A Reed-Solomon Code for Disk Storage, and Efficient Recovery Computations for Erasure-Coded Disk Storage

M. S. MANASSE
Microsoft Research, Mountain View, California, USA

C. A. THEKKATH
Microsoft Research, Mountain View, California, USA

A. SILVERBERG
Mathematics Department, University of California at Irvine, USA

Abstract

Reed-Solomon erasure codes provide efficient simple techniques for redundantly encoding information so that the failure of a few disks in a disk array doesn't compromise the availability of data. This paper presents a technique for constructing a code that can correct up to three errors with a simple, regular encoding, which admits very efficient matrix inversions. It also presents new techniques for efficiently computing the sums of streams of data elements in a finite field, such as is needed in recovering data after a disk failure.

Keywords

Erasure codes, Reed-Solomon codes, Streaming computation, Parallel algorithms.

1 Introduction

The efficient computation of encodings and decodings of erasure codes are important when considering very large storage arrays (arrays of thousands of disks). At that scale, hosting space and power are significant expenses, so erasure codes may be employed to minimize the overall cost of the system. To provide high-probability of data survival, erasure codes handling multiple concurrent failures should be employed; assuming independent failures, correctable within a day, a code which allows up to three failures per group of 250 disks should experience data loss once every 50,000 years in a set of 10,000 disks.

In the first section of this paper, we present a simplified construction of a triple-erasure correcting Reed-Solomon code, which admits one more storage device than previously known for a given Galois field of characteristic two. We further explain why this technique is effective only over Galois fields of characteristic two, and why the technique cannot extend beyond three erasures.

In the second section, we consider the problem of efficiently computing streams of parallel sums. In a practical erasure code setting, this is the core computation for reconstructing a failed disk, using exclusive-or (XOR) as the addition operator, but the technique presented in this paper apply more generally to computing streams of sums in arbitrary monoids (like groups, only without inverses).

2 Erasure Coding

When digital data is transmitted or stored, it is common to be concerned about both errors and erasures. An error occurs in a data stream when some data element is corrupted; an erasure occurs when a data element is missing or known to be faulty. Shannon's basic theorems on coding tell us that to correct e errors, a data stream needs to guarantee a minimum distance between correct streams of at least $2e+1$; to detect e errors, or to correct e erasures, valid entries in the data streams need to be separated by at least $e+1$ bits. Simple parity is an example of a technique for handling one erasure, or detecting one error; all strings of bits which differ by one bit without parity will differ by two bits, once parity is included.

In more practical settings, we are concerned with the transmission or storage of data grouped into bytes (or groups of bytes) where we expect erasures or errors to take place in multiples of these groups, say, blocks or entire disks. To this end, Reed and Solomon devised a method for constructing linear codes over finite fields, where the individual data items can be represented as elements of the finite field. For digital data, it is most convenient to consider the Galois fields $\mathrm{GF}(2^n)$, for $n = 8$, 16, or 32, where the underlying data words are, correspondingly, single octets, pairs of octets, or quads of octets. If we want to correct e erasures from m data words (where $m+e \leq 1+2^n$), we can compute $m+e$ result words as the result of multiplying the length-m vector of the data words by an m by $m+e$ coding matrix. The key observation is that an m by $m+e$ Vandermonde matrix (of which we'll say more below) has the property that any m by m submatrix is invertible, which allows all data elements to be reconstructed from any m of the $m+e$ result words. The further observation is that this code can be made systematic by simple row reduction of the coding matrix to diagonalize the initial m by m portion of the matrix. These row reductions preserve linear independence of submatrices. This normalization results in a code where the representation of a data word is the data word itself, along with some contribution to the e encoding elements; that is to say that the encoding matrix is an m by m identity matrix followed by an m by e checksum computation matrix.

The largest standard Vandermonde matrix which can be constructed over $\mathrm{GF}(2^n)$ has 2^n columns; recent authors [1], [3] have observed that one extra (non-Vandermonde) column can be added while preserving the invertibility property. This matrix, when diagonalized, gives rise to the bounds described above. However, in this matrix, the e columns corresponding to the correction coding have no particularly nice properties. As such, the elements of the coding must be stored in a coding matrix, and consulted on every coding step, and inversion is irregular.

