Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications

Version 1.2

James S. Plank* Scott Simmerman Catherine D. Schuman

Technical Report CS-08-627
Department of Electrical Engineering and Computer Science
University of Tennessee
Knoxville, TN 37996

`http://www.cs.utk.edu/~plank/plank/papers/CS-08-627.html`

This describes revision 1.2 of the code.

Abstract

This paper descibes version 1.2 of **jerasure**, a library in C/C++ that supports erasure coding in storage applications. In this paper, we describe both the techniques and algorithms, plus the interface to the code. Thus, this serves as a quasi-tutorial and a programmer's guide.

Version 1.2 adds Blaum-Roth and Liber8tion coding to the library, provides better examples, and an example file encoder/decoder. Additionally, it removes a bug from the previous writeup: the packetsize must be a multiple of **sizeof(long)**. It does *not* have to be a multiple of w.

If You Use This Library or Document

Please send me an email to let me know how it goes. One of the ways in which I am evaluated both internally and externally is by the impact of my work, and if you have found this library and/or this document useful, I would like to be able to document it. Please send mail to *plank@eecs.utk.edu*.

The library itself is protected by the GNU LGPL. It is free to use and modify within the bounds of the LGPL. None of the techniques implemented in this library have been patented.

Finding the Code

Please see `http://www.cs.utk.edu/~plank/plank/papers/CS-08-627.html` to get the TAR file for this code.

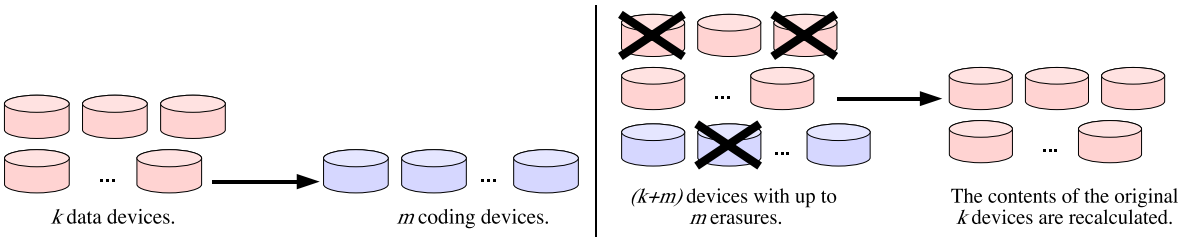
*`plank@cs.utk.edu` or `plank@eecs.utk.edu`, 865-974-4397, This material is based upon work supported by the National Science Foundation under grant CNS-0615221.

Contents

1 Introduction

Erasure coding for storage applications is growing in importance as storage systems grow in size and complexity. This paper describes **jerasure**, a library in C/C++ that supports erasure coding applications. **Jerasure** has been designed to be modular, fast and flexible. It is our hope that storage designers and programmers will find **jerasure** to be a convenient tool to add fault tolerance to their storage systems.

Jerasure supports a *horizontal* mode of erasure codes. We assume that we have k devices that hold data. To that, we will add m devices whose contents will be calculated from the original k devices. If the erasure code is a *Maximum Distance Separable (MDS)* code, then the entire system will be able to tolerate the loss of any m devices.



(a) Encoding. (b) Decoding.

Figure 1: The act of *encoding* takes the contents of k data devices and encodes them on m coding devices. The act of *decoding* takes some subset of the collection of $(k + m)$ total devices and from them recalcalates the original k devices of data.

As depicted in Figure 1, the act of encoding takes the original k data devices, and from them calculates m coding devices. The act of decoding takes the collection of $(k + m)$ devices with erasures, and from the surviving devices recalculates the contents of the original k data devices.

Most codes have a third parameter w, which is the *word size*. The description of a code views each device as having w bits worth of data. The data devices are denoted D_0 through D_{k-1} and the coding devices are denoted C_0 through C_{m-1}. Each device D_i or C_j holds w bits, denoted $d_{i,0}, \ldots d_{i,w-1}$ and $c_{i,0}, \ldots c_{i,w-1}$. In reality of course, devices hold megabytes of data. To map the description of a code to its realization in a real system, we do one of two things:

1. When $w \in \{8, 16, 32\}$, we can consider each collection of w bits to be a byte, short word or word respectively. Consider the case when $w = 8$. We may view each device to hold B bytes. The first byte of each coding device will be encoded with the first byte of each data device. The second byte of each coding device will be encoded with the second byte of each data device. And so on. This is how Standard Reed-Solomon coding works, and it should be clear how it works when $w = 16$ or $w = 32$.

2. Most other codes work by defining each coding bit $c_{i,j}$ to be the bitwise exclusive-or (XOR) of some subset of the other bits. To implement these codes in a real system, we assume that the device is composed of w *packets* of equal size. Now each packet is calculated to be the bitwise exclusive-or of some subset of the other packets. In this way, we can take advantage of the fact that we can perform XOR operations on whole computer words rather than on bits.

   The process is illustrated in Figure 2. In this figure, we assume that $k = 4$, $m = 2$ and $w = 4$. Suppose that a code is defined such that coding bit $c_{1,0}$ is goverened by the equation:

$$c_{1,0} = d_{0,0} \oplus d_{1,1} \oplus d_{2,2} \oplus d_{3,3},$$

where $\oplus$ is the XOR operation. Figure 2 shows how the coding packet corresponding to $c_{1,0}$ is calculated from the data packets corresponding to $d_{0,0}$, $d_{1,1}$, $d_{2,2}$ and $d_{3,3}$. We call the size of each packet the *packet size*, and the size of w packets to be the *coding block size*. The packetsize must be a multiple of the computer's word size so obviously, the coding block size will be a multiple of $w * packetsize$.

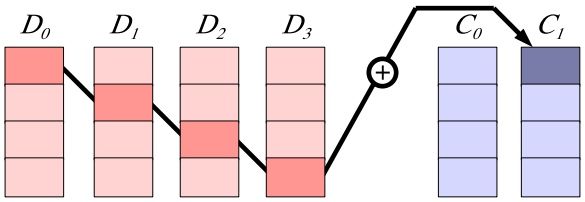


Figure 2: Although codes are described on systems of w bits, their implementation employs *packets* that are much larger. Each packet in the implementation corresponds to a bit of the description. This figure is showing how the equation $c_{1,0} = d_{0,0} \oplus d_{1,1} \oplus d_{2,2} \oplus d_{3,3}$ is realized in an implementation.

2 The Modules of the Library

This library is broken into five modules, each with its own header file and implementation in C. Typically, when using a code, one only needs three of these modules: **galois**, **jerasure** and one of the others. The modules are:

1. **galois.h/galois.c**: These are procedures for Galois Field Arithmetic as described and implemented in [Pla07].
2. **jerasure.h/jerasure.c**: These are kernel routines that are common to most erasure codes. They do not depend on any module other than **galois**. They include support for matrix-based coding and decoding, bit-matrix-based coding and decoding, conversion of bit-matrices to schedules, matrix and bit-matrix inversion.
3. **reedsol.h/reedsol.c**: These are procedures for creating distribution matrices for Reed-Solomon coding [RS60, Pla97, PD05]. They also include the optimized version of Reed-Solomon encoding for RAID-6 as discussed in [Anv07].
4. **cauchy.h/cauchy.c**: These are procedures for performing Cauchy Reed-Solomon coding [BKK +95, PX06], which employs a different matrix construction than classic Reed-Solomon coding. We include support for creating optimal Cauchy distribution matrices for RAID-6, and for creating distribution matrices that are better than those currently published.
5. **liberation.h/liberation.c**: These are procedures for performing RAID-6 coding and decoding with minimal density MDS codes – the RAID-6 Liberation codes [Pla08b], Blaum-Roth codes [BR99] and the RAID-6 Liber8tion code [Pla08a]. These are bit-matrix codes that perform much better than the Reed-Solomon variants and better than EVENODD coding [BBBM95]. In some cases, they even outperform RDP [CEG +04], which is the best currently known RAID-6 code.

Each module is described in its own section below. Additionally, there are example programs that show the usage of each module.

3 Matrix-Based Coding In General

The mechanics of matrix-based coding are explained in great detail in [Pla97]. We give a high-level overview here.

> **Authors' Caveat:** *We are using old nomenclature of "distribution matrices." In standard coding theory, the "distribution matrix" is the transpose of the Generator matrix. In the next revision of* **jerasure***, we will update the nomenclature to be more consistent with classic coding theory.*

Suppose we have k data words and m coding words, each composed of w bits. We can describe the state of a matrix-based coding system by a matrix-vector product as depicted in Figure 3. The matrix is called a *distribution matrix* and is a $(k+m) \times k$ matrix. The elements of the matrix are numbers in $GF(2^w)$ for some value of w. This means that they are integers between 0 and $2^w - 1$, and arithmetic is performed using Galois Field arithmetic: addition is equal to XOR, and multiplication is implemented in a variety of ways. The Galois Field arithmetic library in [Pla07] has procedures which implement Galois Field arithmetic.

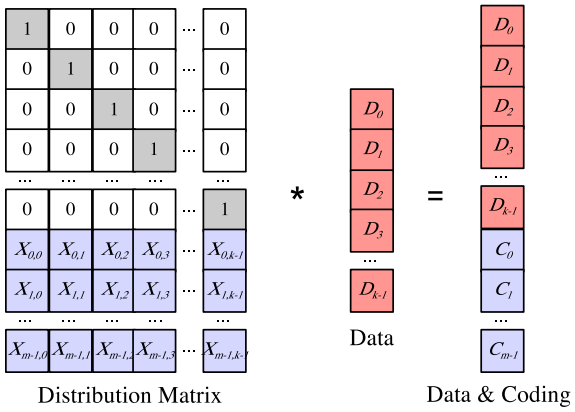


Figure 3: Using a matrix-vector product to describe a coding system.

The top k rows of the distribution matrix compsose a $k \times k$ identity matrix. The remaining m rows are called the *coding matrix*, and are defined in a variety of ways [Rab89, Pre89, BKK+95, PD05]. The distribution matrix is multiplied by a vector that contains the data words and yields a product vector containing both the data and the coding words. Therefore, to encode, we need to perform m dot products of the coding matrix with the data.

To decode, we note that each word in the system has a corresponding row of the distribution matrix. When devices fail, we create a decoding matrix from k rows of the distribution that correspond to non-failed devices. Note that this matrix multiplied by the original data equals the k survivors whose rows we selected. If we invert this matrix and multiply it by both sides of the equation, then we are given a decoding equation – the inverted matrix multiplied by the survivors equals the original data.

4 Bit-Matrix Coding In General

Bit-matrix coding is first described in the original Cauchy Reed-Solomon coding paper [BKK+95]. To encode and decode with a bit-matrix, we expand a distribution matrix in $GF(2^w)$ by a factor of w in each direction to yield

a $w(k+m) \times wk$ matrix which we call a *binary distribution matrix (BDM)*. We multiply that by a wk element vector, which is composed of w bits from each data device. The product is a $w(k+m)$ element vector composed of w bits from each data and coding device. This is depicted in Figure 4. It is useful to visualize the matrix as being composed of $w \times w$ sub-matrices.

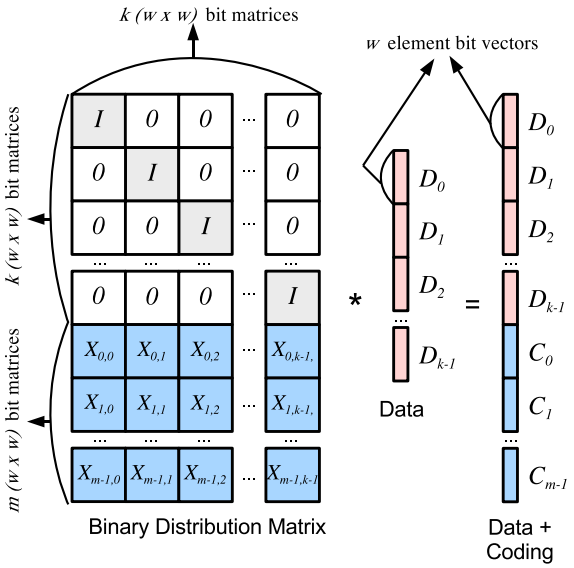


Figure 4: Describing a coding system with a bit-matrix-vector product.

