Generalized Reed Solomon Codes for Erasure Correction in SDDS

THOMAS J.E. SCHWARZ, S.J.
Santa Clara University, USA

Abstract

Scalable Distributed Data Structures (SDDS) need scalable availability. This can be provided through replication, which is storage intensive, or through the use of Erasure Correcting Codes (ECC) to provide redundancy, which is more complicated. We calculate availability under both strategies and show that redundancy through use of an ECC implies significantly less overhead. We introduce a generalized Reed Solomon code as an ECC that uses ordinary parity (XOR) for the first level of redundancy, and adapts to the scaling up and down of an SDDS file. We derive the relevant properties of the ECC directly and discuss its adaptation to the changing needs of a SDDS.

Keywords

SDDS, erasure correcting code, Reed Solomon codes, SDDS failure tolerance

1 Motivation

Multicomputers (collections of computers connected by a high-network) combine affordability and high performance, but also require new data structures and algorithms. Scalable distributed data structures (SDDS) harness the power of multicomputers for data storage. SDDS store data in buckets at distributed sites (in RAM, local disks, network attached storage devices etc.). A SDDS file grows by generating new buckets at different sites. They allow key-based and possibly range based look-ups and through function shipping parallel scans and processing. Operation costs remain constant as a SDDS file grows and spreads over more sites. The SDDS LH* uses distributed RAM memory and offers look-up times using current network technology of less than 1 msec [B90] while allowing file sizes that could centrally only be stored on a disk.

Unfortunately, assembling the resources of a multicomputer brings vulnerability against individual node failures. An SDDS typically cannot tolerate unavailability of even small portions of its data and the overall availability of the SDDS (the probability that all data is accessible) is at best the product of the availabilities

of the individual buckets. In order to maintain the same level of overall availability, the availability of the individual buckets needs to increase as the file scales up. We call this SDDS property *scalable availability*.

We can use replication (mirroring, triplicates, higher number of copies) to safeguard the data in a bucket against failures, but the storage overhead quickly becomes prohibitive. We develop here another possibility, the use of erasure correcting codes. We group *n* buckets in a *(reliability) group* and add *m (generalized) parity buckets* to the group. We use an *erasure correcting code* (ECC) to generate the contents of the parity buckets. We can reconstruct the data of any failed data bucket by accessing a quorum $\geq n$ of data or parity buckets. The availability of the data in a single bucket using this technique is significantly higher than using replication for the same storage overhead. This is particularly the case if the quorum is exactly *n* and the size of the parity buckets is the same as the size of the client data buckets. In this case, the storage overhead $\frac{n+m}{m}$ is exactly the minimum redundancy needed to provide m-availability. Other properties of an ideal erasure correcting code are low parity calculation and low reconstruction complexity.

The simplest ECC code is the *parity code* which stores the parity (XOR) of all data buckets in one parity bucket. We reconstruct any failed data bucket as the parity of the remaining buckets in the group. This code only provides 1-availability (protection against a single failure), but has the fastest possible encoding and decoding times. We use a generalization of this code, a generalized Reed Solomon (GRS) code. This code has optimal storage overhead for a given availability level, decent encoding and reconstruction complexity and offers a large number of possible parity buckets. In addition, changes as little as a byte change are reflected in the update of a single parity byte, which is important if we store records and small objects.

2 System Availability

An SDDS file fails when it experiences data loss. We distinguish between central component failures (e.g. network) and failures of individual nodes. The former are catastrophic since they allow no recovery. We only calculate the Mean Time to Failure resulting from the latter failures. The true MTTF is roughly 1/MTTF1 + 1/MTTF2, where MTTF1 is the time to a catastrophic failure and MTTF2 is the result of calculations in our simplified model.

The SDDS file is stored in a number of reliability groups with *n* data buckets and *m* parity buckets stored at different nodes. If $n = 1$ the file uses replication with *m* mirrors. Each node fails at a given rate λ. If a node failure is detected and the data can be replaced on a hot spare, we have repaired the failure. The repair time (including detection) is assumed to be exponentially distributed at a rate ρ. This assumption is unrealistic, but does not distort the true MTTF value too much.

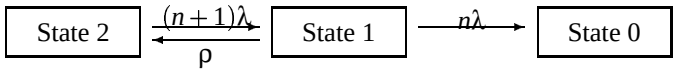

Figure 1: Markov model of a reliability group with n data and one parity bucket.

