A Failure Correction Technique for Parallel Storage Devices with Minimal Device Overhead

James S. Plank

Department of Computer Science
Univeristy of Tennessee
Knoxville, TN 37996
plank@cs.utk.edu

Joel Friedman

Department of Mathematics
Univeristy of British Columbia
Vancouver, BC V6T 172 Canada
jf@math.ubc.edu

Kai Li

Department of Computer Science
Princeton University
Princeton, NJ 08544
li@cs.princeton.edu

August 5, 1994

Technical Report CS-94-243
Department of Computer Science
University of Tennessee
Knoxville, TN 37996

A Failure Correction Technique for Parallel Storage Devices with Minimal Device Overhead

James S. Plank* Joel Friedman† Kai Li‡

Abstract

A common technique for providing reliability in parallel storage designs, network file systems, and diskless checkpointing systems is the $N + 1$-Parity approach. This approach is simple in coding, but requires an excess number of additional "checksum" storage devices to recover more than one arbitrary device failure.

This paper presents a general method to recover from the failure of m arbitrary storage devices with the addition of exactly m checksum devices. The method is an application of Reed-Solomon codes, and can be viewed as a generalization of $N + 1$-Parity.

This paper has two goals concerning this algorithm. First, it provides a complete specification of how to code this problem with this algorithm. To the authors' knowledge, this is the first such specification. Second, we have implemented the coding and recovery algorithm in software and shown that the method is efficient, general, and practical.

1 Introduction

Error-correcting codes have been around for decades [Ber68, PW72, MS77]. However, the technique of distributing data among multiple storage devices to achieve high-bandwidth input and output, and using one or more error-correcting devices for failure recovery is relatively new. It came to the fore with "Redundant Arrays of Inexpensive Disks (RAID)," where batteries of small, inexpensive disks were used to combine high storage capacity, bandwidth, and reliability all at a low cost [PGK88, Gib92]. Since then, the technique has been used to design multicomputer and network file systems with high reliability and bandwidth [CLVW93, HO93], and to design fast checkpointing systems where extra processors provide reliability instead of disks [PL94].

The most common technique for this is called "$N + 1$-Parity." With $N + 1$-Parity, one can efficiently and reliably store Nk bytes of data in parallel on $N + 1$ data storage devices, each of which holds k bytes. The first N devices store the bytes themselves, and the last device stores a parity code of the others: Its k bytes of storage are calculated as the bitwise exclusive or (`XOR`) of the others, and thus if any one of the $N + 1$ devices fails, it can be reconstructed as the `XOR` of the remaining N devices. The main advantage of this approach is its simplicity. It requires one extra storage device, and one extra `write` operation per `write` to any single device. Its main disadvantage is that it cannot recover from more than one simultaneous failure.

Several methods have been proposed to extend the $N + 1$-Parity method to recover multiple failed devices. Gibson *et al* [GHK+89] describe several ways of using extra storage devices to store the parity of different subsets of data storage devices. Their simplest method is to partition the N data storage devices into disjoint groups and to use one parity device per group. This method can recover exactly one failure in each group. To allow recovery from two-device failures in each group, they arrange the groups in a two dimensional grid, and allot a parity device for every row and column. This is called "2d-parity", and requires a minimum of $2\sqrt{N}$ parity devices. Figure 1 shows an example of this method for $N = 9$. It can recover from any two device failures but cannot recover from arbitrary three device failures (such as devices A, C1, and C4).

The extension of this approach to allow recovery from m-device failures in each group requires

*plank@cs.utk.edu, Department of Computer Science, University of Tennessee, Knoxville TN 37996.

†jf@math.ubc.ca, Department of Mathematics, University of British Columbia, Vancouver, BC V6T 172, Canada.

‡li@princeton.edu, Department of Computer Science, Princeton University, Princeton NJ 08544.

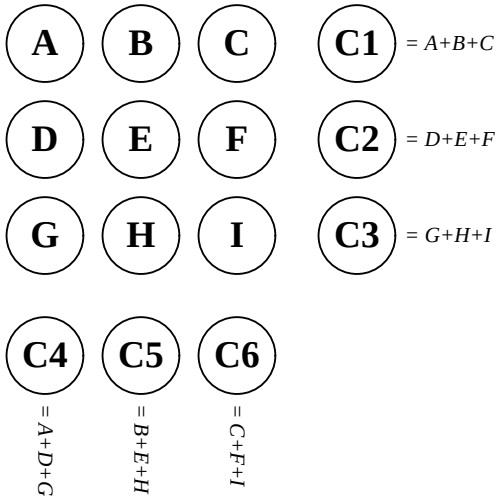

Figure 1: Providing 2-site Fault Tolerance with $2d$-parity

$mN^{\frac{m-1}{m}}$ extra parity devices and is called "m-dimensional parity." Each extra device stores the the parity of $\sqrt[m]{N}$ data devices. In the best case, the system can recover $mN^{\frac{m-1}{m}}$ device failures (for example, if each parity device fails); however, in the worst case, the system can recover only m failures. Although $N+1$-Parity based methods require minimal overhead in coding, they require far more than m devices to ensure the recovery from arbitrary m device failures. The authors state that m-dimensional parity is unreasonable for values of m greater than three [GHK+89].