As with the matrix-vector product in $GF(2^w)$, each row of the product corresponds to a row of the BDM, and is computed as the dot product of that row and the data bits. Since all elements are bits, we may perform the dot product by taking the XOR of each data bit whose element in the matrix's row is one. In other words, rather than performing the dot product with additions and multiplications, we perform it only with XORs. Moreover, the performance of this dot product is directly related to the number of ones in the row. Therefore, it behooves us to find matrices with few ones.

Decoding with bit-matrices is the same as with matrices over $GF(2^w)$, except now each device corresponds to w rows of the matrix, rather than one. Also keep in mind that a bit in this description corresponds to a packet in the implementation.

While the classic construction of bit-matrices starts with a standard distribution matrix in $GF(2^w)$, it is possible to construct bit-matrices that have no relation to Galois Field arithmetic yet still have desired coding and decoding properties. The minimal density RAID-6 codes work in this fashion.

4.1 Using a schedule rather than a bit-matrix

Consider the act of encoding with a bit-matrix. We give an example in Figure 5, where $k = 3$, $w = 5$, and we are calculating the contents of one coding device. The straightforward way to encode is to calculate the five dot products for each of the five bits of the coding device, and we can do that by traversing each of the five rows, performing XORs where there are ones in the matrix.

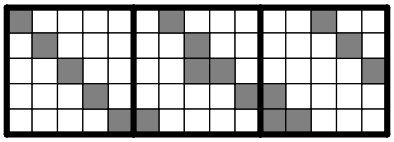


Figure 5: An example super-row of a bit-matrix for $k = 3$, $w = 5$.

Since the matrix is sparse, it is more efficient to precompute the coding operations, rather than traversing the matrix each time one encodes. The data structure that we use to represent encoding is a *schedule*, which is a list of 5-tuples:

$$< op, s_d, s_b, d_d, d_b >,$$

where op is an operation code: 0 for copy and 1 for XOR, s_d is the id of the source device and s_b is the bit of the source device. The last two elements, d_d and d_b are the destination device and bit. By convention, we identify devices using integers from zero to $k + m - 1$. An id $i < k$ identifies data device D_i, and an id $i \geq k$ identifies coding device C_{i-k}.

A schedule for encoding using the bit-matrix in Figure 5 is shown in Figure 6.

$< 0, 0, 0, 3, 0 >, < 1, 1, 1, 3, 0 >, < 1, 2, 2, 3, 0 >,$	$c_{0,0} = d_{0,0} \oplus d_{1,1} \oplus d_{2,2}$
$< 0, 0, 1, 3, 1 >, < 1, 1, 2, 3, 1 >, < 1, 2, 3, 3, 1 >,$	$c_{0,1} = d_{0,1} \oplus d_{1,2} \oplus d_{2,3}$
$< 0, 0, 2, 3, 2 >, < 1, 1, 2, 3, 2 >, < 1, 1, 3, 3, 2 >, < 1, 2, 4, 3, 2 >,$	$c_{0,2} = d_{0,2} \oplus d_{1,2} \oplus d_{1,3} \oplus d_{2,4}$
$< 0, 0, 3, 3, 3 >, < 1, 1, 4, 3, 3 >, < 1, 2, 0, 3, 3 >,$	$c_{0,3} = d_{0,3} \oplus d_{1,4} \oplus d_{2,0}$
$< 0, 0, 4, 3, 4 >, < 1, 1, 0, 3, 4 >, < 1, 2, 0, 3, 4 >, < 1, 2, 1, 3, 4 > .$	$c_{0,4} = d_{0,4} \oplus d_{1,0} \oplus d_{2,0} \oplus d_{2,1}$
(a)	(b)

Figure 6: A schedule of bit-matrix operations for the bit-matrix in Figure 5. (a) shows the schedule, and (b) shows the dot-product equations corresponding to each line of the schedule.

As noted in [HDRT05, Pla08b], one can derive schedules for bit-matrix encoding and decoding that make use of common expressions in the dot products, and therefore can perform the bit-matrix-vector product with fewer XOR operations than simply traversing the bit-matrix. This is how RDP encoding works with optimal performance [CEG +04], even though there are more than kw ones in the last w rows of its BDM. We term such scheduling *smart* scheduling, and scheduling by simply traversing the matrix *dumb* scheduling.

5 MDS Codes

A code is MDS if it can recover the data following the failure of any m devices. If a matrix-vector product is used to define the code, then it is MDS if every combination of k rows composes an invertible matrix. If a bit-matrix is used, then we define a *super-row* to be a row's worth of $w \times w$ submatrices. The code is MDS if every combination of k super-rows composes an invertible matrix. Again, one may generate an MDS code using standard techniques such as employing a Vandermonde matrix [PD05] or Cauchy matrix [Rab89, BKK +95]. However, there are other constructions that also yield MDS matrices, such as EVENODD coding [BBBM95], RDP coding [CEG +04], the STAR code [HX05], Feng's codes [FDBS05a, FDBS05b] and the minimal density RAID-6 codes [BR99, Pla08a, Pla08b].

6 Part 1 of the Library: Galois Field Arithmetic

The files **galois.h** and **galois.c** contain procedures for Galois Field arithmetic in $GF(2^w)$ for $1 \leq w \leq 32$. It contains procedures for single arithmetic operations, for XOR-ing a region of bytes, and for performing multiplication of a region of bytes by a constant in $GF(2^8)$, $GF(2^{16})$ and $GF(2^{32})$. The procedures are defined in a separate technical report which focuses solely on Galois Field arithmetic [Pla07].

For the purposes of **jerasure**, the following procedures from **galois.h** and **galois.c** are used:

- **galois_single_multiply(int a, int b, int w)** and **galois_single_divide(int a, int b, int w)**: These perform multiplication and division on single elements **a** and **b** of $GF(2^{\mathbf{w}})$.
- **galois_region_xor(char *r1, char *r2, char *r3, int nbytes)**: This XORs two regions of bytes, **r1** and **r2** and places the sum in **r3**. Note that **r3** may be equal to **r1** or **r2** if we are replacing one of the regions by the sum. **Nbytes** must be a multiple of the machine's **long** word size.
- **galois_w08_region_multiply(char *region, int multby, int nbytes, char *r2, int add)**: This multiplies an entire region of bytes by the constant **multby** in $GF(2^8)$. If **r2** is **NULL** then **region** is overwritten. Otherwise, if **add** is zero, the products are placed in **r2**. If **add** is non-zero, then the products are XOR'd with the bytes in **r2**.
- **galois_w16_region_multiply()** and **galois_w32_region_multiply()** are identical to **galois_w08_region_multiply()**, except they are in $GF(2^{16})$ and $GF(2^{32})$ respectively.

7 Part 2 of the Library: Kernel Routines

The files **jerasure.h** and **jerasure.c** implement procedures that are common to many aspects of coding. We give example programs that make use of them in Section 7.7 below.

Before describing the procedures that compose **jerasure.c**, we detail the arguments that are common to multiple procedures:

- **int k**: The number of data devices.
- **int m**: The number of coding devices.
- **int w**: The word size of the code.
- **int packetsize**: The packet size as defined in section 1. This must be a multiple of **sizeof(long)**.
- **int size**: The total number of bytes per device to encode/decode. This must be a multiple of **sizeof(long)**. If a bit-matrix is being employed, then it must be a multiple of **packetsize * w**. If one desires to encode data blocks that do not conform to these restrictions, than one must pad the data blocks with zeroes so that the restrictions are met.
- **int *matrix**: This is an array with **k*m** elements that represents the coding matrix — i.e. the last **m** rows of the distribution matrix. Its elements must be between 0 and $2^{\mathbf{w}} - 1$. The element in row i and column j is in **matrix[i*k+j]**.
- **int *bitmatrix**: This is an array of **w*m*w*k** elements that compose the last **wm** rows of the BDM. The element in row i and column j is in **bitmatrix[i*k*w+j]**.

- **char **data_ptrs**: This is an array of **k** pointers to **size** bytes worth of data. Each of these must be long word aligned.
- **char **coding_ptrs**: This is an array of **m** pointers to **size** bytes worth of coding data. Each of these must be long word aligned.
- **int *erasures**: This is an array of id's of erased devices. Id's are numbers between 0 and **k+m-1** as described in Section 4.1. If there are e erasures, then elements 0 through $e-1$ of **erasures** identify the erased devices, and **erasures[**e**]** must equal -1.
- **int *erased**: This is an alternative way of specifying erasures. It is a **k+m** element array. Element i of the array represents the device with id i. If **erased[**i**]** equals 0, then device i is working. If **erased[**i**]** equals 1, then it is erased.
- **int **schedule**: This is an array of 5-element integer arrays. It represents a schedule as defined in Section 4.1. If there are o operations in the schedule, then **schedule** must have at least $o+1$ elements, and **schedule[**o**][0]** should equal -1.
- **int ***cache**: When **m** equals 2, there are few enough combinations of failures that one can precompute all possible decoding schedules. This is held in the **cache** variable. We will not describe its structure – just that it is an **(int ***)**.
- **int row_k_ones**: When $m > 1$ and the first row of the coding matrix is composed of all ones, then there are times when we can improve the performance of decoding by not following the methodology described in Section 3. This is true when coding device zero is one of the survivors, and more than one data device has been erased. In this case, it is better to decode all but one of the data devices as described in Section 3, but decode the last data device using the other data devices and coding device zero. For this reason, some of the decoding procedures take a paramater **row_k_ones**, which should be one if the first row of **matrix** is all ones. The same optimization is available when the first w rows of **bitmatrix** compose k identity matrices – **row_k_ones** should be set to one when this is true as well.
- **int *decoding_matrix**: This is a $k \times k$ matrix or $wk \times wk$ bit-matrix that is used to decode. It is the matrix constructed by employing relevant rows of the distribution matrix and inverting it.
- **int *dm_ids**: As described in Section 3, we create the decoding matrix by selecting k rows of the distribution matrix that correspond to surviving devices, and then inverting that matrix. This yields **decoding_matrix**. The product of **decoding_matrix** and these survivors is the original data. **dm_ids** is a vector with k elements that contains the id's of the devices corresponding to the rows of the decoding matrix. In other words, this contains the id's of the survivors. When decoding with a bit-matrix **dm_ids** still has k elements — these are the id's of the survivors that correspond to the k super-rows of the decoding matrix.

7.1 Matrix/Bitmatrix/Schedule Creation Routines

When we use an argument from the list above, we omit its type for brevity.

- **int *jerasure_matrix_to_bitmatrix(k, m, w, matrix)**: This converts a $m \times k$ matrix in $GF(2^w)$ to a $wm \times wk$ bit-matrix, using the technique described in [BKK+95]. If **matrix** is a coding matrix for an MDS code, then the returned bit-matrix will also describe an MDS code.

- **int **jerasure_dumb_bitmatrix_to_schedule(k, m, w, bitmatrix)**: This converts the given bit-matrix into a schedule of coding operations using the straightforward technique of simply traversing each row of the matrix and scheduling XOR operations whenever a one is encountered.
- **int **jerasure_smart_bitmatrix_to_schedule(k, m, w, bitmatrix)**: This converts the given bit-matrix into a schedule of coding operations using the optimization described in [Pla08b]. Basically, it tries to use encoded bits (or decoded bits) rather than simply the data (or surviving) bits to reduce the number of XORs. Note, that when a smart schedule is employed for decoding, we don't need to specify **row_k_ones**, because the schedule construction technique automatically finds this optimization.
- **int ***jerasure_generate_schedule_cache(k, m, w, bitmatrix, int smart)**: This only works when $m = 2$. In this case, it generates schedules for every combination of single and double-disk erasure decoding. It returns a cache of these schedules. If **smart** is one, then **jerasure_smart_bitmatrix_to_schedule()** is used to create the schedule. Otherwise, **jerasure_dumb_bitmatrix_to_schedule()** is used.
- **void jerasure_free_schedule(schedule)**: This frees all allocated memeory for a schedule that is created by either **jerasure_dumb_bitmatrix_to_schedule()** or **jerasure_smart_bitmatrix_to_schedule()**.
- **void jerasure_free_schedule_cache(k, m, cache)**: This frees all allocated data for a schedule cache created by **jerasure_generate_schedule_cache()**.

