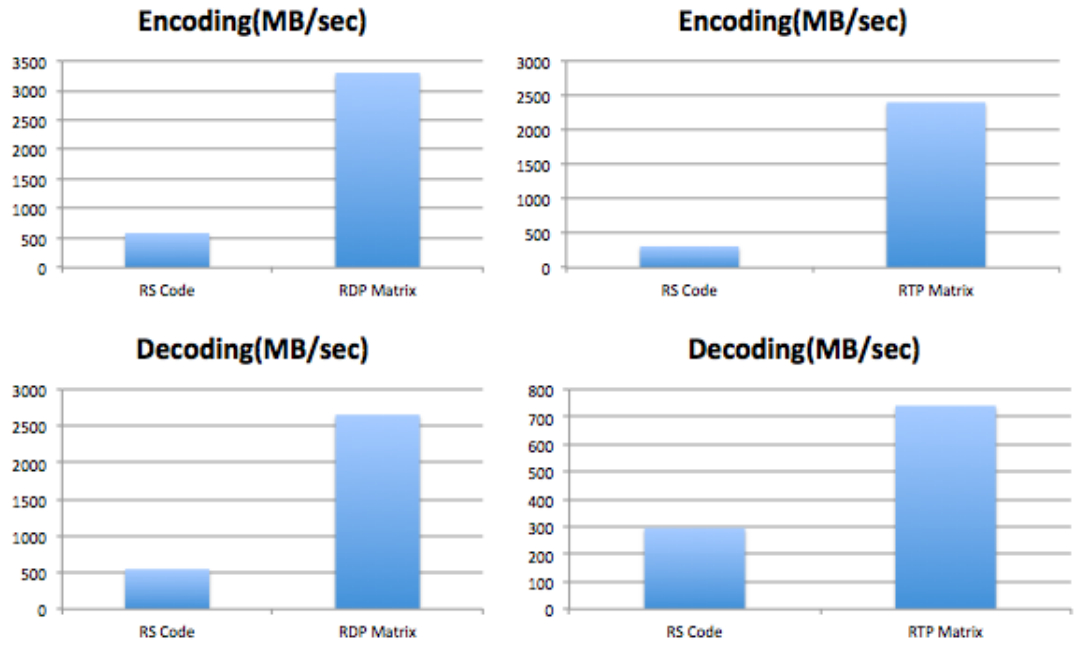

FIGURE 4.1: Reed-Solomon Code and RDP/RTP

This justifies our hypothesis that the computation in Garlois field takes much more time than simple XOR operations. Even though the computation load is very heavy for decoding matrix, sophiscated Garlois field calculation reduce this gap between coding and decoding. This tell us Reed-Solomon code is not suitable for normal CPU operation, and matrix reconstruction method is beneficial to XOR-based erasure codes.

These observation does not imply Reed-Solomon Code is not useful in disk-array. Nowadays, Reed-Solomon is still one of the most popular erasure code in storage world. One reason is the controller leverages special instruction set for Reed-Solomon Code. As some of current research is doing, accelerated with Intel SIMD instructions, operations on Garlois field are extremely fast, and the memory bandwidth can achieve 4 GB/s according to technical papers.

4.2 Close-Form v.s Matrix

Close-Form construction and reconstruction has a very good performance for cache by carefully implememting the RTP algorithm. Since we can shift the data buffer by a symbol each time and XOR whole data buffer with each other, the memory can achieve the relevantly high throughtput. But the downside of close-form

is the complexity of decoding procedure and extra memory space to store immediate data. Rather than close-form, matrix method only contains two steps to reconstruct the failures. First, partial generator matrix has to transform to pseudo-inverse matrix and get the decoding matrx. Once the decoding matrix is ready, each symbol in failure buffers will seach and multiply data symbols from corresponding matrix rows to reconstruct the symbols sequentially. The disadvantage of matrix method is the extra XORs needed than close-form reconstrution. Because some of survived data and parity symbols usually touch multiple failure symbols, which have to be XORed multiple times. This fact can cause some side effect. For example, cache could be polluted by loading scatter symbols from disk pool. Current research address this issue by optimize the sequence of XORs and eliminate the cache pollution. But this still has a performance gap with close-form reconstruction.