Recently, Blaum, Brady, Bruck and Menon have described an algorithm called EVENODD, which uses parity operations to allow recovery from two-device failures with the addition of exactly two checksum devices [BBBM94]. This is the most efficient algorithm known to tolerate two-device failures. They are currently extending this algorithm to tolerate three and four-device failures as well.

This paper describes a general algorithm for tolerating the failure of any m storage devices with the addition of exactly m extra devices. We achieve the minimal number of extra devices by applying Reed-Solomon codes. Reed-Solomon codes are well-known in error-correcting coding theory, and have seen use in correcting errors on noisy communication lines [MS77, vL82, Wig88]. They are also known to be the default mechanism to provide m-device reliability with the addition of exactly m checksum devices [Gib92, BM93, BBBM94].

However, as this is not the "conventional" use of Reed-Solomon codes, their use in this regard is not well-documented: To the authors' knowledge, there is no complete specification of how to use Reed-Solomon coding for reliability from multiple device failures. The primary goal of this paper is to provide such a specification. A systems programmer should need no other references besides this paper to implement Reed-Solomon coding for reliability from multiple device failures.

Second, we analyze the trade-offs of this algorithm and show that the cost of computing checksums is small. The computation overhead of a checksum in this method requires two table lookups, two additions, two conditionals and a parity operation per data word. We have implemented both coding and recovery and experimented on a DEC Alpha workstation. The results show that this algorithm, while not as efficient as parity-based schemes, is indeed practical for many system applications. Moreover, for recovery from m-device failures with m checksum devices where $m > 4$, this is still the only general algorithm known.

2 Problem Specification and General Strategy

Let there be N storage devices, $D_1, D_2, \ldots, D_N$, each of which holds k bytes. We call these the "Data Devices." To these we add m more storage devices $C_1, C_2, \ldots, C_m$, each of which also holds k bytes. We call these the "Checksum Devices." The contents of each checksum device are calculated from the contents of the data devices. Our goal is to define the calculation of each C_i such that if any m of $D_1, D_2, \ldots, D_N, C_1, C_2, \ldots, C_m$ fail, then the contents of the failed devices can be reconstructed from the non-failed devices.

Formally, the failure model of our system is that of an *erasure*. When a device fails, it shuts down, and the system recognizes this shutting down. This is as opposed to an *error*, in which a device failure is manifested by storing and retrieving incorrect values, which are cannot by recognized by the system without some sort of embedded coding [PW72, Wig88].

The calculation of the contents of each checksum device C_i requires a separate function F_i. Figure 2 shows an example configuration using our algorithm (which we henceforth call *RS*) for $N = 9$ and $m = 2$, the same as in Figure 1. The coding on the checksum devices C_1 and C_2 is computed by using functions F_1 and F_2 respectively. It is clear that this configuration requires more complicated coding than parity and far fewer checksum devices than the 2d-parity method in Figure 1.

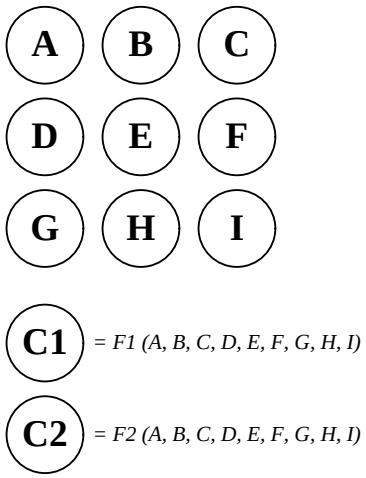

Figure 2: Providing 2-site Fault Tolerance with the RS Algorithm

All coding methods considered in this paper break up each storage device into *words*. The size of each word is w bits, w being chosen by the coding algorithm. The coding functions F_i operate on a word-by-word basis, as in Figure 3, and thus we can view our problem as consisting of N data words $d_1, \ldots, d_N$ and m checksum words $c_1, \ldots, c_m$ which are computed from the data words in such a way that the loss of any m words can be tolerated.

To compute a checksum word c_i for the checksum device C_i, we apply function F_i to to the corresponding data words on all data devices:

$$c_i = F_i(d_1, d_2, \ldots, d_N).$$

If a data word on device D_j is updated from d_j to d'_j, then each checksum word c_i must be recomputed by using a function $G_{i,j}$ such that:

$$c'_i = G_{i,j}(d_j, d'_j, c_i).$$

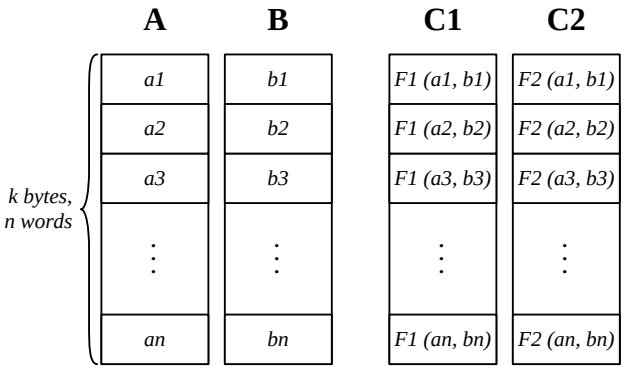

Figure 3: Breaking the Storage Devices into Words

When up to m devices fail, we reconstruct the system as follows. First, for each failed data device D_j, we construct a function to restore the words d_j from the words for the non-failed devices. When that is completed, we recompute any failed checksum devices C_i with F_i.