7.2 Encoding Routines

- **void jerasure_do_parity(k, data_ptrs, char *parity_ptr, size)**: This calculates the parity of **size** bytes of data from each of k regions of memory accessed by **data_ptrs**. It puts the result into the **size** bytes pointed to by **parity_ptr**. Like each of **data_ptrs**, **parity_ptr** must be long word aligned, and **size** must be a multiple of **sizeof(long)**.
- **void jerasure_matrix_encode(k, m, w, matrix, data_ptrs, coding_ptrs, size)**: This encodes with a matrix in $GF(2^w)$ as described in Section 3 above. w must be $\in \{8, 16, 32\}$.
- **void jerasure_bitmatrix_encode(k, m, w, bitmatrix, data_ptrs, coding_ptrs, size, packetsize)**: This encodes with a bit-matrix. Now w may be any number between 1 and 32.
- **void jerasure_schedule_encode(k, m, w, schedule, data_ptrs, coding_ptrs, size, packetsize)**: This encodes with a schedule created from either **jerasure_dumb_bitmatrix_to_schedule()** or **jerasure_smart_bitmatrix_to_schedule()**.

7.3 Decoding Routines

Each of these returns an integer which is zero on success or -1 if unsuccessful. Decoding can be unsuccessful if there are too many erasures.

- **int jerasure_matrix_decode(k, m, w matrix, row_k_ones, erasures, data_ptrs, coding_ptrs, size)**: This decodes using a matrix in $GF(2^w)$, $w \in \{8, 16, 32\}$. This works by creating a decoding matrix and performing the matrix/vector product, then re-encoding any erased coding devices. When it is done, the decoding matrix is discarded. If you want access to the decoding matrix, you should use **jerasure_make_decoding_matrix()** below.

- **int jerasure_bitmatrix_decode(k, m, w bitmatrix, row_k_ones, erasures, data_ptrs, coding_ptrs, size, packetsize)**: This decodes with a bit-matrix rather than a matrix. Note, it does not do any scheduling – it simply creates the decoding bit-matrix and uses that directly to decode. Again, it discards the decoding bit-matrix when it is done.

- **int jerasure_schedule_decode_lazy(k, m, w bitmatrix, erasures, data_ptrs, coding_ptrs, size, packetsize, int smart)**: This decodes by creating a schedule from the decoding matrix and using that to decode. If **smart** is one, then **jerasure_smart_bitmatrix_to_schedule()** is used to create the schedule. Otherwise, **jerasure_dumb_bitmatrix_to_schedule()** is used. Note, there is no **row_k_ones**, because if **smart** is one, the schedule created will find that optimization anyway. This procedure is a bit subtle, because it does a little more than simply create the decoding matrix – it creates it and then adds rows that decode failed coding devices from the survivors. It derives its schedule from that matrix. This technique is also employed when creating a schedule cache using **jerasure_generate_schedule_cache()**. The schedule and all data structures that were allocated for decoding are freed when this procedure finishes.

- **int jerasure_schedule_decode_cache(k, m, w cache, erasures, data_ptrs, coding_ptrs, size, packetsize)**: This uses the schedule cache to decode when $m = 2$.

- **int jerasure_make_decoding_matrix(k, m, w matrix, erased, decoding_matrix, dm_ids)**: This does not decode, but instead creates the decoding matrix. Note that both **decoding_matrix** and **dm_ids** should be allocated and passed to this procedure, which will fill them in. **Decoding_matrix** should have k^2 integers, and **dm_ids** should have k integers.

- **int jerasure_make_decoding_bitmatrix(k, m, w matrix, erased, decoding_matrix, dm_ids)**: This does not decode, but instead creates the decoding bit-matrix. Again, both **decoding_matrix** and **dm_ids** should be allocated and passed to this procedure, which will fill them in. This time **decoding_matrix** should have k^2w^2 integers, while **dm_ids** still has k integers.

- **int *jerasure_erasures_to_erased(k, m, erasures)**: This converts the specification of **erasures** defined above into the specification of **erased** also defined above.

7.4 Dot Product Routines

- **void jerasure_matrix_dotprod(k, w, int *matrix_row, int *src_ids, int dest_id, data_ptrs, coding_ptrs, size)**: This performs the multiplication of one row of an encoding/decoding matrix times data/survivors. The id's of the source devices (corresponding to the id's of the vector elements) are in **src_ids**. The id of the destination device is in **dest_id**. w must be $\in \{8, 16, 32\}$. When a one is encountered in the matrix, the proper XOR/copy operation is performed. Otherwise, the operation is multiplication by the matrix element in $GF(2^w)$ and an XOR into the destination.

- **void jerasure_bitmatrix_dotprod(k, w, int *bitmatrix_row, int *src_ids, int dest_id, data_ptrs, coding_ptrs, size, packetsize)**: This is the analogous procedure for bit-matrices. It performs w dot products according to the w rows of the matrix specified by **bitmatrix_row**.

- **void jerasure_do_scheduled_operations(char **ptrs, schedule, packetsize)**: This performs a schedule on the pointers specified by **ptrs**. Although w is not specified, it performs the schedule on w(**packetsize**) bytes. It is assumed that **ptrs** is the right size to match **schedule**. Typically, this is $k + m$.

7.5 Basic Matrix Operations

- **int jerasure_invert_matrix(int *mat, int *inv, int rows, int w)**: This inverts a (**rows** $\times$ **rows**) matrix in $GF(2^w)$. It puts the result in **inv**, which must be allocated to contain **rows**^{2} integers. The matrix **mat** is destroyed after the inversion. It returns 0 on success, or -1 if the matrix was not invertible.
- **int jerasure_invert_bitmatrix(int *mat, int *inv, int rows)**: This is the analogous procedure for bit-matrices. Obviously, one can call **jerasure_invert_matrix()** with $w = 1$, but this procedure is faster.
- **int jerasure_invertible_matrix(int *mat, int rows, int w)**: This does not perform the inversion, but simply returns 1 or 0, depending on whether **mat** is invertible. It destroys **mat**.
- **int jerasure_invertible_bitmatrix(int *mat, int rows)**: This is the analogous procedure for bit-matrices.
- **void jerasure_print_matrix(int *matrix, int rows, int cols, int w)**: This prints a matrix composed of elements in $GF(2^w)$ on standard output. It uses w to determine spacing.
- **void jerasure_print_bitmatrix(int *matrix, int rows, int cols, int w)**: This prints a bit-matrix on standard output. It inserts a space between every w characters, and a blank line after every w lines. Thus super-rows and super-columns are easy to identify.
- **int *jerasure_matrix_multiply(int *m1, int *m2, int r1, int c1, int r2, int c2, int w)**: This performs matrix multiplication in $GF(2^w)$. The matrix **m1** should be a (**r1** $\times$ **c1**) matrix, and **m2** should be a (**r2** $\times$ **c2**) matrix. Obviously, **c1** should equal **r2**. It will return a (**r1** $\times$ **c2**) matrix equal to the product.

7.6 Statistics

Finally, **jerasure.c** keeps track of three quantities:

- The number of bytes that have been XOR'd using **galois_region_xor()**.
- The number of bytes that have been multiplied by a constant in $GF(2^w)$, using **galois_w08_region_multiply()**, **galois_w16_region_multiply()** or **galois_w32_region_multiply()**.
- The number of bytes that have been copied using **memcpy()**.

There is one procedure that allows access to those values:

- **void jerasure_get_stats(double *fill_in)**: The argument **fill_in** should be an array of three **double**s. The procedure will fill in the array with the three values above in that order. The unit is bytes. After calling **jerasure_get_stats()**, the counters that keep track of the quantities are reset to zero.

The procedure **galois_w08_region_multiply()** and its kin have a parameter that causes it to XOR the product with another region with the same overhead as simply performing the multiplication. For that reason, when these procedures are called with this functionality enabled, the resulting XORs are not counted with the XOR's performed with **galois_region_xor()**.

7.7 Example Programs to Demonstrate Use

In the **Examples** directory, there are eight programs that demonstrate nearly every procedure call in **jerasure.c**. They are as follows:

- **jerasure_01.c**: This takes three parameters: r, c and w. It creates an $r \times c$ matrix in $GF(2^w)$, where the element in row i, column j is equal to 2^{ci+j} in $GF(2^w)$. Rows and columns are zero-indexed. Example:

```
UNIX> jerasure_01 3 15 8
  1   2   4   8  16  32  64 128  29  58 116 232 205 135  19
 38  76 152  45  90 180 117 234 201 143   3   6  12  24  48
 96 192 157  39  78 156  37  74 148  53 106 212 181 119 238
UNIX>
```

  This demonstrates usage of **jerasure_print_matrix()** and **galois_single_multiply()**.

- **jerasure_02.c**: This takes three parameters: r, c and w. It creates the same matrix as in **jerasure_01**, and then converts it to a $rw \times cw$ bit-matrix and prints it out. Example:

```
UNIX> jerasure_01 3 10 4
 1  2  4  8  3  6 12 11  5 10
 7 14 15 13  9  1  2  4  8  3
 6 12 11  5 10  7 14 15 13  9
UNIX> jerasure_02 3 10 4
1000 0001 0010 0100 1001 0011 0110 1101 1010 0101
0100 1001 0011 0110 1101 1010 0101 1011 0111 1111
0010 0100 1001 0011 0110 1101 1010 0101 1011 0111
0001 0010 0100 1001 0011 0110 1101 1010 0101 1011

1011 0111 1111 1110 1100 1000 0001 0010 0100 1001
1110 1100 1000 0001 0010 0100 1001 0011 0110 1101
1111 1110 1100 1000 0001 0010 0100 1001 0011 0110
0111 1111 1110 1100 1000 0001 0010 0100 1001 0011

0011 0110 1101 1010 0101 1011 0111 1111 1110 1100
1010 0101 1011 0111 1111 1110 1100 1000 0001 0010
1101 1010 0101 1011 0111 1111 1110 1100 1000 0001
0110 1101 1010 0101 1011 0111 1111 1110 1100 1000
UNIX>
```

  This demonstrates usage of **jerasure_print_bitmatrix()** and **jerasure_matrix_to_bitmatrix()**.

- **jerasure_03.c**: This takes three parameters: k and w. It creates a $k \times k$ Cauchy matrix in $GF(2^w)$, and tests invertibility.

  The parameter k must be less than 2^w. The element in row i, column j is set to:

$$\frac{1}{i \oplus (2^w - j - 1)}$$

  where division is in $GF(2^w)$, $\oplus$ is XOR and subtraction is regular integer subtraction. When $k > 2^{w-1}$, there will be i and j such that $i \oplus (2^w - j - 1) = 0$. When that happens, we set that matrix element to zero.

  After creating the matrix and printing it, we test whether it is invertible. If $k \leq 2^{w-1}$, then it will be invertible. Otherwise it will not. Then, if it is invertible, it prints the inverse, then multplies the inverse by the original matrix and prints the product which is the identity matrix. Examples:

```
UNIX> jerasure_03 4 3
The Cauchy Matrix:
4 3 2 7
3 4 7 2
2 7 4 3
7 2 3 4

Invertible: Yes

Inverse:
1 2 5 3
2 1 3 5
5 3 1 2
3 5 2 1

Inverse times matrix (should be identity):
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
UNIX> jerasure_03 5 3
The Cauchy Matrix:
4 3 2 7 6
3 4 7 2 5
2 7 4 3 1
7 2 3 4 0
6 5 1 0 4

Invertible: No
UNIX>
```

This demonstrates usage of **jerasure_print_matrix**(), **jerasure_invertible_matrix**(), **jerasure_invert_matrix**() and **jerasure_matrix_multiply**().