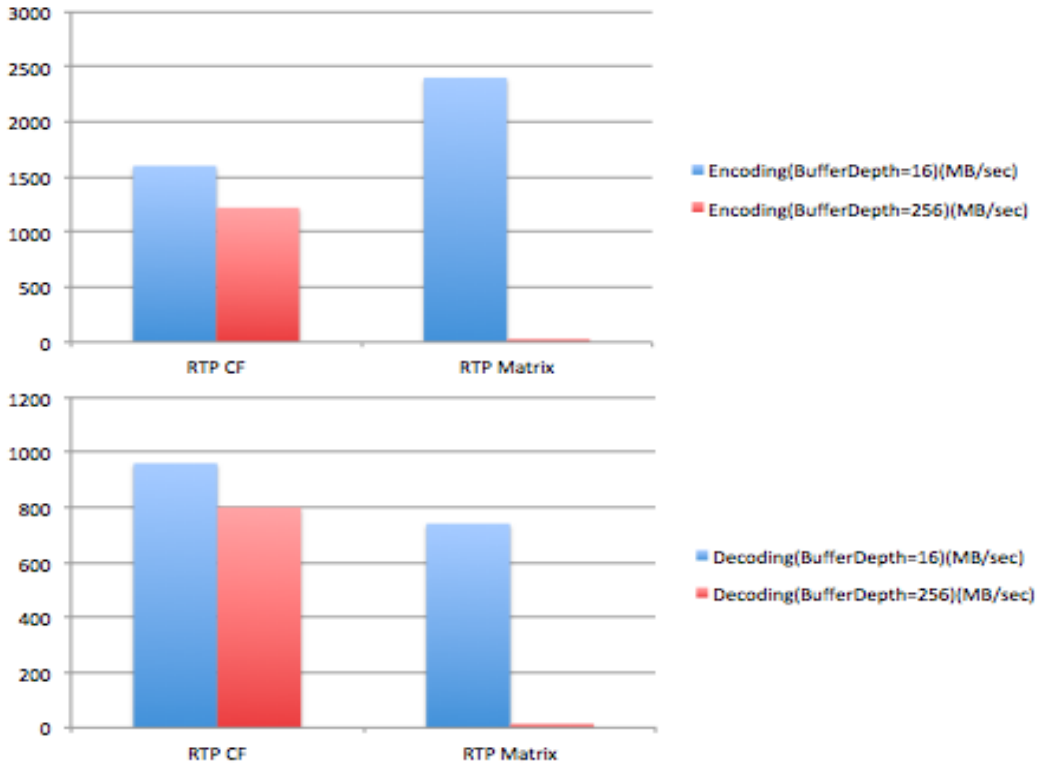

FIGURE 4.2: Close-Form v.s Matrix Method

First observation from Figure 4.2 is that RTP matrix encoding process has better performance than RTP close-form if the buffer size is small. This is because small buffer size allow the disk pool fits into the cache, and even though symbols are loaded in a random way, the memory throughput will not suffer too much.

However, as the buffer depth increases to 256 symbols, the encoding performance dramatically dropped to 30 MB/s. Large buffer size causes a lot of cache misses when data are loaded randomly, and this is only the case for matrix method. Close-form encoding, on the other hand, still loads data in a sequential way and temporarily XOR each buffer and through it out of cache. This will avoid polluting the cache, because if we consistantly load same data multiple times. Additionally, for small buffer depth, matrix encoding perform better than close-form method. This depends on the implementation in which there are some tempory buffer stores the immediate XOR results, and finally store tempory buffer back to parity buffer. These operations increase the overhead.

Let's take a look at decoding process. As we mentioned in encoding, larger data depth causes cache pollution so that the memory performance dramatically reduces. Another interesting observation is that matrix method takes more time than close-form method. The throughput for close-form decoding is 960 MB/s, and matrix reconstruction only has 770 MB/s. This is due to the time cost for calculating decoding matrix.