For example, suppose $m = 1$. Then we can describe $N + 1$-Parity in the above terms. There is one checksum device C_1, and words consist of one bit (i.e. $w = 1$). To compute each checksum word c_1, we take the parity (`XOR`) of the data words:

$$c_1 = F_1(d_1, \ldots, d_N) = d_1 \oplus d_2 \oplus \ldots \oplus d_N.$$

If a word of data device D_j changes from d_j to d'_j, then c_1 is recalculated from the parity of its old value and the two data words:

$$c'_1 = G_{1,j}(d_j, d'_j, c_1) = c_1 \oplus d_j \oplus d'_j.$$

If a device d_j fails, then it may be restored as the parity of the remaining devices:

$$d_i = d_1 \oplus \ldots \oplus d_{j-1} \oplus d_{j+1} \oplus \ldots \oplus d_N \oplus c_1.$$

In such a way, the system is resilient to any one device failure.

To restate, our problem is defined as follows. We are given N data words $d_1, d_2, \ldots, d_N$ all of size w. We will define functions F and G which we use to calculate and maintain the checksum words $c_1, c_2, \ldots, c_m$. We will then describe how to reconstruct the words of any lost data device when up to m devices fail. Once the data words are reconstructed, the checksum words can be recomputed from the data words and F. Thus, the entire system is reconstructed.

3 Overview of the RS Algorithm

There are three main aspects of the RS algorithm: The use of the Vandermonde matrix to calculate and maintain checksum words, the use of Gaussian Elimination to recover from failures, and the use of Galois Fields to perform arithmetic. Each is detailed below:

Calculating and Maintaining Checksum Words

We will define each function F_i as a linear combination of the data words:

$$c_i = F_i(d_1, d_2, \ldots, d_N) = \sum_{j=1}^{N} d_j f_{i,j}$$

In other words, if we represent the data and checksum words as the vectors D and C, and the functions F_i as rows of the matrix F, then the state of the system adheres to the following equation:

$$FD = C.$$

We define F to be the $m \times N$ Vandermonde matrix: $f_{i,j} = j^{i-1}$, and thus the above equation becomes:

$$\begin{bmatrix} f_{1,1} & f_{1,2} & \ldots & f_{1,N} \\ f_{2,1} & f_{2,2} & \ldots & f_{2,N} \\ \vdots & \vdots & & \vdots \\ f_{m,1} & f_{m,2} & \ldots & f_{m,N} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_N \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & 2 & 3 & \ldots & N \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{m-1} & 3^{m-1} & \ldots & N^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_N \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}.$$

When one of the data words d_j changes to d'_j, then each of the checksum words must be changed as well. This can be effected by subtracting out the portion of the checksum word that corresponds to d_j, and adding the required amount for d'_j. Thus, $G_{i,j}$ is defined as follows:

$$c'_i = G_{i,j}(d_j, d'_j, c_i) = c_i + f_{i,j}(d'_j - d_j).$$

Thus, the calculation and maintenance of checksum words can be done by simple arithmetic (however, it is a special kind of arithmetic, as explained below).

Recovering From Failures

To explain recovery from errors, we define the matrix A and the vector E as follows: $A = [\frac{I}{F}]$, and $E = [\frac{D}{C}]$. Then we have the following equation ($AD = E$):

$$\begin{bmatrix} 1 & 0 & 0 & \ldots & 0 \\ 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \ldots & 1 \\ 1 & 1 & 1 & \ldots & 1 \\ 1 & 2 & 3 & \ldots & N \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{m-1} & 3^{m-1} & \ldots & N^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_N \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_N \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}.$$

We can view each device in the system as having a corresponding row of the matrix A and of the vector E. When a device fails, we can reflect it by deleting its row from A and from E. What results a new matrix A' and a new vector E' that adhere to the equation:

$$A'D = E'.$$

Suppose exactly m devices fail. Then A' is a $N \times N$ matrix. Because matrix F is defined to be a Vandermonde matrix, every subset of N rows of matrix A is guaranteed to be linearly independent. Thus, the matrix A' is non-singular, and the the values of D may be calulated from $A'D = E'$ using Gaussian Elimination. Hence all data devices can be recovered.

Once the values of D are obtained, the values of any failed C_i may be recomputed from D. It should be obvious that if fewer than m devices fail, the system may be recovered in the same manner, choosing any N rows of A' to perform the Gaussian Elimination. Thus, the system can tolerate any number of device failures up to m.

Arithmetic over Galois Fields

A major concern of the RS algorithm is that the domain and range of our computation are binary words of a fixed length w. Although the above algebra is guaranteed to be correct when all the elements are infinite precision real numbers, we must make sure that it is correct for these fixed-size words. A common error in dealing with these

codes is to perform all arithmetic over the integers modulo 2^w. This *does not work*, as division is not defined for all pairs of elements (for example, $(3 \div 2)$ is undefined modulo 4), rendering the Gaussian Elimimation unsolvable in many cases. Instead, we must perform addition and multiplication over a *field* with more than $N+m$ elements [PW72].

Fields with 2^w elements are called *Galois Fields* (denoted $GF(2^w)$), and are a fundamental topic in algebra (e.g. [Her75, MS77, vL82]). The key of this section is to define how to perform addition, subtraction, multiplication, and division efficiently over a Galois Field. We will give such a description without fully explaining Galois Fields in general. Appendix A contains a more detailed description of Galois Fields, and provides justification for the arithmetic algorithms in this section.