- **jerasure_04.c**: This does the exact same thing as **jerasure_03**, except it uses **jerasure_matrix_to_bitmatrix**() to convert the Cauchy matrix to a bit-matrix, and then uses the bit-matrix operations to test invertibility and to invert the matrix. Examples:

```
UNIX> jerasure_04 4 3
The Cauchy Bit-Matrix:
010 101 001 111
011 111 101 100
101 011 010 110

101 010 111 001
111 011 100 101
011 101 110 010

001 111 010 101
101 100 011 111
010 110 101 011

111 001 101 010
100 101 111 011
110 010 011 101
```

```
Invertible: Yes

Inverse:
100 001 110 101
010 101 001 111
001 010 100 011

001 100 101 110
101 010 111 001
010 001 011 100

110 101 100 001
001 111 010 101
100 011 001 010

101 110 001 100
111 001 101 010
011 100 010 001

Inverse times matrix (should be identity):
100 000 000 000
010 000 000 000
001 000 000 000

000 100 000 000
000 010 000 000
000 001 000 000

000 000 100 000
000 000 010 000
000 000 001 000

000 000 000 100
000 000 000 010
000 000 000 001
UNIX> jerasure_04 5 3
The Cauchy Bit-Matrix:
010 101 001 111 011
011 111 101 100 110
101 011 010 110 111

101 010 111 001 110
111 011 100 101 001
011 101 110 010 100

001 111 010 101 100
101 100 011 111 010
010 110 101 011 001

111 001 101 010 000
100 101 111 011 000
110 010 011 101 000

011 110 100 000 010
110 001 010 000 011
111 100 001 000 101
```

```
Invertible: No
UNIX>
```

This demonstrates usage of **jerasure_print_bitmatrix()**, **jerasure_matrix_to_bitmatrix()**, **jerasure_invertible_bitmatrix()**, **jerasure_invert_bitmatrix()** and **jerasure_matrix_multiply()**.

- **jerasure_05.c**: This takes four parameters: k, m, w and $size$, and performs a basic Reed-Solomon coding example in $GF(2^w)$. w must be either 8, 16 or 32, and the sum $k + m$ must be less than or equal to 2^w. The total number of bytes for each device is given by $size$ which must be a multiple of **sizeof(long)**. It first sets up an $m \times k$ Cauchy coding matrix where element i, j is:

$$\frac{1}{i \oplus (m + j)}$$

where division is in $GF(2^w)$, $\oplus$ is XOR, and addition is standard integer addition. It prints out these m rows. The program then creates k data devices each with $size$ bytes of random data and encodes them into m coding devices using **jerasure_matrix_encode()**. It prints out the data and coding in hexadecimal– one byte is represented by 2 hex digits. Next, it erases m random devices from the collection of data and coding devices, and prints the resulting state. Then it decodes the erased devices using **jerasure_matrix_decode()** and prints the restored state. Next, it shows what the decoding matrix looks like when the first m devices are erased. This matrix is the inverse of the last k rows of the distribution matrix. And finally, it uses **jerasure_matrix_dotprod()** to show how to explicitly calculate the first data device from the others when the first m devices have been erased. Here is an example for $w = 8$ with 3 data devices and 4 coding devices each with a size of 4 bytes:

```
UNIX> jerasure_05 3 4 8 4
The Coding Matrix (the last m rows of the Distribution Matrix):

 71 167 122
167  71 186
122 186  71
186 122 167

Encoding Complete:

Data                Coding
D0 : 13 e6 54 d3    C0 : 42 99 56 f0
D1 : 14 fd 2d cc    C1 : ed f2 c1 94
D2 : 1d b9 a1 a6    C2 : 80 2d 8d 6f
                    C3 : 48 bf 18 b5

Erased 4 random devices:

Data                Coding
D0 : 13 e6 54 d3    C0 : 00 00 00 00
D1 : 00 00 00 00    C1 : 00 00 00 00
D2 : 00 00 00 00    C2 : 80 2d 8d 6f
                    C3 : 48 bf 18 b5

State of the system after decoding:

Data                Coding
D0 : 13 e6 54 d3    C0 : 42 99 56 f0
D1 : 14 fd 2d cc    C1 : ed f2 c1 94
```

```
D2 : 1d b9 a1 a6    C2 : 80 2d 8d 6f
                    C3 : 48 bf 18 b5

Suppose we erase the first 4 devices.  Here is the decoding matrix:

130  25 182
252 221  25
108 252 130

And dm_ids:

  4   5   6

After calling jerasure_matrix_dotprod, we calculate the value of device 0 to be:

D0 : 13 e6 54 d3

UNIX>
```

Referring back to the conceptual model in Figure 3, it should be clear in this encoding how the first w bits of C_0 are calculated from the first w bits of each data device:

$$\text{byte 0 of } C_0 = (71 \times \text{ byte 0 of } D_0) \oplus (167 \times \text{ byte 0 of } D_1) \oplus (122 \times \text{ byte 0 of } D_2)$$

where multiplication is in $GF(2^8)$.

However, keep in mind that the implementation actually performs dot products on groups of bytes at a time. So in this example, where each device holds 4 bytes, the dot product is actually:

$$4 \text{ bytes of } C_0 = (71 \times 4 \text{ bytes of } D_0) \oplus (167 \times 4 \text{ bytes of } D_1) \oplus (122 \times 4 \text{ bytes of } D_2)$$

This is accomplished using **galois_w08_region_multiply()**.

Here is a similar example, this time with $w = 16$ and each device holding 12 bytes:

```
UNIX> jerasure_05 3 4 16 12
The Coding Matrix (the last m rows of the Distribution Matrix):

52231 20482 30723
20482 52231 27502
30723 27502 52231
27502 30723 20482

Encoding Complete:

Data                                    Coding
D0 : 13e6 54d3 14fd 2dcc 1db9 a1a6      C0 : 28a1 71c9 f807 4440 dbaa 2cc2
D1 : 60c0 cc12 36eb 7fa1 5f96 41b5      C1 : e47a ed35 3391 e02b 5e0c bcf8
D2 : b26b 4631 aa21 0772 3b47 8df8      C2 : b419 73ce af12 f7a2 a943 d33a
                                        C3 : 2a9a 3548 e424 ba90 ef76 dfe6

Erased 4 random devices:

Data                                    Coding
D0 : 13e6 54d3 14fd 2dcc 1db9 a1a6      C0 : 0000 0000 0000 0000 0000 0000
D1 : 0000 0000 0000 0000 0000 0000      C1 : e47a ed35 3391 e02b 5e0c bcf8
```

```
D2 : 0000 0000 0000 0000 0000 0000    C2 : 0000 0000 0000 0000 0000 0000
                                      C3 : 2a9a 3548 e424 ba90 ef76 dfe6

State of the system after decoding:

Data                                  Coding
D0 : 13e6 54d3 14fd 2dcc 1db9 a1a6    C0 : 28a1 71c9 f807 4440 dbaa 2cc2
D1 : 60c0 cc12 36eb 7fa1 5f96 41b5    C1 : e47a ed35 3391 e02b 5e0c bcf8
D2 : b26b 4631 aa21 0772 3b47 8df8    C2 : b419 73ce af12 f7a2 a943 d33a
                                      C3 : 2a9a 3548 e424 ba90 ef76 dfe6

Suppose we erase the first 4 devices.  Here is the decoding matrix:

  130   260   427
  252   448   260
  108   252   130

And dm_ids:

    4     5     6

After calling jerasure_matrix_dotprod, we calculate the value of device 0 to be:

D0 : 13e6 54d3 14fd 2dcc 1db9 a1a6

UNIX>
```

In this encoding, the 6 16-bit half-words of C_0 are calculated as:

$$(52231 \times 6 \text{ half-words of } D_0) \oplus (20482 \times 6 \text{ half-words of } D_1) \oplus (30723 \times 6 \text{ half-words of } D_2)$$

using **galois_w16_region_multiply()**.

This program demonstrates usage of **jerasure_matrix_encode()**, **jerasure_matrix_decode()**, **jerasure_print_matrix()**, **jerasure_make_decoding_matrix()** and **jerasure_matrix_dotprod()**.

- **jerasure_06.c**: This takes four parameters: k, m, w and *packetsize*, and performs a similar example to **jerasure_05**, except it uses Cauchy Reed-Solomon coding in $GF(2^w)$, converting the coding matrix to a bit-matrix. $k + m$ must be less than or equal to 2^w and *packetsize* must be a multiple of **sizeof(long)**. It sets up each device to hold a total of $w * packetsize$ bytes. Here, packets are numbered p_0 through p_{w-1} for each device. It then performs the same encoding and decoding as the previous example but with the corresponding bit-matrix procedures.
  Here is a run with 3 data devices and 4 coding devices with $w = 3$ and a *packetsize* of 4 bytes. (Each device will hold $3 * 4 = 12$ bytes.)

```
UNIX> jerasure_06 3 4 3 4
Last (m * w) rows of the Binary Distribution Matrix:

111 001 101
100 101 111
110 010 011

001 111 010
101 100 011
```

```
010 110 101

101 010 111
111 011 100
011 101 110

010 101 001
011 111 101
101 011 010

Encoding Complete:

Data                Coding
D0  p0 : 15ddb16e   C0  p0 : 15a44685
    p1 : 5ffcc9c0       p1 : 0655bb70
    p2 : 0c55e80a       p2 : 432bc16a
D1  p0 : 6f6b6791   C1  p0 : 617d3117
    p1 : 49e514d0       p1 : 360c93e1
    p2 : 649511f2       p2 : 4e377d52
D2  p0 : 5899d169   C2  p0 : 481b31c9
    p1 : 2f33bbae       p1 : 339d44ef
    p2 : 6fdc16ba       p2 : 2ffd3d6e
                    C3  p0 : 3bdea919
                        p1 : 26f784aa
                        p2 : 1bcbe7e8

Erased 4 random devices:

Data                Coding
D0  p0 : 00000000   C0  p0 : 00000000
    p1 : 00000000       p1 : 00000000
    p2 : 00000000       p2 : 00000000
D1  p0 : 6f6b6791   C1  p0 : 617d3117
    p1 : 49e514d0       p1 : 360c93e1
    p2 : 649511f2       p2 : 4e377d52
D2  p0 : 00000000   C2  p0 : 481b31c9
    p1 : 00000000       p1 : 339d44ef
    p2 : 00000000       p2 : 2ffd3d6e
                    C3  p0 : 00000000
                        p1 : 00000000
                        p2 : 00000000

State of the system after decoding:

Data                Coding
D0  p0 : 15ddb16e   C0  p0 : 15a44685
    p1 : 5ffcc9c0       p1 : 0655bb70
    p2 : 0c55e80a       p2 : 432bc16a
D1  p0 : 6f6b6791   C1  p0 : 617d3117
    p1 : 49e514d0       p1 : 360c93e1
    p2 : 649511f2       p2 : 4e377d52
D2  p0 : 5899d169   C2  p0 : 481b31c9
    p1 : 2f33bbae       p1 : 339d44ef
    p2 : 6fdc16ba       p2 : 2ffd3d6e
                    C3  p0 : 3bdea919
                        p1 : 26f784aa
                        p2 : 1bcbe7e8
```

```
Suppose we erase the first 4 devices.  Here is the decoding matrix:

101 011 010
111 110 011
011 111 101

001 011 011
101 110 110
010 111 111

110 001 101
001 101 111
100 010 011

And dm_ids:

4 5 6

After calling jerasure_matrix_dotprod, we calculate the value of device 0, packet 0 to be:

D0  p0 : 15ddb16e

UNIX>
```

In this encoding, the first packet of C_0 is computed according to the six ones in the first row of the coding matrix:

$$C_0p_0 = D_0p_0 \oplus D_0p_1 \oplus D_0p_2 \oplus D_1p_2 \oplus D_2p_0 \oplus D_2p_2$$

These dotproducts are accomplished with **galois_region_xor()** .