If we compare decoding with encoding, the decoding performance of matrix method drops a lot comparing to encoding. This implies that calculating decoding matrix is the bottleneck. Additionally, because of more complex steps in decoding process, close-form performance also reduces.

4.2.1 Decoding Matrix Calculation

Let's further look at decoding matrix calculation. To better analyze this process, we run a large disk pool size which corresponds to current industrial setup. There are 28 disks store data and 3 extra disks store parities, each disk contains 256 symbols and each symbol is 4 KB data chunk. We run the program which only calculates decoding matrix and statistical results are shown in Figure 4.3. The time cost to calculate decoding matrix highly dependes on the distances between failed disks. The minimum time cost comes from the situation in which three parity disks lost. In this case, decoding matrix is exactly the same as encoding matrix. The largest time cost is the situation that the failures span across the disk-array and three failures have unequal distances between each other. For most of failure cases, the system still needs around 30 seconds to calculate decoding matrix and these are definitely not tolerable by customers. We need to mention

that smaller disk pool requires shorter time to calculate decoding matrix, and the disk-array configurations may vary from different companies and applications.

Max Time (sec)	Min Time (sec)	Ave Time (sec)
137.0666	0.0234	25.4886

FIGURE 4.3: Time Cost of Decoding Matrix

To overcome high time cost for decoding matrix, we need to come up with some methods either to accelerate the calculation speed or to pre-store the decoding matrices somewhere and load the matrix directory into memory when it is needed.

4.3 Erausre Codes Complexity Comparison

Jerasure does not include EVENODD and RAID-DP codes in the open source code because of the patent issue. We implemented EVENODD and RAID-DP codes and extended EVENODD, RAID-DP to triple parity codes which are called STAR and RAID-TP. STAR is design by Microsoft and RAID-TP is owned by NetApp. There is a file called liberation.c, and at the bottom of it, James Plank developed the generator matrix of Blaum-Roth code with two parities. I change the matrix layout of the second parity, and achieve the EVENODD code and RAID-DP. Furthermore, I append the coding matrix for the third parity. Blaum-Roth algorithm calculates the third parity in the diagonal way by the slope of two. EVENODD and RAID-DP, on the other hand, calculate the third parity in the anti-diagnal way, which could be considered as the slope of negative one. We measure the performance for different erasure codes, there are three metrics, number of XORs, memory bandwidth and number of IOs. We will discuss different disk pool operations with these metrics.

4.3.1 One Disk Failure

This figure shows how many XOR it needs to reconstruct the one disk failure. It shows that there is no difference between these three erausre codes. The reason is that all of these three codes will use the row-xored parity disk to reconstruct the

failure, and therefore, they follow the same recontruction procedure. Figure 4.4 show the results.

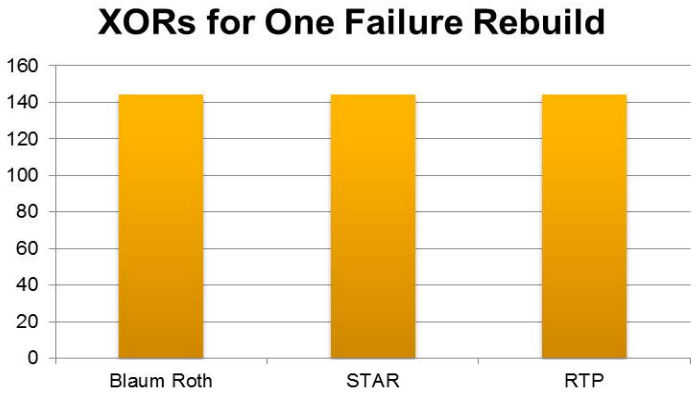

FIGURE 4.4: One Failure Reconstruction

4.3.2 Two Disk Failures

Figure 4.5 shows the number of XOR needs to be used to recontruct two disk failures. We could see that STAR cost most and RTP costs slightly less than Blaum-Roth.