The elements of $GF(2^w)$ are the integers from zero to $2^w - 1$. Thus, they may be represented by all binary words of length w. As detailed in Appendix A, arithmetic of elements in a Galois Field is analagous to polynomial arithmetic modulo a primitive polynomial of degree w over $GF(2)$. However, we can describe this arithmetic without going into the details of such polynomials.

Addition and subtraction of elements of $GF(2^w)$ are simple. They are simply a bitwise exclusive-or. For example, in $GF(16)$:

$$11 + 7 = 1011 \oplus 0111 = 1100 = 12.$$

$$11 - 7 = 1011 \oplus 0111 = 1100 = 12.$$

Multiplication and division are more complex. They require two mapping tables, each of length 2^w, which are analagous to logarithm tables for real numbers:

- `gflog[ NW ]`: A table that maps an integer to its logarithm in the Galois Field. (`NW` $= 2^w$.)
- `gfilog[ NW ]`: An inverse table table that maps an integer to its inverse logarithm in the Galois Field.

With these two tables, we can multiply two elements of $GF(2^w)$ by adding their logs and then taking the inverse log, which yields the product. To divide two numbers, we instead subtract the logs. Figure 4 shows an implementation in C: This implementation makes use of the fact that the inverse log of an integer i is equal to the inverse log of $i \bmod (2^w - 1)$. (This fact is explained in Appendix A).

```
int mult(int a, int b)
{
  int sum_log;

  if (a == 0 || b == 0) return 0;
  sum_log = gflog[a] + gflog[b];
  if (sum_log >= NW-1) sum_log -= NW-1;
  return gfilog[sum_log];
}

int div(int a, int b)
{
  int diff_log;

  if (a == 0) return 0;
  if (b == 0) return -1; /*Can't divide by 0*/
  diff_log = gflog[a] - gflog[b];
  if (diff_log < 0) diff_log += NW-1;
  return gfilog[diff_log];
}
```

Figure 4: C Code for Multiplication and Division over $GF(2^w)$

As with regular logarithms, we must treat zero as a special case, as the logarithm of zero is $-\infty$. Unlike regular logarithms, the log of any non-zero element of a Galois Field is an integer, allowing for exact multiplication of Galois Field elements using these logarithm tables.

An important step, therefore, once w is chosed, is generating the logarithm tables for $GF(2^w)$. The algorithm to generate the logarithm and inverse logarithm tables for any w can be found in Appendix A. As an example, we include the tables for $GF(16)$ in Table 1:

i	0	1	2	3	4	5	6	7
gflog[i]	$-\infty$	0	1	4	2	8	5	10
gfilog[i]	1	2	4	8	3	6	12	11

i	8	9	10	11	12	13	14	15
gflog[i]	3	14	9	7	6	13	11	12
gfilog[i]	5	10	7	14	15	13	9	1

Table 1: Logarithm tables for $GF(16)$

For example, in $GF(16)$:

$$3 * 7 = \texttt{gfilog[4+10]} = \texttt{gfilog[14]} = 9$$
$$13 * 10 = \texttt{gfilog[13+9]} = \texttt{gfilog[7]} = 11$$

$$13 \div 10 = \texttt{gfilog[13-9]} = \texttt{gfilog[4]} = 3$$
$$3 \div 7 = \texttt{gfilog[4-10]} = \texttt{gfilog[9]} = 14$$

Therefore, a multiplication or division requires simply one conditional, three table lookups (two logarithm table lookups and one inverse table lookup), an addition or subtraction, and a modulo operation. For efficiency in the C code above, we implement the modulo operation as a conditional and a subtraction or addition.

4 The Algorithm Summarized

Given N data devices and m checksum devices, the RS algorithm for making them fault-tolerant to up to m failures is as follows.

1. Choose a value of w such that $2^w \geq N + m$. It is easiest to choose $w = 8$ or $w = 16$, as words then fall directly on byte boundaries. Note that with $w = 16$, $N + m$ can be as large as $65,536$.
2. Set up the tables `gflog` and `gfilog` as described in Appendix A.
3. Set up the matrix F to be the $m \times N$ Vandermonde matrix: $f_{i,j} = j^{i-1}$ (for $1 \leq i \leq m, 1 \leq j \leq N$) where multiplication is performed over $GF(2^w)$.
4. Use the matrix F to calculate and maintain each word of the checksum devices from the words of the data devices. Again, all addition and multiplication is performed over $GF(2^w)$.
5. If any number of devices up to m fail, then they can be restored in the following manner. Choose any N of the remaining devices, and construct the matrix A' and vector E' as defined previously. Then solve for D in $A'D = E'$. This enables the data devices to be restored. Once the data devices are restored, the failed checksum devices may be recalculated using the matrix F.

5 An Example

As an example, suppose we have three data devices and three checksum devices, each of which holds one megabyte. Then $N = 3$, and $m = 3$. We choose w to be four, since $2^w > N + m$, and since we can use the logarithm tables in Table 1 to illustrate multiplication.