This paper expands on an earlier note which proposed a simpler construction of the coding matrix in the restricted case when $e \leq 3$. The construction is simple: append up to three columns of a specific Vandermonde matrix to a square identity matrix of size at most $2^n - 1$, leading to a coding matrix supporting $m + e \leq 2^n + 2$. Due to properties of Vandermonde matrices, described below, the encoding is regular, and inverse matrices are also regular. Inversion can be computed using the adjoint matrix; all of the submatrices which arise in the computation of the adjoint of an m by m submatrix of our matrix are shown to be Vandermonde matrices, and thus have easily computed determinants.

A Vandermonde matrix is one whose columns consist of consecutive powers of distinct elements of the ground field over which we are working. In a standard Vandermonde matrix, the first row contains the zeroth powers, the second row the first powers, etc. Invertibility follows from computation of the determinant; which is equal to the product of the differences of the elements used to generate the matrix, later entries minus earlier. If a column begins with a higher power of an element, the determinant is multiplied by that value (and is still non-zero, if the element is non-zero); we call such a matrix an extended Vandermonde matrix. The multiplicative growth of the determinant follows from multi-linearity of the determinant.

For our use, let g be a generator of the finite field. The elements from which we generate our Vandermonde matrix are g^0 (i.e., 1), g^1 (i.e., g), and g^2. More explicitly, if our coding matrix is M, for $i < m$, and $j < m$, $M_{ij} = \delta_{ij}$, and $M_{i,(m+k)} = g^{ki}$ for $k = 0$, 1, or 2. More explicitly, M can defined as follows:

$$\begin{bmatrix}
1 & 0 & 0 & \cdots & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & \cdots & 0 & 0 & 1 & g & g^2 \\
0 & 0 & 1 & \cdots & 0 & 0 & 1 & g^2 & g^4 \\
0 & 0 & 0 & \cdots & 0 & 0 & 1 & g^3 & g^6 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & 1 & 0 & 1 & g^{m-2} & g^{2(m-2)} \\
0 & 0 & 0 & \cdots & 0 & 1 & 1 & g^{m-1} & g^{2(m-1)}
\end{bmatrix}$$

When we take an arbitrary m by m submatrix of this coding matrix, evaluating the determinant by picking available columns from the identity portion of the matrix makes it clear that the determinant is equal (up to sign, which is irrelevant over a field of characteristic two) to the determinant of the 0 by 0, 1 by 1, 2 by 2, or

3 by 3 minor of the Vandermonde section which remains after deleting those rows in which the diagonal elements from the identity contribute a 1 to the expansion of the determinant; that is, those rows with indices of surviving disks, keeping only the rows for missing data disks. If the missing columns are i, j, and k, then the remaining minor is

$$\begin{bmatrix} 1 & g^i & g^{2i} \\ 1 & g^j & g^{2j} \\ 1 & g^k & g^{2k} \end{bmatrix}$$

This is plainly a transposed Vandermonde matrix; g^i, g^j, and g^k are distinct as long as i, j, and k are, so the determinant is $(g^k - g^j)(g^k - g^i)(g^j - g^i)$, and will thus be non-zero.

If only one disk containing data (and up to two check disks) have failed, then the corresponding minor is a singleton of the form g^i for some i, and therefore is non-zero, hence invertible. If two data disks fail, there are three cases, depending on which check disk (if any) has failed.

First, if the disks formed from powers of g^0 and g^1 are available, the resulting matrix is a simple Vandermonde matrix:

$$\begin{bmatrix} 1 & g^i \\ 1 & g^j \end{bmatrix}$$

If the g^0 and g^2 disks are available, the matrix is the same, except i and j are doubled. g^{2i} and g^{2j} are guaranteed to be distinct, because we're in a Galois field of characteristic 2, so g^{2i} and g^{2j} are congruent only if $2(i-j)$ is a multiple of the order of the multiplicative group, which is a power of 2 minus 1. Thus, 2 is relatively prime to the order of the multiplicative group of a Galois field of characteristic 2, so this reduces to asking if i and j are congruent modulo the order of the multiplicative group, hence equal.