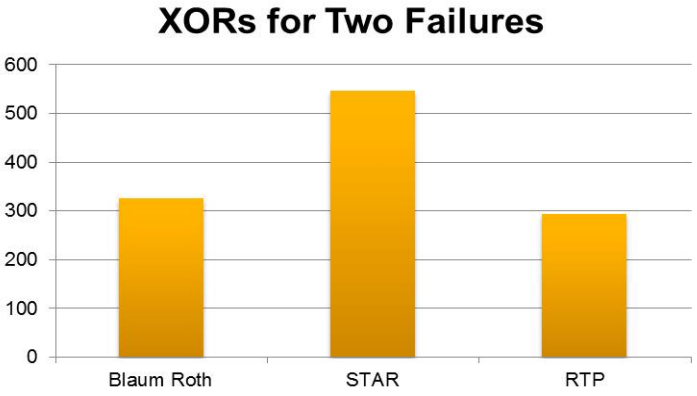

FIGURE 4.5: Two Failure Reconstruction

The interesting thing is that the reconstruction throughtput of Blaum-Roth overwhelms RTP, which is shown in Figure 4.6. We separate the recontruction into three steps, decoding matrix calculation, reconstruction which does all the XORs between corresponding symbols, and the last step is to write all the reconstructed symbols backto the disks. We found that RTP takes longer time to calculate the decoding matrix than Blaum-Roth although the XORing between symbols takes

shorter time. The current project deal with the decoding matrix calculation in more details and this document will talk about some idea in Advance Development.

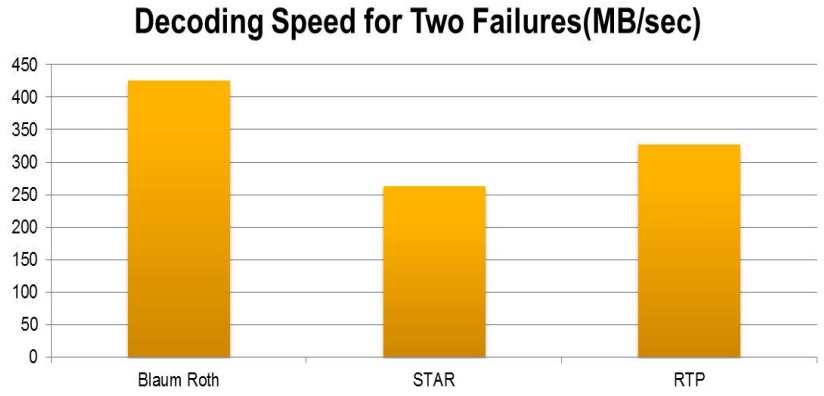

FIGURE 4.6: Two Failure Reconstruction Speed

4.3.3 Three Disk Failures

Figure 4.8 and Figure 4.7 show the number of XOR needs to reconstuct as well as the reconstruction speed. Although RTP needs least of XORs to reconstruct failures, the total speed is less than Blaum-Roth, and Blaum-Roth on the other hand, overwelm the other two in terms of speed without the downside of number of XORs. The reason is the same with the two disk failure reconstruction. The decoding matrix takes too long for RTP to be done and postpone the total reconstuction speed for RTP.

Another observation is that matrix reconstruction of RTP needs more XORs than close-form reconstruction. We can compute theoretical number of XORs required to reconstruct three failures in RTP's paper. Although close-form RTP reconstruction performs not as good as RDP, it still save a lot of computational XORs than matrix reconstruction. But as far as I know, some researchers are trying to eliminates the computational cost of matrix reconstruction by applying different scheduling algorithm to accelerate the matrix multiplication. Another way is using hardware support such as GPU or SIMD instruction set to matipulate matrix operations, this is also helpful to accelerate the matrix reconstruction.