Next, we set up `gflog` and `gfilog` to be as in Table 1. We construct F to be a 3×3 Vandermonde matrix, defined over $GF(16)$:

$$F = \begin{bmatrix} 1^0 & 2^0 & 3^0 \\ 1^1 & 2^1 & 3^2 \\ 1^2 & 2^2 & 3^2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 4 & 5 \end{bmatrix}$$

Now, we can calculate each word of each checksum device using $FD = C$. For example, suppose the first word of D_1 is 3, the first word of D_2 is 13, and the first word of D_3 is 9. Then we use F to calculate the first words of C_1, C_2, and C_3:

$$\begin{aligned} C_1 &= (1)(3) \oplus (1)(13) \oplus (1)(9) \\ &= 3 \oplus 13 \oplus 9 \\ &= 0011 \oplus 1101 \oplus 1001 = 0111 = 7 \\ C_2 &= (1)(3) \oplus (2)(13) \oplus (3)(9) \\ &= 3 \oplus 9 \oplus 8 \\ &= 0011 \oplus 1001 \oplus 1000 = 0010 = 2 \\ C_3 &= (1)(3) \oplus (4)(13) \oplus (5)(9) \\ &= 3 \oplus 1 \oplus 11 \\ &= 0011 \oplus 0001 \oplus 1011 = 1001 = 9 \end{aligned}$$

Suppose we change D_2 to be 1. Then D_2 sends the value $(1 - 13) = (0001 \oplus 1101) = 12$ to each checksum device, which uses this value to recompute its checksum:

$$\begin{aligned} C_1 &= 7 \oplus (1)(12) = 0111 \oplus 1100 = 11 \\ C_2 &= 2 \oplus (2)(12) = 2 \oplus 11 = 0010 \oplus 1011 = 9 \\ C_3 &= 9 \oplus (4)(12) = 9 \oplus 5 = 1001 \oplus 0101 = 12 \end{aligned}$$

Suppose now that devices D_2, D_3, and C_3 are lost. Then we delete the rows of A and E corresponding to D_1, D_2, and C_3 to get $A'D = E$:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{bmatrix} D = \begin{bmatrix} 3 \\ 11 \\ 9 \end{bmatrix}$$

By applying Gaussian elimination, we can invert A' to yield the following equation: $D = (A')^{-1}E'$, or:

$$D = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 1 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 11 \\ 9 \end{bmatrix}.$$

From this, we get:

$$D_2 = (2)(3) \oplus (3)(11) \oplus (1)(9) = 6 \oplus 14 \oplus 9 = 1$$

$$D_3 = (3)(3) \oplus (2)(11) \oplus (1)(9) = 5 \oplus 5 \oplus 9 = 9$$

And then:

$$C_3 = (1)(3) \oplus (4)(1) \oplus (5)(9) = 3 \oplus 4 \oplus 11 = 12$$

Thus, the system is recovered.

6 Analysis

In this section, we analyze the cost of computing and maintaining checksums and the cost of recovery. We also detail our implementations and report experimental results.

6.1 Cost of Coding

The cost of computing a checksum is usually measured by a metric called the *update penalty* of the RS algorithm. Update penalty is the extra cost of maintaining the coding for fault-tolerance per data device update. The optimal update penalty for a system tolerating m faults is m updates to checksum devices. All the algorithms presented by Gibson *et al* have this optimal update penalty [GHK+89], as does the algorithm presented in this paper. The algorithms differ, however, in the complexity of each checksum device update.

In the parity-based algorithms, when a word of a data device is changed from d to d', the difference of the two is calculated $(d \oplus d')$, and sent to the m checksum devices, which `XOR` this difference with their copy. Thus, an update to a data word consists of one parity operation performed at the data device, followed by one parity operation performed in parallel by each of m checksum devices.

In the RS algorithm, things proceed similarly. When a data device changes a word from d to d', the difference is again calculated $(d \oplus d')$, and sent to the m checksum devices. Each checksum device however, now multiplies this difference by a constant $(f_{i,j})$, and then adds it to its device with a parity operation. Thus, an update to a data word consists of one parity operation performed at the data device, followed by one multiplication and one parity operation performed in parallel by each of m checksum devices. As such, the cost of coding is only slightly more complex than parity.

To assess exactly how much of a penalty is incurred by this extra multiplication, we implemented the RS algorithm for coding checksum devices. The code emulates N data devices with random contents and calculates the contents of m checksum devices. Each device holds one megabyte. For the RS algorithm, we used a value of $w = 16$, because that is the largest value of w for which the logarithm tables are a reasonable size (128K each).