2.1 MTTF Calculations

Because we assume constant failure and repair rates, we model the survival of an SDDS with a Markov model. A Markov model consists of various states reflecting the current state of the system. The system makes a transition to another state whenever something happens (e.g. a failure, reconstruction of data on a hot standby), these occur with fixed *transition probabilities.*

Figure 1 gives the Markov model for the case $m = 1$. In the initial state 2, all buckets function. With probability λ per time unit, any one of the $n+1$ buckets becomes unavailable. The system then enters into state 1, where all the data is available but the redundancy is zero. Since there are $n+1$ buckets that can fail, the transition rate is $(n+1)\lambda$. The failure is detected and the data is reconstructed with probability ρ, so that the system returns to state 2. However, at a rate of $n\lambda$ one of the surviving n buckets fails, so that the system transitions into the failure state 0. The failure state is absorbing, there are no transitions out of it.

We denote the probability that the system is in state i at time t with $p_i(t)$. We collect these probabilities in the probability vector $\wp(t) = (p_n(t), \ldots, p_1(t))$. The probability $p_0(t)$ to be in the absorbing state,is $p_0(t) = 1 - \sum_{\nu=1}^{\nu=n} p_\nu(t)$. $\wp$ is determined by the differential equation $d\wp(t)/dt = \wp(t)\mathbf{A}$ with a *transition matrix* $\mathbf{A}$. In our example, we have

$$dp_2(t)/dt, dp_1(t)/dt = (p_2(t), p_1(t)) \cdot \begin{pmatrix} -(n+1)\lambda & (n+1)\lambda \\ \rho & \rho - n\lambda \end{pmatrix}.$$

The survival probability $r(t)$ is the probability that the system is not in the absorbing State 0: $r(t) = p_1(t) + \ldots p_n(t) = \wp(t)\vec{e}$, where $\vec{e} = (1,1,\ldots,1)^t$. The failure probability density is $f(t) = d(1-r(t))/dt$. Then

$$\text{MTTF} = \int_0^\infty t f(t)dt = -\int_0^\infty t dr/dt\, dt = \int_0^\infty r(t)dt.$$

The Laplace transform of r is $\Lambda(r)(s) = \int_0^\infty r(t)e^{-st}dt$. Obviously, $\text{MTTF} = \Lambda(r)(0)$. We take the Laplace transform of $d\wp(t)/dt = \wp(t)\mathbf{A}$ and obtain

$$s\Lambda(\wp)(s) - \wp(0) = \Lambda(d\wp/dt) = \Lambda(\wp)(s) \cdot \mathbf{A}$$

Setting $s = 0$ we have $-\wp(0) = \Lambda(\wp)(0) \cdot \mathbf{A}$ and hence

$$\text{MTTF} = \int_0^\infty r(t)dt = \int_0^\infty \wp(t)dt\vec{e} = \Lambda(\wp)(0)\vec{e} = -\wp(0) \cdot A^{-1} \cdot \vec{e}.$$

m	MTTF
0	$1/\lambda$
1	$\frac{1}{(n+1)\lambda} + \frac{\rho}{(n+1)n\lambda^2}$
2	$(\frac{1}{n+2} + \frac{1}{n+1} + \frac{1}{n})\frac{1}{\lambda} + (\frac{1}{(n+2)(n+1)} + \frac{2}{(n+1)n})\frac{\rho}{\lambda^2} + \frac{2}{(n+2)(n+1)n}\frac{\rho^2}{\lambda^3}$
3	$(\frac{1}{n+3} + \frac{1}{n+2} + \frac{1}{n+1} + \frac{1}{n}))\frac{1}{\lambda} + (\frac{1}{(n+3)(n+2)} + \frac{2}{(n+2)(n+1)} + \frac{3}{(n+1)n})\frac{\rho}{\lambda^2}$ $+(\frac{2}{(n+3)(n+2)(n+1)} + \frac{6}{(n+2)(n+1)n})\frac{\rho^2}{\lambda^3} + \frac{6}{(n+3)(n+2)(n+1)n}\frac{\rho^3}{\lambda^4}$

Table 1: Formulae for MTTF: λ bucket failure rate, ρ repair rate, n group size, m number of parity buckets.