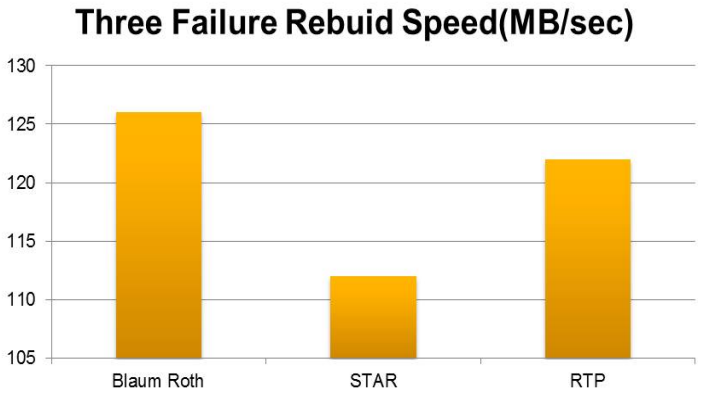

FIGURE 4.7: Three Failure Reconstruction Speed

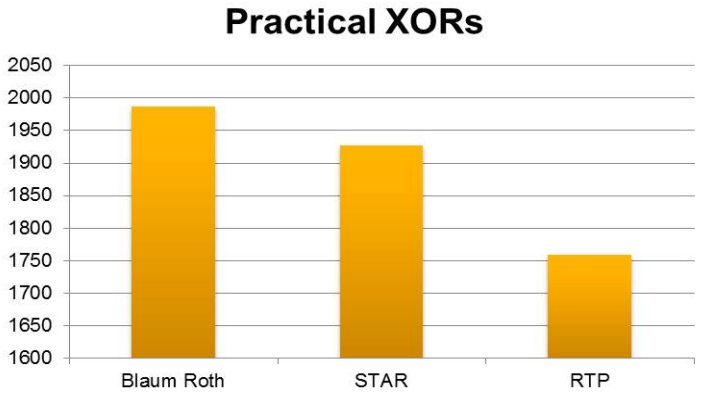

FIGURE 4.8: Three Failure Reconstruction

4.3.4 Small Write

We will show some ideas how to reduce the decoding matrix calculation time later and provide potential ways to prove RTP has the best performance to reconstuct the failures. But RTP has a significant downside comparing to other codes, it generates more back-end IOs for small write operation. We will show our experimental results in Figure 4.9.

We need to address this problem seriously because small write updates could generate much more back-end IOs than application expected. For example, Figure 4.11 shows an experimental result where how badly small writes can kill the system.

We can see that as the RAID level increases, the driver taks longer time to reponse to the user. This is the IO impact on the back-end, and we are talking a look at what happend in the back-end. Figure 4.10 shows the overal performance between different RAID levels. As we see, on the back-end side, disk-array generate much more IOs to updates the parity compared to the front-end side. Additionally, as

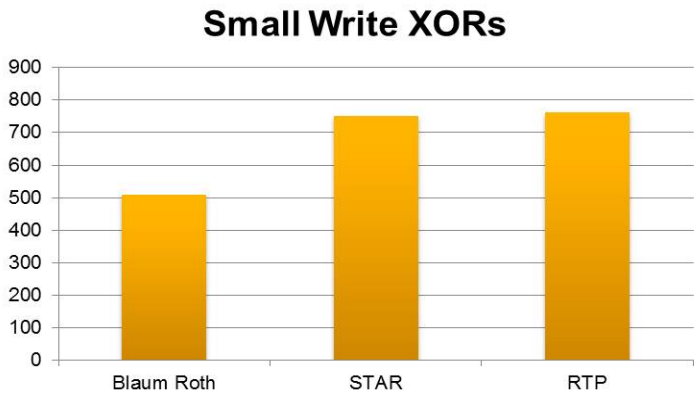

FIGURE 4.9: Small Write

the RAID level increases, the amount of the back-end IOs becomes larger. This write application effect will slow down the total IO reponse.

RAID OP	# IOs	Percentage	Resp. Time
RAID5: Rd	77258	54.5%	13.92
RAID5: Wr	64474	45.5%	31.48
RAID6: Rd	77258	47.2%	14.84
RAID6: Wr	86560	57.8%	35.85
RAID7: Rd	77258	41.6%	15.36
RAID7: Wr	108340	58.4%	39.04

FIGURE 4.10: Overal Performance

We could see the write application effect in Figure 4.11.