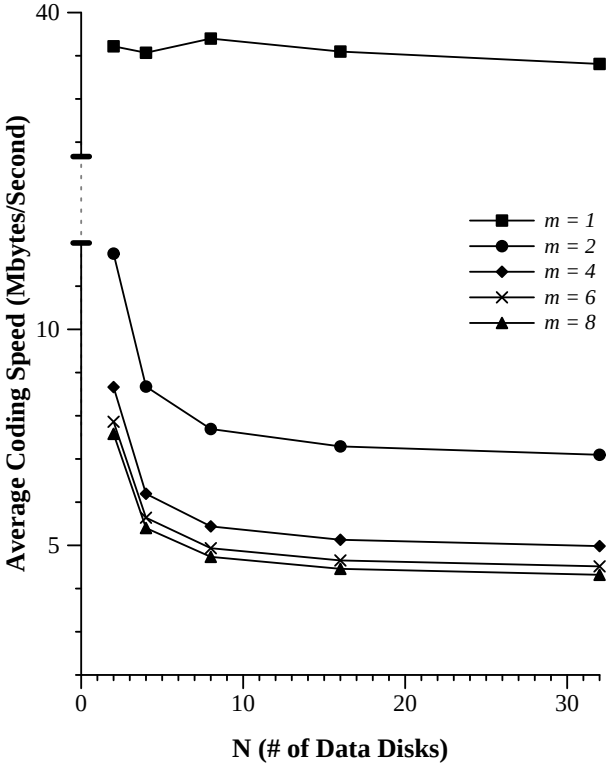

Figure 5: Average Speed of Coding Checksum Devices on the DEC Alpha

The graph in Figure 5 shows the average speed of coding for many values of N and m on a DEC Alpha workstation. Each point on the graph represents the speed of coding N megabytes of data on each of m check devices, using the RS algorithm over $GF(2^{16})$. All points are the average of ten runs. The first result to notice is the line for $m = 1$. When $m = 1$, the RS algorithm is equivalent to $N + 1$-Parity. Thus, this line represents the speed of coding using straight `XOR`. As the Alpha can perform `XOR` on 64 bits in parallel, the speed of coding is extremely

fast.

For the other lines on the graph, the average speed of coding depends on both N and m. This is for the following reason: Whenever a checksum device C_i needs to update its checksum for a data device D_j, and $f_{i,j} = 1$, then no multiplication need be performed to update the checksum. Instead, only an `XOR` need be executed. Thus, the speed of coding gets slower when N and m get bigger because the associated Vandermonde matrix has proportionally fewer elements with values of one.

In the worst case, the speed of coding is just above four megabytes per second. This is roughly ten times slower than coding with straight `XOR`, which is to be expected, since a Galois Field multiplication has to perform two conditionals, two additions and two table lookups for each 16-bit quantity (since we code a megabyte at a time, we only need two table lookups per word, as `gflog[`$f_{i,j}$`]` can be looked up in advance). This speed of coding should be sufficient for disk controllers and checkpointing systems. Currently, few disks and operate at greater speeds.

One possible way to increase the speed of coding is to provide separate multiplication procedures for each value of $f_{i,j}$. As stated above, when $f_{i,j}$ is one, no multiplication is necessary. When $f_{i,j}$ is two, any value less than 2^{15} can be multiplied by a simple bit shift, and values greater than 2^{15} can be multiplied by a bit shift followed by an `XOR`. There are similar optimizations for other values of $f_{i,j}$.

Another obvious optimization for coding is to unroll the coding loop for each $f_{i,j}$. This optimization is reasonable for designing systems that have fixed N and m, and fixed block sizes. Such an optimization can eliminate the loops.

Finally, it should be noted that multiplication over Galois Fields can be implemented in hardware in a rather simple fashion [PW72]. Were a processor to provide such an operation in hardware, the speed of coding would be much closer to the speed of a parity operation.[1]

[1] We should note that implementing division or logarithm functions in hardware is a much harder problem [CW94]. However, a hardware multiplier for $GF(2^{16})$ would significantly speed up coding, while still allowing us to use the table-based division for recovery.

6.2 Cost of Recovery

In the RS algorithm, recovery consists of performing Gaussian Elimination of an equation $A'D = E'$ so that $(A')^{-1}$ is determined. Then, the contents of all the failed data devices may be calculated as a linear combination of the devices in E'. Thus, recovery has two parts: the Gaussian Elimination, and the recalculation. Since at least $N - m$ rows of A' are identity rows, the Gaussian Elimination takes $O(m^2 N)$ steps. The recalculation of the data devices takes N multiplications and N `XOR`'s per word in the device. Since the number of words per device is most likely to be much larger than m^2, the recalculation should be the dominant part of recovery.

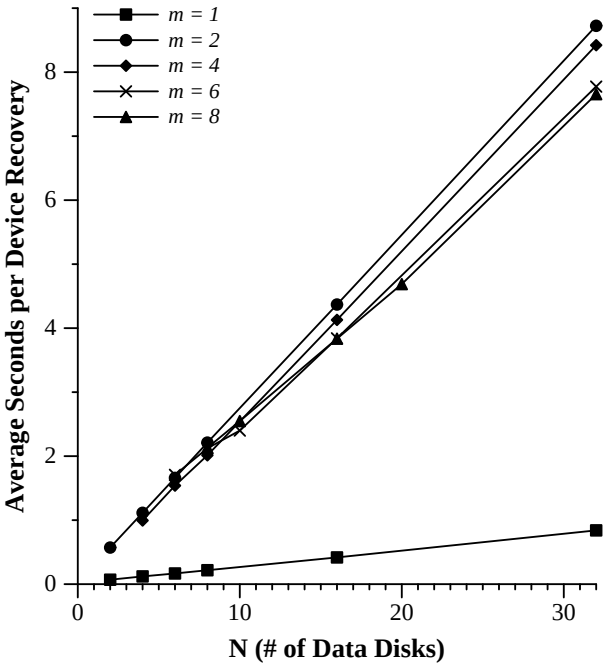

Figure 6: Average Time of Failed Device Recovery

The data in Figure 6 corroborates these claims. Here, each data point represents the average time to recover a failed device in a system with N data devices and m checksum devices, and m failures. As before, each device holds one megabyte. Each recovery point in the graph includes a full Gaussian Elimination; however, no Gaussian Elimination in the graph took longer than eight milliseconds. Thus, as stated above, the dominant cost of recovery is recalculating the contents of the data devices, which takes N `XOR`'s and N multiplications (or just N `XOR`'s in the case $m = 1$) for each word in the

restored device.