m	$n=1$	$n=4$	$n=8$	$n=16$	$n=1$	$n=4$	$n=8$	$n=16$
0	0.001	0.001	0.001	0.001	5e-005	5e-005	5e-005	5e-005
1	26.720	10.689	5.9394	3.1453	0.010	0.004	0.002	0.001
2	951,91	190,400	63,474	18,673	3.071	0.621	0.210	0.063
3	3.82e10	4.36e09	9.25e08	1.58e08	986.54	113.80	24.445	4.268
4	1.63e15	1.17e14	1.65e13	1.68e12	338121	24364	3486	364.6
5	7.26e19	3.46e18	3.39e17	2.14e16	1.21e08	5.80e06	573977	37127
6	3.33e24	1.11e23	7.76e21	3.12e20	4.43e10	1.49e09	1.05e08	4.33e06
7	1.56e29	3.77e27	1.93e26	5.08e24	1.66e13	4.06e11	2.11e10	5.65e08
8	7.39e33	1.34e32	5.17e30	9.05e28	6.33e15	1.16e14	4.51e12	8.06e10
9	3.55e38	4.97e36	1.46e35	1.74e33	2.44e18	3.44e16	1.02e15	1.24e13

Table 2: Sample Mean Time to Failure in years ($\lambda = 1$ per year, $\rho = 6.25$ per hour (left) and $\lambda = 20$ per year, $\rho = 1$ per hour) depending on group size n and parity buckets m per group. The numbers represent the MTTF for 1000 buckets.

This calculation reduces the MTTF calculation ot matrix inversion. Typically, $\wp(0) = (1,0,\ldots 0)$ so that the MTTF is the negative of the sum of the top row of the inverse of the transition matrix. In our example, MTTF $= \rho/(n+1)n\lambda^2 + 3/2\lambda$. We apply our calculations to calculate the MTTF of a reliability group with n data buckets and r parity buckets. Repair of a failed bucket is possible as long as there are a total of n (data or parity) buckets available. Replication is the specialization $n = 1$ and $r > 1$. We give the formulae for the first four values of m in Table 1. The typical failure rate of a bucket is at worst several times a year but the typical repair time is in minutes. In consequence, the ratio ρ/λ is large, and the addend with the highest power of ρ/λ dominates the expression. Since the MTTF of L replicated buckets is $1/L$ of the MTTF of an individual buckets, we can compare the MTTF of a reliability group with n data buckets to that of a replicated bucket by dividing the former by n. We give sample values (with a failure rate of 1 per year (20 per year) and repairs every 9.6 minutes (every hour)) in Table 2.

p	$n=1$	$n=2$	$n=4$	$n=6$	$n=8$	$n=10$
0.2	0.000	0.000	0.073	0.522	0.831	0.945
0.1	0.000	0.157	0.898	0.992	0.999	1.000
0.01	0.905	0.998	0.999	1.000	1.000	1.000

Table 3: Survival probability of a 1000 bucket SDDS file.

2.2 Related Failures

Mean Time to Failure calculations assume independent failures (and repairs). We model common causes by assuming that all buckets fail with a given probability p and calculate the survival of the system, that is, the probability that in any group of $n+m$ buckets n survive. If the individual survival rate $1-p$ is smaller than $(n+m)/n$, the survival of a group has probability less than $1/2$. For larger survival rates, the odds for group survival increase with larger n at fixed ratio of $n+m/n$. In Table 3, we calculate the file survival rate for different group sizes n and for different individual site failure rates p. We again observe that larger group sizes result in better file survival. This is a consequence of the law of large numbers.

2.3 Conclusions

The MTTF for groups with several parity buckets quickly becomes fantastic because we do not take failure of essential components into account. Despite this caveat, our results show that adding a parity data to an existing reliability group increases MTTF considerably. A given number of parity storage overhead is thus best divided among few groups. Larger groups protect even against common failures. A designer has to balance these gains against the increasing complexity of parity generation and data reconstruction of larger groups. The ratio $(\rho\ \lambda)^m$ contributes most to the MTTF and shows the importance of fast repair.