This program demonstrates usage of **jerasure_bitmatrix_encode()**, **jerasure_bitmatrix_decode()**, **jerasure_print_bitmatrix()**, **jerasure_make_decoding_bitmatrix()** and **jerasure_bitmatrix_dotprod()**.

- **jerasure_07.c**: This takes three parameters: k, m and w. It performs the same coding/decoding as in **jerasure_06**, except it uses bit-matrix scheduling instead of bit-matrix operations. The *packetsize* is set at **sizeof(long)** bytes. It creates a "dumb" and "smart" schedule for encoding, encodes with them and prints out how many XORs each took. The smart schedule will outperform the dumb one.

  Next, it erases m random devices and decodes using **jerasure_schedule_decode_lazy()**. Finally, it shows how to use **jerasure_do_scheduled_operations()** in case you need to do so explicitly.
  Example:

```
UNIX> jerasure_07 3 4 3
Last m rows of the Binary Distribution Matrix:

111 001 101
100 101 111
110 010 011

001 111 010
101 100 011
010 110 101

101 010 111
```

```
111 011 100
011 101 110

010 101 001
011 111 101
101 011 010

Dumb Encoding Complete: - 216 XOR'd bytes

Data                Coding
D0  p0 : 15ddb16e   C0  p0 : 15a44685
    p1 : 5ffcc9c0       p1 : 0655bb70
    p2 : 0c55e80a       p2 : 432bc16a
D1  p0 : 6f6b6791   C1  p0 : 617d3117
    p1 : 49e514d0       p1 : 360c93e1
    p2 : 649511f2       p2 : 4e377d52
D2  p0 : 5899d169   C2  p0 : 481b31c9
    p1 : 2f33bbae       p1 : 339d44ef
    p2 : 6fdc16ba       p2 : 2ffd3d6e
                    C3  p0 : 3bdea919
                        p1 : 26f784aa
                        p2 : 1bcbe7e8

Smart Encoding Complete: - 132 XOR'd bytes

Data                Coding
D0  p0 : 15ddb16e   C0  p0 : 15a44685
    p1 : 5ffcc9c0       p1 : 0655bb70
    p2 : 0c55e80a       p2 : 432bc16a
D1  p0 : 6f6b6791   C1  p0 : 617d3117
    p1 : 49e514d0       p1 : 360c93e1
    p2 : 649511f2       p2 : 4e377d52
D2  p0 : 5899d169   C2  p0 : 481b31c9
    p1 : 2f33bbae       p1 : 339d44ef
    p2 : 6fdc16ba       p2 : 2ffd3d6e
                    C3  p0 : 3bdea919
                        p1 : 26f784aa
                        p2 : 1bcbe7e8

Erased 4 random devices:

Data                Coding
D0  p0 : 00000000   C0  p0 : 00000000
    p1 : 00000000       p1 : 00000000
    p2 : 00000000       p2 : 00000000
D1  p0 : 6f6b6791   C1  p0 : 617d3117
    p1 : 49e514d0       p1 : 360c93e1
    p2 : 649511f2       p2 : 4e377d52
D2  p0 : 00000000   C2  p0 : 481b31c9
    p1 : 00000000       p1 : 339d44ef
    p2 : 00000000       p2 : 2ffd3d6e
                    C3  p0 : 00000000
                        p1 : 00000000
                        p2 : 00000000

State of the system after decoding: 124 XOR'd bytes
```

```
Data                Coding
D0  p0 : 15ddb16e   C0  p0 : 15a44685
    p1 : 5ffcc9c0       p1 : 0655bb70
    p2 : 0c55e80a       p2 : 432bc16a
D1  p0 : 6f6b6791   C1  p0 : 617d3117
    p1 : 49e514d0       p1 : 360c93e1
    p2 : 649511f2       p2 : 4e377d52
D2  p0 : 5899d169   C2  p0 : 481b31c9
    p1 : 2f33bbae       p1 : 339d44ef
    p2 : 6fdc16ba       p2 : 2ffd3d6e
                    C3  p0 : 3bdea919
                        p1 : 26f784aa
                        p2 : 1bcbe7e8

State of the system after deleting the coding devices and
using jerasure_do_scheduled_operations(): 124 XOR'd bytes

Data                Coding
D0  p0 : 15ddb16e   C0  p0 : 15a44685
    p1 : 5ffcc9c0       p1 : 0655bb70
    p2 : 0c55e80a       p2 : 432bc16a
D1  p0 : 6f6b6791   C1  p0 : 617d3117
    p1 : 49e514d0       p1 : 360c93e1
    p2 : 649511f2       p2 : 4e377d52
D2  p0 : 5899d169   C2  p0 : 481b31c9
    p1 : 2f33bbae       p1 : 339d44ef
    p2 : 6fdc16ba       p2 : 2ffd3d6e
                    C3  p0 : 3bdea919
                        p1 : 26f784aa
                        p2 : 1bcbe7e8

UNIX>
```

This demonstrates usage of **jerasure_dumb_bitmatrix_to_schedule()**, **jerasure_smart_bitmatrix_to_schedule()**, **jerasure_schedule_encode()**, **jerasure_schedule_decode_lazy()**, **jerasure_do_scheduled_operations()** and **jerasure_get_stats()**.

- **jerasure_08.c**: This takes two parameters: k and w, and performs a simple RAID-6 example using a schedule cache. Again, *packetsize* is **sizeof(long)**. It sets up a RAID-6 coding matrix whose first row is composed of ones, and where the element in column j of the second row is equal to 2^j in $GF(2^w)$. It converts this to a bit-matrix and creates a smart encoding schedule and a schedule cache for decoding.

  It then encodes twice – first with the smart schedule, and then with the schedule cache, by setting the two coding devices as the erased devices. Next it deletes two random devices and uses the schedule cache to decode them. Next, it deletes the first coding devices and recalculates it using **jerasure_do_parity()** to demonstrate that procedure. Finally, it frees the smart schedule and the schedule cache.

  Example:

```
UNIX> jerasure_08 5 3
Encoding Complete: - 124 XOR'd bytes

Data                Coding
D0  p0 : 15ddb16e   C0  p0 : 20b37529
    p1 : 5ffcc9c0       p1 : 217c9aaa
    p2 : 0c55e80a       p2 : 18222f93
```

```
D1  p0 : 6f6b6791    C1  p0 : 3f54690b
    p1 : 49e514d0        p1 : 23616b27
    p2 : 649511f2        p2 : 285e33c7
D2  p0 : 5899d169
    p1 : 2f33bbae
    p2 : 6fdc16ba
D3  p0 : 5f5f46b4
    p1 : 39180848
    p2 : 2d46f73b
D4  p0 : 5dc3340b
    p1 : 214ef45c
    p2 : 327837ea

Encoding Using the Schedule Cache: - 124 XOR'd bytes

Data                 Coding
D0  p0 : 15ddb16e    C0  p0 : 20b37529
    p1 : 5ffcc9c0        p1 : 217c9aaa
    p2 : 0c55e80a        p2 : 18222f93
D1  p0 : 6f6b6791    C1  p0 : 3f54690b
    p1 : 49e514d0        p1 : 23616b27
    p2 : 649511f2        p2 : 285e33c7
D2  p0 : 5899d169
    p1 : 2f33bbae
    p2 : 6fdc16ba
D3  p0 : 5f5f46b4
    p1 : 39180848
    p2 : 2d46f73b
D4  p0 : 5dc3340b
    p1 : 214ef45c
    p2 : 327837ea

Erased 2 random devices:

Data                 Coding
D0  p0 : 15ddb16e    C0  p0 : 20b37529
    p1 : 5ffcc9c0        p1 : 217c9aaa
    p2 : 0c55e80a        p2 : 18222f93
D1  p0 : 00000000    C1  p0 : 3f54690b
    p1 : 00000000        p1 : 23616b27
    p2 : 00000000        p2 : 285e33c7
D2  p0 : 5899d169
    p1 : 2f33bbae
    p2 : 6fdc16ba
D3  p0 : 00000000
    p1 : 00000000
    p2 : 00000000
D4  p0 : 5dc3340b
    p1 : 214ef45c
    p2 : 327837ea

State of the system after decoding: 124 XOR'd bytes

Data                 Coding
D0  p0 : 15ddb16e    C0  p0 : 20b37529
    p1 : 5ffcc9c0        p1 : 217c9aaa
    p2 : 0c55e80a        p2 : 18222f93
```

```
D1  p0 : 6f6b6791    C1  p0 : 3f54690b
    p1 : 49e514d0        p1 : 23616b27
    p2 : 649511f2        p2 : 285e33c7
D2  p0 : 5899d169
    p1 : 2f33bbae
    p2 : 6fdc16ba
D3  p0 : 5f5f46b4
    p1 : 39180848
    p2 : 2d46f73b
D4  p0 : 5dc3340b
    p1 : 214ef45c
    p2 : 327837ea

State of the system after deleting coding device 0 and using
jerasure_do_parity to re-encode it:

Data                 Coding
D0  p0 : 15ddb16e    C0  p0 : 20b37529
    p1 : 5ffcc9c0        p1 : 217c9aaa
    p2 : 0c55e80a        p2 : 18222f93
D1  p0 : 6f6b6791    C1  p0 : 3f54690b
    p1 : 49e514d0        p1 : 23616b27
    p2 : 649511f2        p2 : 285e33c7
D2  p0 : 5899d169
    p1 : 2f33bbae
    p2 : 6fdc16ba
D3  p0 : 5f5f46b4
    p1 : 39180848
    p2 : 2d46f73b
D4  p0 : 5dc3340b
    p1 : 214ef45c
    p2 : 327837ea

Smart schedule and cache freed

UNIX>
```

This demonstrates usage of **jerasure_generate_schedule_cache**(), **jerasure_smart_bitmatrix_to_schedule**(), **jerasure_schedule_encode**(), **jerasure_schedule_decode_cache**(), **jerasure_free_schedule**(), **jerasure_free_schedule_cache**(), **jerasure_get_stats**() and **jerasure_do_parity**().

8 Part 3 of the Library: Classic Reed-Solomon Coding Routines

The files **reed_sol.h** and **reed_sol.c** implement procedures that are specific to classic Vandermonde matrix-based Reed-Solomon coding, and for Reed-Solomon coding optimized for RAID-6. Refer to [Pla97, PD05] for a description of classic Reed-Solomon coding and to [Anv07] for Reed-Solomon coding optimized for RAID-6. Where not specified, the parameters are as described in Section 7.

8.1 Vandermonde Distribution Matrices

There are three procedures for generating distribution matrices based on an extended Vandermonde matrix in $GF(2^w)$. It is anticipated that only the first of these will be needed for coding applications, but we include the other two in case a user wants to look at or modify these matrices.

- **int *reed_sol_vandermonde_coding_matrix(k, m, w)**: This returns the last m rows of the distribution matrix in $GF(2^w)$, based on an extended Vandermonde matrix. This is a $m \times k$ matrix that can be used with the matrix routines in **jerasure.c**. The first row of this matrix is guaranteed to be all ones. The first column is also guaranteed to be all ones.
- **int *reed_sol_extended_vandermonde_matrix(int rows, int cols, w)**: This creates an extended Vandermonde matrix with **rows** rows and **cols** columns in $GF(2^w)$.
- **int *reed_sol_big_vandermonde_distribution_matrix(int rows, int cols, w)**: This converts the extended matrix above into a distribution matrix so that the top **cols** rows compose an identity matrix, and the remaining rows are in the format returned by **reed_sol_vandermonde_coding_matrix()**.