In comparison to the parity-based algorithms of Gibson *et al*, the recovery of the RS algorithm is indeed more complex. The reason is twofold: First, the recalculation of data devices involves the contents of N storage devices in the RS algorithm, as opposed to $\sqrt[m]{N}$ for m-dimensional parity. Second, the recalculation itself involves both multiplication and parity, as opposed to just parity operations. However, as recovery is by far the least frequent operation in a system, the extra complexity added by the RS algorithm should not contribute to degraded system performance. It is indeed in recovery where we can afford to absorb some extra complexity in order to minimize the number of extra check devices.

7 Conclusion

We have presented an algorithm to recover from the failure of m arbitrary storage devices with the addition of a minimum number of checksum devices, exactly m. This is a substantial improvement over the m-dimensional parity method which requires $mN^{\frac{m-1}{m}}$ checksum devices.

The tradeoff of our method is to do a bit more computing to produce a checksum than the straight parity-based methods. In our method, computing a checksum requires three table lookups, an addition or subtraction, a modulo operation and an `XOR` operation for each data word, whereas the parity-based methods require only an `XOR` operation. Our software implementation and experiments show that the coding overhead is small and that this method is practical.

This algorithm is useful for large parallel storage system designs where recovering multiple device failures is necessary and individual devices are expensive. The application domain includes large scale disk arrays, striped network and multicomputer file systems, and diskless checkpointing systems.

8 Acknowledgements

The authors thank Michael Vose and Heather Booth for their helpful discussions. Kai Li is supported by ARPA and ONR under contracts N00014-91-J-4039, and by Intel Supercomputer Systems Division.

References

[BBBM94] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. In *21st Annual International Symposium on Computer Architecture*, pages 245—254, Chicago, IL, April 1994.

[Ber68] E. R. Berlekamp. *Algebraic Coding Theory*. McGraw-Hill, New York, 1968.

[BM93] W. A. Burkhard and J. Menon. Disk array storage system reliability. In *23rd International Symposium on Fault-Tolerant Computing*, pages 432–441, Toulouse, France, June 1993.

[CLVW93] P. Cao, S. B. Lim, S. Venkataraman, and J. Wilkes. The TickerTAIP parallel RAID architecture. In *20th International Symposium on Computer Architecture*, 1993.

[CW94] D. W. Clark and L-J. Weng. Maximal and near-maximal shift register sequences: Efficient event counters and easy discrete logarithms. *IEEE Transactions on Computers*, to appear, 1994.

[GHK+89] G. A. Gibson, L. Hellerstein, R. M. Karp, R. H. Katz, and D. A. Patterson. Failure correction techniques for large disk arrays. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–132, Boston, MA, April 1989.

[Gib92] G. A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. The MIT Press, Cambridge, Massachusetts, 1992.

[Her75] I. N. Herstein. *Topis in Algebra, Second Edition*. Xerox College Publishing, Lexington, Massachusetts, 1975.

[HO93] J. H. Hartman and J. K. Ousterhout. The zebra striped network file system. *Operating Systems Review*, 27(5):29–43, December 1993.

[MS77] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes, Part I*. North-Holland Publishing Company, Amsterdam, New York, Oxford, 1977.

[PGK88] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *1988 ACM Conference on Management of Data*, pages 109–116, June 1988.

[PL94] J. S. Plank and K. Li. Faster checkpointing with $N + 1$ parity. In *24th International Symposium on Fault-Tolerant Computing*, pages 288–297, Austin, TX, June 1994.

[PW72] W. W. Peterson and E. J. Weldon, Jr. *Error-Correcting Codes, Second Edition*. The MIT Press, Cambridge, Massachusetts, 1972.

[vL82] J. H. van Lint. *Introduction to Coding Theory*. Springer-Verlag, New York, 1982.

[Wig88] D. Wiggert. *Codes for Error Control and Synchronization*. Artech House, Inc., Norwood, Massachusetts, 1988.

Appendix A: Galois Fields, as Applied to this Algorithm

Galois Fields are a fundamental topic of algebra, and are given a full treatment in a number of texts [Her75, MS77, vL82]). This Appendix does not attempt to rigorously define and prove all the properties of Galois Fields necessary for this algorithm. Instead, our goal is to give enough information about Galois Fields so that anyone desiring to implement this algorithm will have a good intuition concerning the underlying theory.

A *field* $GF(n)$ is a set of n elements closed under addition and multiplication, for which every element has an additive and multiplicative inverse (except for the 0 element which has no multiplicative inverse). For example, the field $GF(2)$ can be represented as the set $\{0, 1\}$, where addition and multiplication are both performed modulo 2 (i.e. addition is `XOR`, and multiplication is the bit operator `AND`). Similarly, if n is a prime number, then we can represent the field $GF(n)$ to be the set $\{0, 1, \ldots, n-1\}$ where addition and multiplication are both performed modulo n.

However, suppose $n > 1$ is not a prime. Then the set $\{0, 1, \ldots, n-1\}$ where addition and multiplication are both performed modulo n is *not a field*. For example, let n be four. Then the set $\{0, 1, 2, 3\}$ is indeed closed under addition and multiplication modulo 4, however, the element 2 has no multiplicative inverse (there is no $a \in \{0, 1, 2, 3\}$ such that $2a \equiv 1 \pmod 4$). Thus, we *cannot* perform our coding with binary words of size $w > 1$ using addition and multiplication modulo 2^w. Instead, we need to use Galois Fields.