3 Implementation of Galois Fields

Galois fields are finite sets together with an addition and a multiplication that obey the arithmetical rules (associativity, distributivity, etc.) of the real numbers, the complex numbers, and the rational numbers. They have a zero element 0 and a one element 1. Two Galois fields with the same number of elements are essentially ("up to isomorphy") the same and we call the Galois field with l elements $GF(2^l)$. Because computers use bits, we use a Galois field $F = GF(2^l$. The elements of F are all possible bit strings of length f. The addition in F is the bitwise XOR (exclusive-or) of two strings. The zero element 0 is the bitstring $00\ldots0$. Since $x \oplus x = 0$, each element is its own negative and subtraction is the same as addition.

The mathematically preferred way to define multiplication in F is to interpret all bit strings as polynomials of degree f with coefficients 0 or 1. For example, if $f = 8$, the byte 00100011 corresponds to the polynomial $t^5 + t + 1$. We choose a *generator polynomial,* which is a binary polynomial of degree $f + 1$ that cannot be expressed as a non-trivial product of two bionary polynomials. To multiply two elements of F, we multiply the corresponding polynomials, then divide with remainder by the generator polynomial. The product is the string corresponding to the remainder. The whole procedure can be implemented using $f - 1$ left shifts, $f - 1$ bit testing, and up to $2f - 2$ XORs.

Instead of this expensive procedure, we use look-up tables. The straightforward approach is a two-dimensional multiplication table. The storage cost is $2^f \cdot 2^f$ entries for a total of $f \cdot 2^{2f}$. For $f = 8$, this amounts to 64KB. A lookup costs an integer multiplication, two additions, and a load.

To limit the size of the table, we can use logarithms. First, we pick a *primitive element* α, that is, all nonzero elements β can be written as $\beta = \alpha^i$. i is a uniquely determined number between 0 and $2^f - 1$. We say $i = \log_\alpha(\beta)$ and call i the logarithm of β (with respect to α). We call $\beta = \text{antilog}_\alpha(i)$ the antilogarithm of i. For convenience's sake we put $-\infty = \log_\alpha(0)$ and $0 = \text{antilog}_\alpha(-\infty)$. Since $\log_\alpha(\gamma \cdot \delta) = \log_\alpha(\gamma) + \log_\alpha(\delta) \bmod 2^f - 1$, we can implement multiplication according to

$$\gamma \cdot \delta = \text{antilog}_\alpha(\log_\alpha(\gamma) + \log_\alpha(\delta)) \text{ if } \gamma, \delta \neq 0 \text{ and } 0 \text{ otherwise}$$

by using a table of logarithms and one of antilogarithms. We implement this with three table look-ups, two tests for equality of zero, and an addition modulo $2^f - 1$. The latter operation itself costs an integer addition, a check for overflow and possibly a subtraction. We can avoid this awkward operation by storing the antilogarithm of all possible sums of logarithms. The antilogarithm table then has $2 \cdot (2^f - 1)$ entries and the logarithm table $2^f - 1$ entries for a total of $3 \cdot 2^f$. For example, in the case of byte-sized elements $(f = 8)$ the table storage is 0.75KB. The costs of multiplication are two checks for equality to zero, three lookups into a one-dimensional table, and one integer addition. Thus, the system uses two checks for equality, four additions, and three loads for a total of nine elementary operations.

We can implement division in an analogous way. However, since we need division only rarely, we can implement the division operation by an inversion followed by a multiplication. Since $\gamma^{-1} = \text{antilog}_\alpha(2^f - 1 - \log_\alpha(\gamma))$, inversion takes two table-lookups and a subtraction for a total of five elementary operations.

Our implementation has constant time complexity and exponential storage complexity (pace the otherwise superb [BFT98].) Even though we do not believe that in practice one has to go beyond byte-sized Galois field elements, we give by example a method to deal with larger Galois field elements without a correspondingly large table. Assume that f is divisible and let g be a divisor so that $gh = f$. Each bit string of length f consists h strings of length g. We write a string of length f as the concatenation $[s_{h-1}, s_{h-2}, \ldots, s_0]$ of h strings s_i of length g. We