8.2 Procedures Related to Reed-Solomon Coding Optimized for RAID-6

In RAID-6, m is equal to two. The first coding device, P is calculated from the others using parity, and the second coding device, Q is calculated from the data devices D_i using:

$$Q = \sum_{i=0}^{k-1} 2^i D_i$$

where all arithmetic is in $GF(2^w)$. The reason that this is an optimization is that one may implement multiplication by two in an optimized fashion. The following procedures facilitate this optimization.

- **int reed_sol_r6_encode(k, w, data_ptrs, coding_ptrs, size)**: This encodes using the optimization. w must be 8, 16 or 32. Note, m is not needed because it is assumed to equal two, and no matrix is needed because it is implicit.
- **int *reed_sol_r6_coding_matrix(k, w)**: Again, w must be 8, 16 or 32. There is no optimization for decoding. Therefore, this procedure returns the last two rows of the distribution matrix for RAID-6 for decoding purposes. The first of these rows will be all ones. The second of these rows will have 2^j in column j.
- **reed_sol_galois_w08_region_multby_2(char *region, int nbytes)**: This performs the fast multiplication by two in $GF(2^8)$ using Anvin's optimization [Anv07]. **region** must be long-word aligned, and **nbytes** must be a multiple of the word size.
- **reed_sol_galois_w16_region_multby_2(char *region, int nbytes)**: This performs the fast multiplication by two in $GF(2^{16})$.
- **reed_sol_galois_w32_region_multby_2(char *region, int nbytes)**: This performs the fast multiplication by two in $GF(2^{32})$.

8.3 Example Programs to Demonstrate Use

There are four example programs to demonstrate the use of the procedures in **reed_sol**.

- **reed_sol_01.c**: This takes three parameters: k, m and w. It performs a classic Reed-Solomon coding of k devices onto m devices, using a Vandermonde-based distribution matrix in $GF(2^w)$. w must be 8, 16 or 32. Each device is set up to hold **sizeof(long)** bytes. It uses **reed_sol_vandermonde_coding_matrix()** to generate the distribution matrix, and then procedures from **jerasure.c** to perform the coding and decoding.
  Example:

```
UNIX> reed_sol_01 7 7 8
Last m rows of the Distribution Matrix:

  1   1   1   1   1   1   1
  1 199 210 240 105 121 248
  1  70  91 245  56 142 167
  1 170 114  42  87  78 231
  1  38 236  53 233 175  65
  1  64 174 232  52 237  39
  1 187 104 210 211 105 186

Encoding Complete:

Data                Coding
D0 : 6e b1 dd 15    C0 : 7e 23 f6 5c
D1 : c0 c9 fc 5f    C1 : 4e 9a cc 16
D2 : 0a e8 55 0c    C2 : eb 3b 82 7d
D3 : 91 67 6b 6f    C3 : 3f e0 c8 71
D4 : d0 14 e5 49    C4 : 0f 7b 21 eb
D5 : f2 11 95 64    C5 : ca d1 11 5d
D6 : 69 d1 99 58    C6 : 6b e1 0e 16

Erased 7 random devices:

Data                Coding
D0 : 00 00 00 00    C0 : 00 00 00 00
D1 : c0 c9 fc 5f    C1 : 4e 9a cc 16
D2 : 00 00 00 00    C2 : 00 00 00 00
D3 : 91 67 6b 6f    C3 : 00 00 00 00
D4 : 00 00 00 00    C4 : 0f 7b 21 eb
D5 : f2 11 95 64    C5 : ca d1 11 5d
D6 : 00 00 00 00    C6 : 6b e1 0e 16

State of the system after decoding:

Data                Coding
D0 : 6e b1 dd 15    C0 : 7e 23 f6 5c
D1 : c0 c9 fc 5f    C1 : 4e 9a cc 16
D2 : 0a e8 55 0c    C2 : eb 3b 82 7d
D3 : 91 67 6b 6f    C3 : 3f e0 c8 71
D4 : d0 14 e5 49    C4 : 0f 7b 21 eb
D5 : f2 11 95 64    C5 : ca d1 11 5d
D6 : 69 d1 99 58    C6 : 6b e1 0e 16

UNIX>
```

This demonstrates usage of **jerasure_matrix_encode()**, **jerasure_matrix_decode()**, **jerasure_print_matrix()** and **reed_sol_vandermonde_coding_matrix()**.

- **reed_sol_02.c**: This takes three parameters: k, m and w. It creates and prints three matrices in $GF(2^w)$:

  1. A $(k+m) \times k$ extended Vandermonde matrix.
  2. The $(k+m) \times k$ distribution matrix created by converting the extended Vandermonde matrix into one where the first k rows are an identity matrix. Then row k is converted so that it is all ones, and the first column is also converted so that it is all ones.

3. The $m \times k$ coding matrix, which is last m rows of the above matrix. This is the matrix which is passed to the encoding/decoding procedures of **jerasure.c**. Note that since the first row of this matrix is all ones, you may set **int row_k_ones** of the decoding procedures to one.

Note also that w may have any value from 1 to 32.
Example:

```
UNIX> reed_sol_02 6 4 11
Extended Vandermonde Matrix:

   1    0    0    0    0    0
   1    1    1    1    1    1
   1    2    4    8   16   32
   1    3    5   15   17   51
   1    4   16   64  256 1024
   1    5   17   85  257 1285
   1    6   20  120  272 1632
   1    7   21  107  273 1911
   1    8   64  512   10   80
   0    0    0    0    0    1

Vandermonde Distribution Matrix:

   1    0    0    0    0    0
   0    1    0    0    0    0
   0    0    1    0    0    0
   0    0    0    1    0    0
   0    0    0    0    1    0
   0    0    0    0    0    1
   1    1    1    1    1    1
   1 1879 1231 1283  682 1538
   1 1366 1636 1480  683  934
   1 1023 2045 1027 2044 1026

Vandermonde Coding Matrix:

   1    1    1    1    1    1
   1 1879 1231 1283  682 1538
   1 1366 1636 1480  683  934
   1 1023 2045 1027 2044 1026

UNIX>
```

This demonstrates usage of **reed_sol_extended_vandermonde_matrix()**, **reed_sol_big_vandermonde_coding_matrix()**, **reed_sol_vandermonde_coding_matrix()** and **jerasure_print_matrix()**.

- **reed_sol_03.c**: This takes two parameters: k and w. It performs RAID-6 coding using Anvin's optimization [Anv07] in $GF(2^w)$, where w must be 8, 16 or 32. It then decodes using **jerasure_matrix_decode()**.
  Example:

```
UNIX> reed_sol_03 9 8
Last 2 rows of the Distribution Matrix:

  1   1   1   1   1   1   1   1   1
  1   2   4   8  16  32  64 128  29
```

```
Encoding Complete:

Data                Coding
D0 : 6e b1 dd 15    C0 : 6a 8e 19 1c
D1 : c0 c9 fc 5f    C1 : e1 c3 fa 8e
D2 : 0a e8 55 0c
D3 : 91 67 6b 6f
D4 : d0 14 e5 49
D5 : f2 11 95 64
D6 : 69 d1 99 58
D7 : ae bb 33 2f
D8 : ba 16 dc 6f

Erased 2 random devices:

Data                Coding
D0 : 6e b1 dd 15    C0 : 6a 8e 19 1c
D1 : c0 c9 fc 5f    C1 : e1 c3 fa 8e
D2 : 0a e8 55 0c
D3 : 00 00 00 00
D4 : 00 00 00 00
D5 : f2 11 95 64
D6 : 69 d1 99 58
D7 : ae bb 33 2f
D8 : ba 16 dc 6f

State of the system after decoding:

Data                Coding
D0 : 6e b1 dd 15    C0 : 6a 8e 19 1c
D1 : c0 c9 fc 5f    C1 : e1 c3 fa 8e
D2 : 0a e8 55 0c
D3 : 91 67 6b 6f
D4 : d0 14 e5 49
D5 : f2 11 95 64
D6 : 69 d1 99 58
D7 : ae bb 33 2f
D8 : ba 16 dc 6f

UNIX>
```

This demonstrates usage of **reed_sol_r6_encode()**, **reed_sol_r6_coding_matrix()**, **jerasure_matrix_decode()** and **jerasure_print_matrix()**.

- **reed_sol_04.c**: This simply demonstrates doing fast multiplication by two in $GF(2^w)$ for $w \in \{8, 16, 32\}$. It has one parameter: w.

```
UNIX> reed_sol_04 16
Short  0:  8562 *2 = 17124
Short  1: 11250 *2 = 22500
Short  2: 16429 *2 = 32858
Short  3: 37513 *2 = 13593
Short  4: 57579 *2 = 53725
Short  5: 24136 *2 = 48272
Short  6: 59268 *2 = 57091
Short  7: 41368 *2 = 21307
```

```
UNIX>
```

This demonstrates usage of **reed_sol_galois_w08_region_multby_2**(), **reed_sol_galois_w16_region_multby_2**() and **reed_sol_galois_w32_region_multby_2**().

9 Part 4 of the Library: Cauchy Reed-Solomon Coding Routines

The files **cauchy.h** and **cauchy.c** implement procedures that are specific to Cauchy Reed-Solomon coding. See [BKK +95, PX06] for detailed descriptions of this kind of coding. The procedures in **jerasure.h/jerasure.c** do the coding and decoding. The procedures here simply create coding matrices. We don't use the Cauchy matrices described in [PX06], because there is a simple heuristic that creates better matrices:

- Construct the usual Cauchy matrix M such that $M[i,j] = \frac{1}{i \oplus (m+j)}$, where division is over $GF(2^w)$, $\oplus$ is XOR and the addition is regular integer addition.
- For each column j, divide each element (in $GF(2^w)$) by $M[0,j]$. This has the effect of turning each element in row 0 to one.
- Next, for each row $i > 0$ of the matrix, do the following:
  - Count the number of ones in the bit representation of the row.
  - Count the number of ones in the bit representation of the row divided by element $M[i,j]$ for each j.
  - Whichever value of j gives the minimal number of ones, if it improves the number of ones in the original row, divide row i by $M[i,j]$.

While this does not guarantee an optimal number of ones, it typically generates a good matrix. For example, suppose $k = m = w = 3$. The matrix M is as follows:

$$\begin{vmatrix} 6 & 7 & 2 \\ 5 & 2 & 7 \\ 1 & 3 & 4 \end{vmatrix}$$

First, we divide column 0 by 6, column 1 by 7 and column 2 by 2, to yield:

$$\begin{vmatrix} 1 & 1 & 1 \\ 4 & 3 & 6 \\ 3 & 7 & 2 \end{vmatrix}$$

Now, we concentrate on row 1. Its bitmatrix representation has 5+7+7 = 19 ones. If we divide it by 4, the bitmatrix has 3+4+5 = 12 ones. If we divide it by 3, the bitmatrix has 4+3+4 = 11 ones. If we divide it by 6, the bitmatrix has 6+7+3 = 16 ones. So, we replace row 1 with row 1 divided by 3.

We do the same with row 2 and find that it will have the minimal number of ones when it is divided by three. The final matrix is:

$$\begin{vmatrix} 1 & 1 & 1 \\ 5 & 1 & 2 \\ 1 & 4 & 7 \end{vmatrix}$$

This matrix has 34 ones, a distinct improvement over the original matrix that has 46 ones. The best matrix in [PX06] has 39 ones. This is because the authors simply find the best X and Y, and do not modify the matrix after creating it.