Finally, if the g^1 and g^2 disks are available, the resulting matrix is an extended Vandermonde matrix:

$$\begin{bmatrix} g^i & g^{2i} \\ g^j & g^{2j} \end{bmatrix}$$

After factoring out g^i from the top row, and g^j from the bottom, we return to the first case, multiplied by an extra g^{i+j}.

Inverting the larger matrix can be done by computing forward to remove the influence of surviving data disks on the check disks, or by computing all the needed subdeterminants to compute the useful elements of the adjoint matrix. Inversion of the matrix is unlikely to be a significant component of recovery, but the simplicity of these inversions makes the process potentially more suitable for implementation in a micro-controller.

If we were to try to extend this to finite fields of other characteristics, we would need to be more careful: 2 is a factor of the order of the multiplicative

group, so we would need to restrict m to be less than half the order of the multiplicative group. The simplicity of the construction would remain, but the maximum number of data drives would be only half as many as is achieved using the standard technique of diagonalizing a full Vandermonde matrix, or by using Cauchy matrices.

If performing multiplication using logarithms [4], the coding matrix becomes especially simple: the logarithms in the encoding columns are all 0's, the consecutive integers starting from 0, and integers going up by twos starting from 0; when updating data element i by XOR-ing with data d, the update to check value j is to XOR antilog($\log(d) + j \times i$). Compared to conventional techniques, this saves us the cost of looking up the entry; the lookup is trivial, but in a small microcontroller, the storage for the table may not be.

For reconstruction, note that the first check function is parity, so correcting one erasure is easy. Correcting two or more erasures requires inverting the minor; this tells us by what to multiply the check elements; the other data elements' multipliers are then linear combinations of the corresponding check element multipliers by the elements of the coding matrix, which are easy to compute.

3 Reconstruction

In almost every form of erasure-coding, the final computation in recovering a block resolves to repeatedly computing the sum of many data elements held by different storage units. Typically, this sum can be viewed as a sum of vectors of elements of a finite field; there may need to be a multiplication step prior to computing the sum, corresponding, in the encoding above, to multiplication by the elements of the inverse matrix. Note that exclusive-or is the addition operator for $\mathrm{GF}(2^n)$. In simple RAID-5, there is no multiplication, and we can view the data as a vector of bits. In the Reed-Solomon code described above (or any other code over $\mathrm{GF}(2^n)$), we can apply this in a straight forward fashion, over elements from $\mathrm{GF}(2^n)$. For EvenOdd [2], we offset the vectors, but many sums are still required. Stating this with a bit more generality, we are trying to compute sums in a monoid. Let '+' be the associative binary operator with identity; in some extensions, we require a commutative monoid; that is, we ask that the addition operation be commutative. Due to associativity, we can define $\sum$, an operator for summing vectors, and observe that any grouping of the terms (or, in the commutative case, any ordering or shuffling of the terms) produces the same sum. Let d^i_j be vectors of data from source i; our aim is to compute $S_j = \sum_i d^i_j$, for all j. But we want more than this; sometimes we want to specify a j for which S_j should be computed promptly. This corresponds to our desire to reconstruct all the blocks of a failed disk, but to reconstruct client-requested blocks in preference to reconstructing other blocks.

3.1 Environment

We take, as our computational environment:

- A set of data sources D_j (the surviving data sources) containing data values d_j^i.
- A single data sink D_∞ (the target).
- Processing elements P_j, which can compute '+', and read values only from the corresponding D_j, (P_∞ is able to write D_∞). In the non-commutative case, we allow D_j to be supplied as either the left or right argument, as needed.
- A network that synchronously allows each processing element to receive at most one, and transmit at most one data value; the processing elements can simultaneously read one value, and perform one addition using the value just read, and either the local data value, or the most-recently computed sum, supplying the two selected values in either order to be summed, in the non-commutative case.

While this model is somewhat unrealistic, it is close to what can be achieved by connecting processors to small Ethernet switches, with high-speed uplinks to a fast switch. Disks provide data at speeds only slightly faster than 100 megabit networks; by replicating the network we can attain rough parity. Constructing a two-level network with a high-performance gigabit switch above slower ones allows us to appear to have arbitrary independent pairs communicate concurrently, as long as much of the communication never leaves a slower switch. In the examples we consider, we use 12 port gigabit switches as our fast switches, and 24 port 100 megabit switches with two gigabit uplinks as our slow switches. We will present design communication patterns in which at most 8 values in each direction transit the gigabit link concurrently.