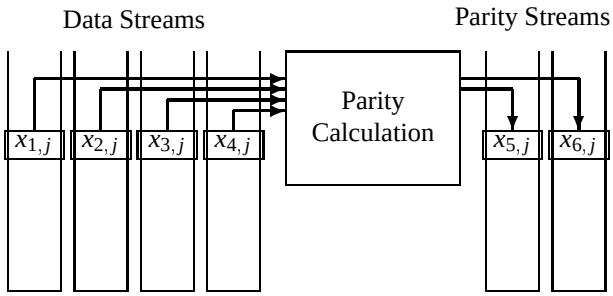

Figure 2: Generation of Parity Streams from Data Streams

implement Galois field arithmetic on the strings of length g. We call the resulting field G to distinguish it from the field with elements of length f, which we call F. Let $p(x) = x^h + p_{h-1}x^{h-1} + p_{h-2}x^{h-2} + \ldots + p_1x + p_0$ be an irreducible polynomial with coefficients in G. Algebra assures us that such polynomials always exist. We now introduce Galois field operations on the bit strings of length f. Addition and subtraction are both the Exclusive Or (XOR) operation. We multiply two strings $[0,0,\ldots,0,s_0]$ and $[0,0,\ldots,0,t_0]$ as $[0,0,\ldots,0,s_0t_0]$, where we use the multiplication in G. We identify the bit string $[s_{h-1},s_{h-2},\ldots,s_0]$ with the polynomial $s_{h-1}x^{h-1} + s_{h-2}x^{h-2} + \ldots + s_1x + s_0$. We now multiply two elements of F as polynomials, and reduce the result by p. This can be achieved with $2h^2 + h - 1$ multiplications and $2h - 2$ additions.

4 Definition of GRS Codes and Application to SDDS

Generalized Reed Solomon codes belong to a wide class of linear, algebraic block codes. We use them to provide generalized parity streams to n streams of data. The data streams are bit streams, but we group f bits together as a symbol, so that the streams are now streams of symbol. We define a Galois field F over these symbols and use it to generate the parity streams from the data streams as illustrated in Figure 4. To simplify notation we assume that we have data streams $i = 1,\ldots,n$ and parity streams $i = n+1,\ldots n+m$. We write stream i as $(x_{i,0},x_{i,1},x_{i,2},\ldots)$. We form a *data word* $\vec{x}_r = (x_{1,r},x_{2,r},\ldots,x_{n,r})$ from the symbols in parallel position in all the data streams. For each data word, we calculate a *code word* $\vec{y}_r = (x_{1,r},x_{2,r},\ldots,x_{n,r},x_{n+1,r}\ldots x_{n+m,r})$. The last m coefficients in the code word are the parity symbols.

4.1 Generator Matrix Definition and Existence

For the code word calculation we use a *generator matrix* **G** with coefficients in F defined by the following properties: (1) **G** has n rows and m columns. (2) The first n columns of **G** form the identity matrix. (3) All coefficients in the $n+1^{\text{st}}$ column are one. (4) Any $n \times n$ matrix formed of any n different columns of **G** is invertible. We make the following important observation:

Theorem 1 *There exists a generator matrix with properties (1) to (4) for* $m = 2^f + 1$.

Proof. Let **V** be the n by 2^f Vandermonde matrix formed by all 2^f elements $\{r_j\}$ of F, i.e. $\mathbf{V} = (v_{i,j})_{1 \le i \le n, 1 \le j \le 2^f} = (r_j^{i-1})$. Row i is formed by the $i-1^{\text{st}}$ power of the elements of F. We append one additional column $(0,0,\ldots 0,1)^{\text{t}}$ to **V**. It is well known that any n by n submatrix of **V** has a non-zero determinant and is therefore invertible ([MM69], 2.4.10). In particular, the first n columns v_j are linearly independent. We write $v_{n+1} = \sum_{j=1}^{n} \alpha_j v_j$. $\alpha_j \neq 0$ as a quotient of two Vandermonde determinants. We change the first columns v_j, $1 \le j \le n$ by multiplying with α_j. The resulting matrix still fulfills property (4). Let **A** be matrix formed by the newly changed first n columns. We set $\mathbf{G} = \mathbf{A}^{-1} \cdot \mathbf{V}$. Clearly, **G** satisfies (1), (2), and (4). Since $\mathbf{A}(1,1,\ldots 1)^{\text{t}} = v_{n+1}$, the $n+1^{\text{st}}$ column of **G** equals $(1,1,\ldots 1)^{\text{t}}$. □

4.2 Parity Calculation

Given a data word $\vec{x}_r$, we form $\vec{y}_r = \vec{x}_r \cdot \mathbf{G}$. Because of property (2), $\vec{y}_r$ contains the coefficients of $\vec{x}_r$ in the first n positions. To calculate the j^{th} coefficient of $\vec{y}_r$ we need to multiply $\vec{x}_r$ with the j^{th} column of **G**. The creation of a parity symbol costs n Galois field multiplication and $n-1$ XOR operations. Since a site calculates a whole parity stream, the logarithms of the coefficients of **G** are used in every operation and can be cached. A site storing parity only stores the logarithms of the corresponding column of **G**. The calculation of the first parity stream is easier because of (3). The symbols in this stream are merely the XOR of all the data symbols.