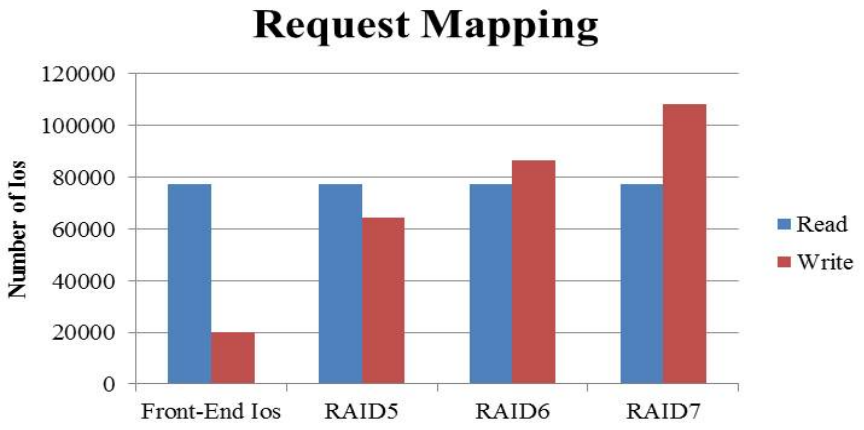

FIGURE 4.11: Request Mapping

Thus, if we do not carefully implement RTP in a way that eliminate small writes, the system will suffers from much more back-end small IOs.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

5.1.1 Matrix and Close-form Comparison for RTP

In encoding process, matrix reconstrucion of RTP performs better than close-form for smaller buffer size. When we choose the disk-array size which fit into L2 cache, all the data buffers could load into lower level cache and therefore multiplying symbols from different locations will not badly pollute the cache. But as the buffer size becomes larger, lower level cache can not hold all the data. Therefore, some symbols might be loaded into cache and used to compute some symbol in first failure. After recovering this failure symbol, the survived symbols could flushed out of the cache because new data symbols need to be loaded into cache to recover other failure symbols. Later on, those symbols which were initially loaded into cache and flushed out have to be loaded into cache again and calculate new failures. Thus, survived symbols need to repeatedly load into cache by calculating different failure symbols. Matrix reconstruction is only helpful for small disk-array.

In decoding process, matrix reconstruction still only performs well for small buffer size. Additionally, matrix reconstruction needs to calculate decoding matrix before multiplying this matrix with survived symbols. As our experiment shows decoding matrix is the bottleneck during reconstruction process.

5.1.2 Different Types of Erasure Codes with Matrix Method

Reed-Solomon is one of the most popular erasure code in storage systems. However, without any advanced computational CPU modes, Reed-Solomon's performance is far less than RTP. The reason is that Reed-Solomon needs to multiply symbols in Garlois-Field. CPU requires much more efforts to run these operations than XOR-based erasure codes like RTP. Thus, XOR-based erasure codes are more suitable for matrix reconstruction than Reed-Solomon.

Regarding to XOR-based erasure codes, we chose another two candidates Blaum-Roth code and STAR, and compare their performance with RTP. We found that these three codes performs the same when there is only one failure occurs. This makes sense because we only need to touch row parity to reconstruct single failure, and all these three codes have the same row parity. Double and Triple failures are different from single failure, and RTP needs less XORs to reconstruct failures than other two codes. However, the overall performance of RTP is not the best. The is due to the longer time cost to compute decoding matrix. Symbols must wait until decoding matrix is ready and then multiply it with survived symbols to reconstruct failures. Therefore, to further apply matrix method to disk-array, we need to address this bottleneck casefully.