9.1 The Procedures in cauchy.c

The procedures are:

- **int *cauchy_original_coding_matrix(k, m, w)**: This allocates and returns the originally defined Cauchy matrix from [BKK+95]. This is the same matrix as defined above: $M[i, j] = \frac{1}{i \oplus (m+j)}$.
- **int *cauchy_xy_coding_matrix(k, m, w, int *X, int *Y)**: This allows the user to specify sets X and Y to define the matrix. Set X has m elements of $GF(2^w)$ and set Y has k elements. Neither set may have duplicate elements and $X \cap Y = \emptyset$. The procedure does not double-check X and Y - it assumes that they conform to these restrictions.
- **void cauchy_improve_coding_matrix(k, m, w, matrix)**: This improves a matrix using the heuristic above, first dividing each column by its element in row 0, then improving the rest of the rows.
- **int *cauchy_good_general_coding_matrix()**: This allocates and returns a good matrix. When $m = 2$, $w \leq 11$ and $k \leq 1023$, it will return the optimal RAID-6 matrix. Otherwise, it generates a good matrix by calling **cauchy_original_coding_matrix()** and then **cauchy_improve_coding_matrix()**. If you need to generate RAID-6 matrices that are beyond the above parameters, see Section 9.3 below.
- **int cauchy_n_ones(int n, w)**: This returns the number of ones in the bit-matrix representation of the number n in $GF(2^w)$. It is much more efficient than generating the bit-matrix and counting ones.

9.2 Example Programs to Demonstrate Use

There are four example programs to demonstrate the use of the procedures in **cauchy.h/cauchy.c**.

- **cauchy_01.c**: This takes two parameters: n and w. It calls **cauchy_n_ones()** to determine the number of ones in the bit-matrix representation of n in $GF(2^w)$. Then it converts n to a bit-matrix, prints it and confirms the number of ones:

```
UNIX> cauchy_01 01 5
# Ones: 5

Bitmatrix has 5 ones

10000
01000
00100
00010
00001
UNIX> cauchy_01 31 5
# Ones: 16

Bitmatrix has 16 ones

11110
11111
10001
11000
11100
UNIX>
```

This demonstrates usage of **cauchy_n_ones()**, **jerasure_matrix_to_bitmatrix()** and **jerasure_print_bitmatrix()**.

- **cauchy_02.c**: This takes three parameters: k, m and w. (In this and the following examples, *packetsize* is **sizeof(long)**.) It calls **cauchy_original_coding_matrix()** to create an Cauchy matrix, converts it to a bit-matrix then encodes and decodes with it. Smart scheduling is employed. Lastly, it uses **cauchy_xy_coding_matrix()** to create the same Cauchy matrix. It verifies that the two matrices are indeed identical.

  Example:

```
UNIX> cauchy_02 3 3 3
Matrix has 46 ones

6 7 2
5 2 7
1 3 4

Smart Encoding Complete: - 112 XOR'd bytes

Data                 Coding
D0  p0 : 15ddb16e    C0  p0 : 7e6e55c3
    p1 : 5ffcc9c0        p1 : 120fd8ec
    p2 : 0c55e80a        p2 : 4fc9584b
D1  p0 : 6f6b6791    C1  p0 : 36c21521
    p1 : 49e514d0        p1 : 5f324f00
    p2 : 649511f2        p2 : 2b92cf79
D2  p0 : 5899d169    C2  p0 : 31107ca3
    p1 : 2f33bbae        p1 : 5d080667
    p2 : 6fdc16ba        p2 : 16602afb

Erased 3 random devices:

Data                 Coding
D0  p0 : 15ddb16e    C0  p0 : 00000000
    p1 : 5ffcc9c0        p1 : 00000000
    p2 : 0c55e80a        p2 : 00000000
D1  p0 : 00000000    C1  p0 : 36c21521
    p1 : 00000000        p1 : 5f324f00
    p2 : 00000000        p2 : 2b92cf79
D2  p0 : 00000000    C2  p0 : 31107ca3
    p1 : 00000000        p1 : 5d080667
    p2 : 00000000        p2 : 16602afb

State of the system after decoding: 96 XOR'd bytes

Data                 Coding
D0  p0 : 15ddb16e    C0  p0 : 7e6e55c3
    p1 : 5ffcc9c0        p1 : 120fd8ec
    p2 : 0c55e80a        p2 : 4fc9584b
D1  p0 : 6f6b6791    C1  p0 : 36c21521
    p1 : 49e514d0        p1 : 5f324f00
    p2 : 649511f2        p2 : 2b92cf79
D2  p0 : 5899d169    C2  p0 : 31107ca3
    p1 : 2f33bbae        p1 : 5d080667
    p2 : 6fdc16ba        p2 : 16602afb

Generated the identical matrix using cauchy_xy_coding_matrix()
UNIX>
```

This demonstrates usage of **cauchy_original_coding_matrix()**, **cauchy_xy_coding_matrix()**, **cauchy_n_ones()**, **jerasure_smart_bitmatrix_to_schedule()**, **jerasure_schedule_encode()**, **jerasure_schedule_decode_lazy()**, **jerasure_print_matrix()** and **jerasure_get_stats()**.

- **cauchy_03.c**: This is identical to **cauchy_02.c**, except that it improves the matrix with **cauchy_improve_coding_matrix()**.
  Example:

```
UNIX> cauchy_03 3 3 3 | head -n 8
The Original Matrix has 46 ones
The Improved Matrix has 34 ones

1 1 1
5 1 2
1 4 7

Smart Encoding Complete: - 96 XOR'd bytes
UNIX>
```

  This demonstrates usage of **cauchy_original_coding_matrix()**, **cauchy_improve_coding_matrix()**, **cauchy_n_ones()**, **jerasure_smart_bitmatrix_to_schedule()**, **jerasure_schedule_encode()**, **jerasure_schedule_decode_lazy()**, **jerasure_print_matrix()** and **jerasure_get_stats()**.

- **cauchy_04.c**: Finally, this is identical to the previous two, except it calls **cauchy_good_general_coding_matrix()**. Note, when $m = 2$, $w \leq 11$ and $k \leq 1023$, these are optimal Cauchy encoding matrices. That's not to say that they are optimal RAID-6 matrices (RDP encoding [CEG+04], and Liberation encoding [Pla08b] achieve this), but they are the best Cauchy matrices.

```
UNIX> cauchy_04 10 2 8 | head -n 6
Matrix has 229 ones

  1   1   1   1   1   1   1   1   1   1
  1   2 142   4  71   8  70 173   3  35

Smart Encoding Complete: - 836 XOR'd bytes
UNIX> cauchy_03 10 2 8 | head -n 6
The Original Matrix has 608 ones
The Improved Matrix has 354 ones

  1   1   1   1   1   1   1   1   1   1
 82 200 151 172   1 225 166 158  44  13

UNIX> cauchy_02 10 2 8 | head -n 6
Matrix has 608 ones

142 244  71 167 122 186 173 157 221 152
244 142 167  71 186 122 157 173 152 221

Smart Encoding Complete: - 1876 XOR'd bytes
UNIX>
```

  This demonstrates usage of **cauchy_original_coding_matrix()**, **cauchy_n_ones()**, **jerasure_smart_bitmatrix_to_schedule()**, **jerasure_schedule_encode()**, **jerasure_schedule_decode_lazy()**, **jerasure_print_matrix()** and **jerasure_get_stats()**.

9.3 Extending the Parameter Space for Optimal Cauchy RAID-6 Matrices

It is easy to prove that as long as $k < 2^w$, then any matrix with all ones in row 0 and distinct non-zero elements in row 1 is a valid MDS RAID-6 matrix. Therefore, the best RAID-6 matrix for a given value of w is one whose k elements in row 1 are the k elements with the smallest number of ones in their bit-matrices. **Cauchy.c** stores these elements in global variables for $k \leq 1023$ and $w \leq 11$. The file **cauchy_best_r6.c** is identical to **cauchy.c** except that it includes these values for $w \leq 32$. You will likely get compilation warnings when you use this file, but in my tests, all runs fine. The reason that these values are not in **cauchy.c** is simply to keep the object files small.

10 Part 5 of the Library: Minimal Density RAID-6 Coding

Minimal Density RAID-6 codes are MDS codes based on binary matrices which satisfy a lower-bound on the number of non-zero entries. Unlike Cauchy coding, the bit-matrix elements do not correspond to elements in $GF(2^w)$. Instead, the bit-matrix itself has the proper MDS property. Minimal Density RAID-6 codes perform faster than Reed-Solomon and Cauchy Reed-Solomon codes for the same parameters. Liberation coding, Liber8tion coding, and Blaum-Roth coding are three examples of this kind of coding that are supported in **jerasure**.

With each of these codes, m must be equal to two and k must be less than or equal to w. The value of w has restrictions based on the code:

- With Liberation coding, w must be a prime number [Pla08b].
- With Blaum-Roth coding, $w + 1$ must be a prime number [BR99].
- With Liber8tion coding, w must equal 8 [Pla08a].

The files **liberation.h** and **liberation.c** implement the following procedures:

- **int *liberation_coding_bitmatrix(k, w)**: This allocates and returns the bit-matrix for liberation coding. Although w must be a prime number greater than 2, this is not enforced by the procedure. If you give it a non-prime w, you will get a non-MDS coding matrix.
- **int *liber8tion_coding_bitmatrix(int k)**: This allocates and returns the bit-matrix for liber8tion coding. There is no w parameter because w must equal 8.
- **int *blaum_roth_coding_bitmatrix(int k, int w)**: This allocates and returns the bit-matrix for Blaum Roth coding. As above, although w+1 must be a prime number, this is not enforced.

10.1 Example Program to Demonstrate Use

liberation_01.c: This takes two parameters: k and w, where w should be a prime number greater than two and k must be less than or equal to w. As in other examples, *packetsize* is **sizeof(long)**. It sets up a Liberation bit-matrix and uses it for encoding and decoding. It encodes by converting the bit-matrix to a dumb schedule. The dumb schedule is used because that schedule cannot be improved upon. For decoding, smart scheduling is used as it gives a big savings over dumb scheduling.

```
UNIX> liberation_01 3 7
Coding Bit-Matrix:

1000000 1000000 1000000
```

```
0100000 0100000 0100000
0010000 0010000 0010000
0001000 0001000 0001000
0000100 0000100 0000100
0000010 0000010 0000010
0000001 0000001 0000001

1000000 0100000 0010000
0100000 0010000 0001000
0010000 0001000 0000100
0001000 0001100 0000010
0000100 0000010 0000001
0000010 0000001 1000000
0000001 1000000 1100000

Smart Encoding Complete: - 120 XOR'd bytes

Data                Coding
D0  p0 : 15ddb16e   C0  p0 : 08963d2a
    p1 : 5ffcc9c0       p1 : 534d051c
    p2 : 0c55e80a       p2 : 3f20b23d
    p3 : 6f6b6791       p3 : 1fc00258
    p4 : 49e514d0       p4 : 3f352723
    p5 : 649511f2       p5 : 33c9c7ec
    p6 : 5899d169       p6 : 438f5f67
D1  p0 : 2f33bbae   C1  p0 : 162bbb57
    p1 : 6fdc16ba       p1 : 4910e2f5
    p2 : 5f5f46b4       p2 : 6edb248a
    p3 : 39180848       p3 : 71aa7af7
    p4 : 2d46f73b       p4 : 2e7e5a89
    p5 : 5dc3340b       p5 : 77a3d244
    p6 : 214ef45c       p6 : 26bf874b
D2  p0 : 327837ea
    p1 : 636dda66
    p2 : 6c2a1c83
    p3 : 49b36d81
    p4 : 5b96c4c8
    p5 : 0a9fe215
    p6 : 3a587a52

Erased 2 random devices:

Data                Coding
D0  p0 : 00000000   C0  p0 : 08963d2a
    p1 : 00000000       p1 : 534d051c
    p2 : 00000000       p2 : 3f20b23d
    p3 : 00000000       p3 : 1fc00258
    p4 : 00000000       p4 : 3f352723
    p5 : 00000000       p5 : 33c9c7ec
    p6 : 00000000       p6 : 438f5f67
D1  p0 : 00000000   C1  p0 : 162bbb57
    p1 : 00000000       p1 : 4910e2f5
    p2 : 00000000       p2 : 6edb248a
    p3 : 00000000       p3 : 71aa7af7
    p4 : 00000000       p4 : 2e7e5a89
    p5 : 00000000       p5 : 77a3d244
    p6 : 00000000       p6 : 26bf874b
```

```
D2  p0 : 327837ea
    p1 : 636dda66
    p2 : 6c2a1c83
    p3 : 49b36d81
    p4 : 5b96c4c8
    p5 : 0a9fe215
    p6 : 3a587a52

State of the system after decoding: 120 XOR'd bytes

Data                Coding
D0  p0 : 15ddb16e   C0  p0 : 08963d2a
    p1 : 5ffcc9c0       p1 : 534d051c
    p2 : 0c55e80a       p2 : 3f20b23d
    p3 : 6f6b6791       p3 : 1fc00258
    p4 : 49e514d0       p4 : 3f352723
    p5 : 649511f2       p5 : 33c9c7ec
    p6 : 5899d169       p6 : 438f5f67
D1  p0 : 2f33bbae   C1  p0 : 162bbb57
    p1 : 6fdc16ba       p1 : 4910e2f5
    p2 : 5f5f46b4       p2 : 6edb248a
    p3 : 39180848       p3 : 71aa7af7
    p4 : 2d46f73b       p4 : 2e7e5a89
    p5 : 5dc3340b       p5 : 77a3d244
    p6 : 214ef45c       p6 : 26bf874b
D2  p0 : 327837ea
    p1 : 636dda66
    p2 : 6c2a1c83
    p3 : 49b36d81
    p4 : 5b96c4c8
    p5 : 0a9fe215
    p6 : 3a587a52

UNIX>
```