3.2 Bucket brigade: A high-latency, optimal-throughput solution

If we care only about throughput, we can easily arrange for a bucket-brigade: the first processor reads blocks and forwards them to the second processor, which adds the received value to the corresponding data value and forwards the result to the third processor, which adds the received value to the corresponding data value and forwards the result to the fourth processor and so on. If we arrange the processors so that most consecutive processors are on the same small switch, we can limit our inbound and outbound cross-switch traffic to one slow link in each direction. Doing so (depending on how the wiring is arranged) may require

processors to be arranged in ways where consecutive doesn't correspond to consecutive in the sum, which requires us to depend on commutativity of addition; if the processors naturally come in the right order, we don't even need associativity.

This organization of the computation has high throughput, but suffers from high latency. With n summands, the first result appears after n packets have been passed down the chain. On the other hand, a finished result is computed on every packet transmission to the last node thereafter, and packet transmissions can be overlapped, so the pipeline has optimal throughput, computing results at the speed of the slowest of the input, network, and processing speeds, which we've assumed to all be identical.

The latency is of no concern when we're reconstructing complete disks; the time to read or write a complete disk is many orders of magnitude larger than the latency of sending hundreds of network packets. If a disk array is removed from service during reconstruction, this would be adequate. If, however, we want the disk array to satisfy read or write requests while operating in degraded mode, we will be unsatisfied with the latency in this scheme.

3.3 An in-place binary tree: an optimal-latency low-throughput solution

To achieve lower latency, we begin by exploiting associativity. Noting that $a+b+c+d=(a+b)+(c+d)$, we can construct an in-place binary tree that can compute a sum in $\log_2 n$ rounds of communication, which is optimal. We can construct such a network by sending from all P_{2i+1} to P_{2i} in the first round, all P_{4i+2} to P_{4i} in the second round, all P_{8i+4} to P_{8i} in the third round, and so on, up to the last round, in which $P_{2\lceil log_2 n\rceil-1}$ sends to P_0, summing the values as received (and omitting any communications where the subscript is n or larger). The difficulty with this construction is now that the throughput is reduced by a factor of $\log_2 n$, because P_0 receives a message in $\log_2 n$ consecutive steps.

We can combine these two techniques, using the bucket-brigade for batch reconstruction, and switching to the tree-based approach for demand reconstruction, and this is probably adequate for any practical situation. It remains theoretically unsatisfying, however; we're underutilizing our network in the tree approach, and we can easily do better. As a trivial improvement, observe that we are never receiving any messages at odd nodes, nor sending any messages from even nodes in the first round, nor sending from odd nodes except in the first round. We can overlay a second tree sending from $2i$ to $2i+1$, from $4i+1$ to $4i+3$, and so on in consecutive rounds if we add one extra unit of latency and some buffering to perform an extra disk read before starting the communication. The summands need to be provided in the opposite order, but this arrangement computes two totals as quickly as the previous method computed one. This improves the throughput by a factor of two, trivially, raising the question: can we do better?

We would need approximately $\log_2 n$ non-interfering trees to raise the through-

put to match the bucket brigade. Even if we add more latency and buffering, in the first step of an in-place binary tree, half of the nodes are sending and half receiving. Thus, we can't hope to improve on a factor of 2, if we start all of our trees at the same time. We must instead stagger the initiation of our trees, so that while we're using half the available communications bandwidth for the leaves of the tree, we're using a quarter for the next level on the previous block, an eighth for the level above that on the block started two cycles ago, and so on. This is the basic idea of the scheme presented in the next section.