Assume that we update a single data stream i. Let $\vec{x}_p^{\text{old}}$ be the old p^{th} data word and $\vec{x}_p^{\text{new}}$ the new one. $\vec{x}_p^{\text{old}}$ and $\vec{x}_p^{\text{new}}$ only differ in the i^{th} coefficient by a value δ_p. Let x_j, $n < j \le n+m$ be a parity symbol. Write $\mathbf{G}_j$ for column j of **G** and $\mathbf{G}_{(i,j)}$ for the coefficient in row i and column j of **G**. Then $x_j^{\text{new}} = \vec{x}_p^{\text{new}} \cdot \mathbf{G}_j$ $= (\vec{x}_p^{\text{old}} + (0,\ldots,0,\delta_p,0,\ldots,0) \cdot \mathbf{G}_j = x_j^{\text{old}} + \delta_p \cdot \mathbf{G}_{(i,j)}$. Thus, the old and new parity symbol differ by the change in the data symbol δ_p multiplied with the coefficient of **G** located in the row indexed by the changed data stream and in the column corresponding to the parity stream.

We organize the update of a single data stream by calculating the *delta stream* $(\delta_0, \delta_1, \ldots)$ consisting of the differences between new and old value at the data bucket and then send the delta stream to all the parity buckets together with the number i of the parity bucket. At the parity buckets, the delta stream is multiplied with the generator matrix coefficient found in the row corresponding to the updated data stream and the column corresponding to the parity stream. This is similar to the "small write" in disk arrays (RAIDs).

The column number of **G** limits the total number $n + m$ of data and parity streams. For LH*RS [LS00] using the Galois field $GF(256)$ elements has the best performance (a total table size of 768B) while allowing up to 257 streams.

4.3 Data Reconstruction

If one or more streams $j_1, \ldots j_l$ are lost, we need to reconstruct these streams. If the lost streams are all parity streams, we can of course recalculate them using our generator matrix **G**. If one or more of the lost streams is a data stream, we need n data or parity streams x_{i_ν}. If these cannot be found, then the file has suffered irretrievable data loss. Otherwise, we form a submatrix **H** consisting of columns i_ν of **G**. Matrix **H** is invertible. We have $\vec{x}_r \cdot \mathbf{H} = (x_{i_1,r}, x_{i_2,r}, \ldots x_{i_n,r})$. Consequentially, $\vec{x}_r = (x_{i_1,r}, x_{i_2,r}, \ldots x_{i_n,r}) \cdot \mathbf{H}^{-1}$. This reconstructs all data streams.

We can calculate missing parity streams from the generator matrix or use the following better method. Let **A** be the matrix consisting of the columns $j_1 \ldots$ of **G** corresponding only to the parity streams. **A** is not necessarily a square matrix. Then $(x_{j_1,r}, \ldots, x_{j_l,r}) = (x_{i_1,r}, x_{i_2,r}, \ldots x_{i_n,r}) \cdot \mathbf{H}^{-1} \cdot \mathbf{A}$. The costs of inverting **H** only occurs once in the initial phase of reconstruction. For reasonably small number of streams, this cost amortizes to something negligible over the complete reconstruction process.

The reconstruction process starts with collecting data on the availability of streams and the collection of the columns of **G** stored at participating parity sites. We form the matrix **H** and invert it at the site that reconstructs. Since we usually reconstruct more than a single record, we cache the inverse matrix. The cost of inversion (negligible for small size **H**) are thus distributed over many reconstructed records. We use $\mathbf{H}^{-1}$ to calculate the reconstructed data symbols and code symbols from the participating streams. This calculation can be distributed over the participating sites by letting the sites that contribute a stream perform multiplications with elements of $\mathbf{H}^{-1}$ directly before sending them on to the spare sites, which then merely calculate the parity (XOR) of the arriving pre-processed streams.