This demonstrates usage of **liberation_coding_bitmatrix()**, **jerasure_dumb_bitmatrix_to_schedule()**, **jerasure_schedule_encode()**, **jerasure_schedule_decode_lazy()**, **jerasure_print_bitmatrix()** and **jerasure_get_stats()**.

11 Example Encoder and Decoder

- **encoder.c**: This program is used to encode a file using any of the available methods in **jerasure**. It takes seven parameters:
  - *inputfile* or negative number S: either the file to be encoded or a negative number S indicating that a random file of size $-S$ should be used rather than an existing file
  - k: number of data files
  - m: number of coding files
  - *coding technique*: must be one of the following:
    - reed_sol_van: calls **reed_sol_vandermonde_coding_matrix()** and **jerasure_matrix_encode()**
    - reed_sol_r6_op: calls **reed_sol_r6_encode()**

  - cauchy_orig: calls **cauchy_original_coding_matrix()**, **jerasure_matrix_to_bitmatrix**, **jerasure_smart_bitmatrix_to_schedule**, and **jerasure_schedule_encode()**
  - cauchy_good: calls **cauchy_good_general_coding_matrix()**, **jerasure_matrix_to_bitmatrix**, **jerasure_smart_bitmatrix_to_schedule**, and **jerasure_schedule_encode()**
  - liberation: calls **liberation_coding_bitmatrix**, **jerasure_smart_bitmatrix_to_schedule**, and **jerasure_schedule_encode()**
  - blaum_roth: calls **blaum_roth_coding_bitmatrix**, **jerasure_smart_bitmatrix_to_schedule**, and **jerasure_schedule_encode()**
  - liber8tion: calls **liber8tion_coding_bitmatrix**, **jerasure_smart_bitmatrix_to_schedule**, and **jerasure_schedule_encode()**
- w: word size
- *packetsize*: can be set to 0 if not required by the selected coding method
- *buffersize*: approximate size of data (in bytes) to be read in at a time; will be adjusted to obtain a proper multiple and can be set to 0 if desired

This program reads in *inputfile* (or creates random data), breaks the file into k blocks, and encodes the file into m blocks. It also creates a metadata file to be used for decoding purposes. It writes all of these into a directory named **Coding**. The output of this program is the rate at which the above functions run and the total rate of running of the program, both given in MB/sec.

```
UNIX> ls -l Movie.wmv
-rwxr-xr-x   1 plank  plank  55211097 Aug 14 10:52 Movie.wmv
UNIX> encoder Movie.wmv 6 2 liberation 7 1024 500000
Encoding (MB/sec): 1405.3442614500
En_Total (MB/sec): 5.8234765527
UNIX> ls -l Coding
total 143816
-rw-r--r--   1 plank  plank  9203712 Aug 14 10:54 Movie_k1.wmv
-rw-r--r--   1 plank  plank  9203712 Aug 14 10:54 Movie_k2.wmv
-rw-r--r--   1 plank  plank  9203712 Aug 14 10:54 Movie_k3.wmv
-rw-r--r--   1 plank  plank  9203712 Aug 14 10:54 Movie_k4.wmv
-rw-r--r--   1 plank  plank  9203712 Aug 14 10:54 Movie_k5.wmv
-rw-r--r--   1 plank  plank  9203712 Aug 14 10:54 Movie_k6.wmv
-rw-r--r--   1 plank  plank  9203712 Aug 14 10:54 Movie_m1.wmv
-rw-r--r--   1 plank  plank  9203712 Aug 14 10:54 Movie_m2.wmv
-rw-r--r--   1 plank  plank       54 Aug 14 10:54 Movie_meta.txt
UNIX> echo "" | awk '{ print 9203712*6 }'
55222272
UNIX>
```

In the above example a 52.7 MB movie file is broken into six data and two coding blocks using Liberation codes with $w = 7$ and *packetsize* of 1K. A buffer of 500000 bytes is specified but **encoder** modifies the buffer size so that it is a multiple of $w * packetsize$ $(7 * 1024)$.

The new directory, **Coding**, contains the six files **Movie_k1.wmv** through **Movie_k6.wmv** (which are parts of the original file) plus the two encoded files **Movie_m1.wmv** and **Movie_m2.wmv**. Note that the file sizes are multiples of 7 and 1024 as well – the original file was padded with zeros so that it would encode properly. The metadata file, **Movie_meta.txt** contains all information relevant to **decoder**.

- **decoder.c**: This program is used in conjunction with **encoder** to decode any files remaining after erasures and reconstruct the original file. The only parameter for **decoder** is *inputfile*, the original file that was encoded. This file does not have to exist; the file name is needed only to find files created by **encoder**, which should be in the **Coding** directory.

  After some number of erasures, the program locates the surviving files from **encoder** and recreates the original file if at least k of the files still exist. The rate of decoding and the total rate of running the program are given as output.

  Continuing the previous example, suppose that Movie_k2.wmv and Movie_m1.wmv are erased.

```
UNIX> rm Coding/Movie_k1.wmv Coding/Movie_k2.wmv
UNIX> mv Movie.wmv Old-Movie.wmv
UNIX> decoder Movie.wmv
Decoding (MB/sec): 1167.8230894030
De_Total (MB/sec): 16.0071713224

UNIX> ls -l Coding
total 215704
-rw-r--r--   1 plank  plank  55211097 Aug 14 11:02 Movie_decoded.wmv
-rw-r--r--   1 plank  plank   9203712 Aug 14 10:54 Movie_k3.wmv
-rw-r--r--   1 plank  plank   9203712 Aug 14 10:54 Movie_k4.wmv
-rw-r--r--   1 plank  plank   9203712 Aug 14 10:54 Movie_k5.wmv
-rw-r--r--   1 plank  plank   9203712 Aug 14 10:54 Movie_k6.wmv
-rw-r--r--   1 plank  plank   9203712 Aug 14 10:54 Movie_m1.wmv
-rw-r--r--   1 plank  plank   9203712 Aug 14 10:54 Movie_m2.wmv
-rw-r--r--   1 plank  plank        54 Aug 14 10:54 Movie_meta.txt
UNIX> diff Coding/Movie_decoded.wmv Old-Movie.wmv
UNIX>
```

  This reads in all of the remaining files and creates **Movie_decoded.wmv** which, as shown by the **diff** command, is identical to the original **Movie.wmv**. Note that **decoder** does not recreate the lost data files – just the original.

11.1 Judicious Selection of Buffer and Packet Sizes

In our tests, the buffer and packet sizes have as much impact on performance as the code used. Initial performance results are in [SP08]; however these will be fleshed out more thoroughly. To give a compelling example, look at the following coding times for a randomly created 256M file on a MacBook Pro (2.16 GHz processor, 32KB L1 cache, 2MB L2 cache):

```
UNIX> encoder -268435456 6 2 liberation 7 1024 50000000
Encoding (MB/sec): 172.6522847357
En_Total (MB/sec): 141.8509585895
UNIX> encoder -268435456 6 2 liberation 7 1024 5000000
Encoding (MB/sec): 1066.8065148470
En_Total (MB/sec): 526.1761192874
UNIX> encoder -268435456 6 2 liberation 7 10240 5000000
Encoding (MB/sec): 1084.6304288755
En_Total (MB/sec): 555.2568545979
UNIX> encoder -268435456 6 2 liberation 7 102400 5000000
Encoding (MB/sec): 943.4553565388
En_Total (MB/sec): 525.2790538399
UNIX>
```

When using these routines, one should pay attention to packet and buffer sizes.

References

[Anv07] H. P. Anvin. The mathematics of RAID-6. `http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf`, 2007.

[BBBM95] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computing*, 44(2):192– 202, February 1995.

[BKK+95] J. Blomer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute, August 1995.

[BR99] M. Blaum and R. M. Roth. On lowest density MDS codes. *IEEE Transactions on Information Theory*, 45(1):46–59, January 1999.

[CEG+04] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row diagonal parity for double disk failure correction. In *4th Usenix Conference on File and Storage Technologies*, San Francisco, CA, March 2004.

[FDBS05a] G. Feng, R. Deng, F. Bao, and J. Shen. New efficient MDS array codes for RAID Part I: Reed-Solomon-like codes for tolerating three disk failures. *IEEE Transactions on Computers*, 54(9):1071–1080, September 2005.

[FDBS05b] G. Feng, R. Deng, F. Bao, and J. Shen. New efficient MDS array codes for RAID Part II: Rabin-like codes for tolerating multiple (≥ 4) disk failures. *IEEE Transactions on Computers*, 54(12):1473–1483, Decemeber 2005.

[HDRT05] J. L. Hafner, V. Deenadhayalan, K. K. Rao, and A. Tomlin. Matrix methods for lost data reconstruction in erasure codes. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies*, pages 183–196, San Francisco, December 2005.

[HX05] C. Huang and L. Xu. STAR: An efficient coding scheme for correcting triple storage node failures. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies*, pages 197–210, San Francisco, December 2005.

[PD05] J. S. Plank and Y. Ding. Note: Correction to the 1997 tutorial on Reed-Solomon coding. *Software – Practice & Experience*, 35(2):189–194, February 2005.

[Pla97] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[Pla07] J. S. Plank. Fast Galois Field arithmetic library in C/C++. Technical Report CS-07-593, University of Tennessee, April 2007.

[Pla08a] J. S. Plank. A new minimum density RAID-6 code with a word size of eight. In *NCA-08: 7th IEEE International Symposium on Network Computing Applications*, Cambridge, MA, July 2008.

[Pla08b] J. S. Plank. The RAID-6 Liberation codes. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies*, pages 97–110, San Jose, February 2008.

[Pre89] F. P. Preparata. Holographic dispersal and recovery of information. *IEEE Transactions on Information Theory*, 35(5):1123–1124, September 1989.

[PX06] J. S. Plank and L. Xu. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *NCA-06: 5th IEEE International Symposium on Network Computing Applications*, Cambridge, MA, July 2006.

[Rab89] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the Association for Computing Machinery*, 36(2):335–348, April 1989.

[RS60] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8:300–304, 1960.

[SP08] C. D. Schuman and J. S. Plank. A performance comparison of open-source erasure coding libraries for storage applications. Technical Report UT-CS-08-625, University of Tennessee, August 2008.