3.4 In-place binary trees: an optimal-latency, optimal-throughput solution

A recursive form of a construction along the lines described above follows. As the base case, consider two nodes. In this case, the bucket brigade is optimal in both latency and throughput, so we'll take that as the base of our recursion, consisting of the tree sequence ($\{0 \rightarrow_0 1, 1 \rightarrow_1 \infty\}$), where the notation $\rightarrow_i$ denotes the action of sending data in step i. Thus P_0 sends block 0 to P_1 during the first step (step 0), and block 0 is written to its destination during the second step (step 1). We extend our list by incrementing the block and step numbers to form a sequence of the appropriate length, by adding the number of sets in our list to all subscripts. A processor receiving a block always adds it to the value it already knows for that block, which in this case is just the local value; a processor sending a block always sends the accumulated sum known for the block, so the value sent to the destination disk is the sum of the values held at processors 0 and 1.

Given our base case, we extend to a system for four nodes producing two overlaid tree patterns ($\{0 \rightarrow_0 1,\ 2 \rightarrow_0 3,\ 1 \rightarrow_1 3,\ 3 \rightarrow_2 \infty\}$, $\{0 \rightarrow_1 1,\ 3 \rightarrow_1 2,\ 1 \rightarrow_2 2, 2 \rightarrow_3 \infty\}$) which extends to ($\{ 0 \rightarrow_0 1, 2 \rightarrow_0 3, 1 \rightarrow_1 3, 3 \rightarrow_2 \infty \}$, $\{0 \rightarrow_1 1,\ 3 \rightarrow_1 2, 1 \rightarrow_2 2, 2 \rightarrow_3 \infty\}$, $\{0 \rightarrow_2 1, 2 \rightarrow_2 3, 1 \rightarrow_3 3, 3 \rightarrow_4 \infty\}$, $\{0 \rightarrow_3 1, 3 \rightarrow_3 2,\ 1 \rightarrow_4 2, 2 \rightarrow_5 \infty\}$, $\{0 \rightarrow_4 1, 2 \rightarrow_4 3, 1 \rightarrow_5 3, 3 \rightarrow_6 \infty\}$, $\{0 \rightarrow_5 1, 3 \rightarrow_5 2, 1 \rightarrow_6 2,\ 2 \rightarrow_7 \infty\}$, ...).

In general, even-numbered blocks will follow the first pattern, and odd-numbered blocks will follow the second. Note that the constraints on communication are satisfied: for a given subscript, no node number appears to the left of the arrow or the right of the arrow twice anywhere in the list; this corresponds to one input, one read, one addition, and one output per step. We now consider the sets individually. Every number appears on the left exactly once in every set, this guarantees that every input is considered as part of the sum. Every node number appears on the left with an index one greater than the largest index for which it appears on the right, thus communications are timely; a sum, once fully computed, is transmitted on the next step. Every node number appearing on the right in some step in some set appears on the right on consecutive steps in that set until it appears on the left in that set; this guarantees that no buffering is required at a node except for its accumulator; considering, say, node 3, we see that node 3 receives block 0

from node 2, and sends nothing on step 0. It receives block 0 from node 1, and sends block 1 to node 2 on step 1. It receives block 2 from node 1, and sends block 0 to the output on step 2. And so on. Even in more complicated schedules, while a node is receiving different values for the same block, it will be sending consecutive disk values off to other nodes, until it receives all of the values for the one block for which it is currently accumulating values. Every node number other than 0 appears on the right of some arrow for a given subscript, and 0 never does; this shows that we're maximally using our receiving capacity, except for node 0. Note further that the sums arriving at a node consist of the sum of values from a contiguous block of nodes immediately preceding or immediately following the set of nodes present in the sum currently known to a node; by adding on the left or right as needed, we don't require a commutative addition operator, and so this technique works in arbitrary monoids.