4.4 Variable Group Size

Typically, the number of data streams n is an invariant of the SDDS. The choice of n is a compromise based on availability needs, storage overhead, reconstruction

difficulty, the availability of parity buckets, and parity bucket loads. Some flexibility might prove advantageous. We provide this flexibility within the bounds of $n+m \leq C\mathit{ardinality}(F)+1$ fixed by the number of columns in the generator matrix **G.** We pick a maximum number $n_{\max}$ of data streams. We form a generator matrix **G** according to Theorem 1 with $n_{\max}$ rows. For example, if we choose $F = \mathrm{GF}(2^8)$, then 128 would be a good choice for $n_{\max}$ that still allows 129 parity streams.

We start by creating $n < n_{\max}$ data streams. The remaining $n_{\max} - n$ data streams are zero. We create an adequate number of parity streams using the right or parity columns of **G.** Since the data streams $n+1$ to $n_{\max}$ are zero streams, they do not enter into the calculation of the parities. In consequence, the encoding complexity of the scheme has not changed. Similarly, the reconstruction needs only n data or parity streams since we always have $n_{\max} - n$ dummy data streams. These dummy streams do not enter into the actual reconstruction calculation. The reconstruction only inverts the matrix **H** consisting of the n columns corresponding to the n existing streams and the rows corresponding to the n actual data streams.

5 Extension of GRS Codes

We choose a Galois field $F_f = GF(2^f)$ to accomodate a maximum of $2^f + 1$ data and parity streams. While not of interest for current SDDS, it is at least theoretically interesting to observe that this bound can be dynamically expended, though at the cost of higher parity calculation. We only sketch the procedure.

We embed $F_f = GF(2^f)$ in a Galois field $F_{2f} = GF(2^{2f})$ as in Section 3. Calculations of elements in F_f are independent of the field in which they are performed. We form the generator matrix **G** in F_{2f} in the manner outlined in Section 4.1, but so that the first columns of the Vandermonde matrix **V** are the powers of the elements in the F_f. The next column of **V** is $(0,0,\ldots 0,1)^t$. The remaining columns of **V** contain the powers of the remaining elements in F_{2f}. The Vandermonde matrix over F_f consists of the first $2^f + 1$ columns of **V.** We form the generator matrix **G** in the manner described in the proof of Theorem 4.1. The resulting generator matrix **G** over F_{2f} is a n by $2^{2f} + 1$ matrix that contains the $2^f + 1$ columns of the generator matrix of over F_f as the first left columns.

A symbol s of the code over F_{2f} is a bit string of length $2f$ and thus the concatenation $[s_0, s_1]$ of two symbols of the code over F_{2f}. Recall that we identify element s_0 in F_f with element $[0, s_0]$ in F_{2f}. The definition of addition and multiplication in F_{2f} is such that $[0, s_0] \cdot [t_0, t_1] = [s_0 t_0, s_0 t_1]$ and such that $[s_0, s_1] \oplus [t_0, t_1]$ $= [s_0 \oplus t_0, s_1 \oplus t_1]$. Therefore, the parity streams among the first $2^f + 1$ streams in the code over F_{2f} are exactly the same as the parity streams in the code over F_f, though the calculation in the first code generates $2f$ bits at a time in contrast to f bits in the code over F_f. The log and antilog tables for multiplication in F_{2f} are $2 * 2^f$ larger than the tables for F_f. If f is reasonably large, we need to use

Operation	Time
Even-Odd Update	2.20 μsec
GRS Update	4.89 μsec
XOR Update / Reconstruction	2.20 μsec
RS Reconstruction (2 streams)	31.47 μsec
EvenOdd Reconstruction (2 streams)	6.98 μsec

Table 4: Coding and Encoding Times, AMD 1700+ machine under Win98, 1/2KB records, reconstruction without setup.

the mixed multiplication strategy of Section 3. Then the calculation in F_{2^f} uses 5 multiplications in F_f.

If we start with bit strings of length 8, we can implement a total of 257 streams. Doubling once yields 65,537 streams maximum and one more doubling yields more than 4 billion streams. Chances are that we run out of parity servers before.