Additionally, there is a very critical downside of RTP. It needs more XORs to do small write update. Blaum-Roth and STAR prohibit row parity from touching diagonal and anti-diagonal parities. But RTP combines row parity with data disks to construct diagonal parity and anti-diagonal parity. This causes more XORs to do small writes. For example, if we want to update one symbol in a data disk, Blaum-Roth and STAR first need to update that symbol. In addition, they also need to update corresponding symbols in three parities which this symbol contributed to. Let's take a look at RTP. Besides that data symbol and three parity symbols, it also need to update two extra symbols(may only one symbol, it depends on the failure pattern) on diagonal and anti-diagonal parities on which the modified symbol on row parity contributes to. Small write overhead could be very harmful to disk-array because it could generate much more back-end IOs behind applications.

5.2 Future Work

This is not the end of this research, and erasure codes is still very popular in storage world. As we mentioned before, there is a critical downside of matrix reconstruction. It may loads same symbols after a random period or different symbols from random locations. Neither of them allows cache to benefit from temporal or spatial locality. Hence, in order to apply matrix reconstruction in disk-array, we need to eliminate cache pollutions. This problem can be addressed in two perspectives. One is to carefully schedule XORs orders [11]. Every entry in decoding matrix correponds to an symbol need to be XORed in survived buffers. Therefore, we can apply good scheduling algorithms and reuse the XOR results. This could reduce the total number of XORs. Another way to address this problem is to increase cache spatial locality. We can load decompose matrix and multiply with corresponding data buffers. Since different data buffers independently contribute to failures, we can reconstruct them in a parallel mode.

We also need to think out of the box. Matrix reconstruction is promising but will not necessarily replace close-from. Contrarily, a lot of latest research focus on how to accelerate computations in Garlois-Field, which will definitely make Reed-Solomon code more competitive in future. Thus, how to optimize other erasure codes is another interesting track of this research. Furthermore, computation cost in Cloud is not as expensive as in disk-array [12], [13], [14], [15]. Therefore, Reed-Solomon code is also considered as a competitive option in distributed file system.

Chapter 6

Bibliography

[1] A. Goel P. Corbett, B. English. Row-diagonal parity for double disk failure correction. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies*, December 2005.

[2] J. S. Plank, K. M. Greenan, and E. L. Miller. Screaming fast Galois Field arithmetic using Intel SIMD instructions. In *FAST-2013: 11th Usenix Conference on File and Storage Technologies*, San Jose, February 2013.

[3] J. Hafner, V. Deenadhayalan, and K. Rao. Matrix methods for lost data reconstruction in erasure codes. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies*, December 2005.

[4] J. S. Plank and C. Huang. Tutorial: Erasure coding for storage applications. Slides presented at FAST-2013: 11th Usenix Conference on File and Storage Technologies, February 2013.

[5] Atul Goel and Peter Corbett. Raid triple parity. *SIGOPS Oper. Syst. Rev.*, 46(3), December 2012.

[6] H. Jin R. Buyya, T. Cortes. The evenodd code and its generalization: An efficient scheme for tolerating multiple disk failures in raid architectures. In *High Performance Mass Storage and Parallel I/O:Technologies and Applications*, 2002.

[7] L. Xu C. Huang. Star : An efficient coding scheme for correcting triple storage node failures. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies*, December 2005.

[8] M. Blaum. New array codes for multiple phased burst correction. In *Information Theory*, January 1993.

[9] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *FAST-2009: 7th Usenix Conference on File and Storage Technologies*, February 2009.

[10] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2. Technical Report CS-08-627, University of Tennessee, August 2008.

[11] J. Luo, L. Xu, and J. S. Plank. An efficient XOR-Scheduling algorithm for erasure codes encoding. In *DSN-2009: The International Conference on Dependable Systems and Networks*, Lisbon, Portugal, June 2009. IEEE.

[12] O. Khan, R. Burns, J. S. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *FAST-2012: 10th Usenix Conference on File and Storage Technologies*, San Jose, February 2012.

[13] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. Xoring elephants: Novel erasure codes for big data. *Proc. VLDB Endow.*, 6(5), March 2013.

[14] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, 2012.

[15] Bin Fan, Wittawat Tantisiriroj, Lin Xiao, and Garth Gibson. Diskreduce: Raid for data-intensive scalable computing. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, 2009.