To perform the recursion, let's suppose that the extended schedule for n nodes is $(T_0, T_1, ..., T_k, ...)$. For any set T_k from the schedule, define Double(T_k) to be the result of replacing every occurrence of node number i by $2i+1$, incrementing all subscripts by one, and adding in links $2i \rightarrow_k 2i+1$, except if a previously doubled set contains an edge $x \rightarrow_k 2i+1$, or an edge $2i \rightarrow_k x$, in which case we instead replace all occurrences of i by $2i$, and add link $2i+1 \rightarrow_k 2i$. In every tree so constructed, the first communication step connects the paired nodes $2i$ and $2i+1$, and edges in the original schedule turn into edges between paired nodes in different pairs. We easily see inductively that the property of consecutive summation is preserved. Each doubled tree is individually a valid tree for computing a sum; half of the elements appear on the left with subscript k, and the other half participate in a tree which previously worked, appropriately relabeled. Suppose that some edges conflict between trees. This means that some earlier tree contains an edge $x \rightarrow_{k+m} y$, and this tree contains either $x \rightarrow_{k+m} z$ or $z \rightarrow_{k+m} y$, for some x, y, z, and m. We know that x and y must not be paired; all edges between paired nodes happen in the first step. Thus, an edge between $x/2$ and $y/2$ existed in the original sequence. If $m > 0$, the same is true of x,z or z,y, contradicting our inductive hypothesis. Hence $m = 0$, but then we avoided collisions by construction. This would fail only if previous trees contain edges which force our decision to go both ways, but these edges would come from the original tree. The only way this could happen is with edges $x \rightarrow_k y$ and $y \rightarrow_k z$, or the reverse. Suppose the first edge occurs earlier, and that the second comes from tree $j < k$. By another inductive hypothesis, y receives messages in the earlier tree on all steps, which would have prevented this outcome.

We claim, with the proof omitted, that this construction is cyclic, in that the extended lists can be reduced to lists of length $2^{\lceil \log_2 \log_2 n \rceil}$, and then extended.

3.5 Reducing the number of nodes in in-place binary trees

We next show that the number of nodes can be reduced by any amount up to half, so that arbitrary numbers of nodes can be handled, based on reducing power-of-two sized trees. The construction presented above exactly doubles the number of nodes, leading to solutions only for numbers of nodes equal to powers of two. We observe that we can reduce the doubled sets produced above by picking any pair $(2i, 2i+1)$ introduced in doubling which we have not yet reduced. Delete node $2i+1$ from all of the sets, delete edges connecting $2i$ and $2i+1$, and replace any remaining instances of node $2i+1$ by $2i$. Having done so, we have reduced the number of nodes by one; we can renumber all nodes densely in a post-processing step after we complete all desired reductions. This can not introduce any collisions; the trees we constructed above, contain exactly one edge per labeling of the arrow between $2i$ and $2i+1$, at most one other edge per labeling pointing to one of the pair, and at most one other edge per labeling leaving one of the pair.

3.6 Rooted binary and ternary trees: a hint towards a simple near-optimal solution

Consider an alternative construction for a binary tree to that presented above; one in which for every two child nodes, there is an explicit parent node. In a naive version of this approach, each communication from children to parent takes two units of time; one for the left child to send a value, and one for the right. Thus, the total latency of this approach is $2\lceil \log_2 (n+1) \rceil - 1$. The throughput is half that of the previous scheme, because the root node produces an output only every other step. This tree most naturally has $2^k - 1$ nodes; we can reduce this to a smaller value by discarding leaves as desired.

In an extended version of this paper, we will present techniques for constructing multiple binary or ternary trees with explicit parent nodes in which two or three trees suffice for all numbers of nodes to produce full throughput solutions, with latency no more than a factor of two worse than optimal.

References

[1] ALVAREZ, G. A., BURKHARD, W. A., AND CRISTIAN, F. Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering. In *Proceedings of the 24th International Symposium on Computer Architecture* (Denver, CO, June 1997), ACM, pp. 62–72.

[2] BLAUM, M., BRADY, J., BRUCK, J., AND MENON, J. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computers 44*, 2 (1995), 192–202.

[3] LITWIN, W., AND SCHWARZ, T. LH*$_{RS}$: A high-availability scalable distributed data structure using Reed Solomon codes. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (Dallas, TX, May 2000), ACM, pp. 237–248.

[4] PLANK, J. S. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software—Practice and Experience (SPE) 27*, 9 (Sept. 1997), 995–1012. Correction in James S. Plank and Ying Ding, Technical Report UT-CS-03-504, U Tennessee, 2003.

Mark Manasse and Chandu Thekkath are Senior Researchers at Microsoft Research – Silicon Valley; Alice Silverberg is a Professor at the University of California at Irvine, and was a consultant to Microsoft at the time of this research. E-mail: manasse@microsoft.com, thekkath@acm.org, asilverb@uci.edu