6 Alternative Codes

Our GRS code is not the only possibility to define storage efficient erasure correcting codes. Array codes [BFT98, X98, XB99] are formed by placing symbols (as small as bits and as large as disk blocks) from data streams into some positions of a matrix. Using XOR operations only, the remaining coefficients of the matrix are calculated and form the parity streams. Evenodd, B- and X-codes generate only two parity streams but have very fast encoding and reconstruction times. Unfortunately, array codes with more parity streams have comparable reconstruction times to GRS codes and encode larger portions of data streams at a time as the number of parity streams increases. We compare Evenodd and our GRS code in Table 3. GRS is noticeable slower, but still fast enough for encoding and decoding to become a bottleneck in a 1GB/sec network. If we use only the first parity bucket to reconstruct a single data stream, then reconstruction for both codes defaults to calculating the ordinary XOR parity.

The digital fountain project [BL98] optimizes downloads of large files from several servers. Each file is broken into chunks. Each server sends out GRS encoded blocks calculated for the chunks. At the receiving end, once a sufficient number of blocks has been received, we decode the blocks and regenerate the chunk. In this way, the digital fountain avoids the need to ask for retransmission of missed packages. The GRS used for the digital fountain [BK95] uses Cauchy matrices $\left(\frac{1}{a_i+a_j}\right)_{i,j}$ instead of Vandermonde matrices because the matrices $\mathbf{H}$ can be easier inverted. Doing so limits the maximum number of parity streams.

A very different type of code, Tornado code, can be used for the digital foun-

tain as well [BL98]. Tornado codes have very low encoding and reconstruction complexity, but they need large code words and they have a larger storage overhead. With other words, if we use Tornado codes, then our parity buckets need to be larger than in an Array or a GRS code and a single data symbol would be so long that it contains complete small records. Updating records then incurrs additional overhead. Future work on ECC should exploit the potential of both Tornado and array codes.

References

[ABC97] Guillermo Alvarez, Walter Burkhard, and Flaviu Cristian: Tolerating Multiple Failures in RAID architectures with Optimal Storage and Uniform Declustering. ISCA 1997, Denver, pp. 62-72.

[BL00] Fethi Bennour Sahli et al.: Scalable Distributed Linear Hashing LH*LH Under Windows NT, IEEE SCI-2000 Orlando, Florida, USA. July 23-26, 2000.

[BFT98] M. Blaum et al.: ”Array Codes” in: V.S. Pless and W.C. Huffman (ed): Handbook of Coding Theory, North-Holland; ISBN: 0444814728.

[BK95] J. Blömer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, D. Zuckerman: ”An XOR-Based Erasure Resilient Coding Scheme” ICSI TR-95-048, August 1995. (Available at www.icsi.berkeley.edu/~luby)

[BL98] John Byers, Michael Luby, Michael Mitzenmacher, Ashu Rege: ”A Digital Fountain Approach to Reliable Distribution of Bulk Data”, ACM SIGCOMM 1998.

[DL01] A. W. Dine and W. Litwin: ”Implementation and Performance Measurements of the RP* Scalable Distributed Data Structure for Windows Multicomputers.” Intl. Workshop on Performance-Oriented Program Development for Distributed Architectures (PADDA), Munich, 2001.

[MM69] Marvin Marcus and Henryk Minc: A Survey of Matrix Theory and Matrix Inequalities. Prindle, Weber, & Schmidt, Boston 1969, Dover, Mineola, 1992.

[MS97] F. J. MacWilliams and N. J. A. Sloane: The Theory of Error Correcting Codes, North Holland, Amsterdam 1997.

[LS00] W. Litwin, Th. Schwarz. LH*RS: A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes. ACM-SIGMOD-2000 Intl. Conf. On Management of Data. Dallas 2000.

[LNS96] W. Litwin, M.-A. Neimat, D. Schneider: ”LH* - A Scalable Distributed Data Structure”, ACM Transactions on Database Systems, Dec. 1996 Vol. 21 Issue 4

[XB99] L. Xu and J. Bruck ”X-Code: MDS Array Codes with Optimal Encoding,” IEEE Trans. on Information Theory , Vol. 45, No. 1, pp. 272–276, January 1999.

[X98] L. Xu, Highly Available Distributed Storage Systems,” Ph.D. thesis, California Institute of Technology, 1998. Also available at: http://paradise.caltech.edu/ lihao/thesis.html.

Thomas J.E. Schwarz, S.J. is with the Department of Computer Engineering Santa Clara University, Santa Clara, CA 95053. E-mail: tschwarz@calprov.org