To explain Galois Fields, we work with polynomials of x whose coefficients are in $GF(2)$. This means, for example, that if $r(x) = x + 1$, and $s(x) = x$, then $r(x) + s(x) = 1$. This is because

$$x + x = (1 + 1)x = 0x = 0.$$

Moreover, we will be taking such polynomials modulo other polynomials, using the following identity:

> If $r(x) \bmod q(x) = s(x)$, then $s(x)$ is a polynomial with a degree less than $q(x)$, and $r(x) = q(x)t(x) + s(x)$, where $t(x)$ is any polynomial of x.

Thus, for example, if $r(x) = x^2 + x$, and $q(x) = x^2 + 1$, then $r(x) \bmod q(x) = x + 1$.

Let $q(x)$ be a *primitive* polynomial of degree w whose coefficients are in $GF(2)$. This means that $q(x)$ cannot be factored, and that the polynomial x can be considered a *generator* of $GF(2^w)$. To see how x generates $GF(2^w)$, we start with the elements 0, 1, and x, and then continue to enumerate the elements by multiplying the last element by x and taking the result modulo $q(x)$ if it has a degree $\geq w$. This enumeration will end at 2^w elements – the last element multiplied by x mod $q(x)$ will equal 1.

For example, suppose $w = 2$, and $q(x) = x^2 + x + 1$. To enumerate $GF(4)$ we start with the three elements 0, 1, and x, then then continue with $x^2 \bmod q(x) = x + 1$. Thus we have four elements: $\{0, 1, x, x+1\}$. If we continue, we see that $(x+1)x \bmod q(x) = x^2 + x \bmod q(x) = 1$, thus ending the enumeration.

The field $GF(2^w)$ is constructed by finding a primitive polynomial $q(x)$ of degree w over $GF(2)$, and then enumerating the elements (which are polynomials) with the generator x. Addition in this field is performed using polynomial addition, and multiplication is perfomed using polynomial multiplication and taking the result modulo $q(x)$. Such a field is typically written $GF(2^w) = GF(2)[x]/q(x)$.

Now, to use $GF(2^w)$ in the RS algorithm, we need to map the elements of $GF(2^w)$ to binary words of size w. Let $r(x)$ be a polynomial in $GF(2^w)$. Then we can map $r(x)$ to a binary word b of size w by setting the ith bit of b to the coefficient of x^i in $r(x)$. For example, in $GF(4) = GF(2)[x]/x^2 + x + 1$, we get the following table:

Generated Element of $GF(4)$	Polynomial Element of $GF(4)$	Binary Element b of $GF(4)$	Decimal Representation of b
0	0	00	0
x^0	1	01	1
x^1	x	10	2
x^2	$x + 1$	11	3

Addition of binary elements of $GF(2^w)$ can be performed by bitwise exclusive or. Multiplication is a little more difficult. One must convert the binary numbers to their polynomial elements, multiply the polynomials modulo $q(x)$, and then convert the answer back to binary. This can be implemented, in a simple fashion, by using the two logarithm tables described in Section 3: one that maps from a binary element b to power j such that x^j is equivalent to b (this is the `gflog` table, and is referred to in the literature as a "discrete logarithm"), and one that maps from a power j to its binary element b. Each table will have $2^w - 1$ elements (there is no j such that $x^j = 0$). Multiplication then consists of converting each binary element to its discrete logarithm, then adding the logarithms modulo $2^w - 1$ (this is equivalent to multiplying the polynomials modulo $q(x)$) and converting the result back to a binary element. Division is performed in the same manner, except the logarithms are subtracted instead of added. Obviously, elements where $b = 0$ must be treated as special cases. Therefore, multiplication and division of two binary elements takes three table lookups and a modular addition.

Thus, to implement multiplication over $GF(2^w)$, we must first set up the tables `gflog` and `gfilog`. To do this, we first need a primitive polynomial $q(x)$ of degree w over $GF(2^w)$. Such polynomials can be found in texts on error correcting codes [Ber68, PW72]. We list examples for powers of two up to 64 below:

$$\begin{aligned} w = 4: &\quad x^4 + x + 1 \\ w = 8: &\quad x^8 + x^4 + x^3 + x^2 + 1 \\ w = 16: &\quad x^{16} + x^{12} + x^3 + x + 1 \\ w = 32: &\quad x^{32} + x^{22} + x^2 + x + 1 \\ w = 64: &\quad x^{64} + x^4 + x^3 + x + 1 \end{aligned}$$

We then start with the element $x^0 = 1$, and enumerate all non-zero polynomials over $GF(2^w)$ by multiplying the last element by x, and taking the result modulo $q(x)$. This is done in Table 2 below for $GF(4)$, where $q(x) = x^4 + x + 1$.

It should be clear how this enumeration can be used to generate the `gflog` and `gfilog` arrays in Table 1. The C code in Figure 7 shows how to generate these arrays for $w = 16$:

Generated Element	Polynomial Element	Binary Element	Decimal Element
0	0	0000	0
x^0	1	0001	1
x^1	x	0010	2
x^2	x^2	0100	4
x^3	x^3	1000	8
x^4	$x+1$	0011	3
x^5	x^2+x	0110	6
x^6	x^3+x^2	1100	12
x^7	x^3+x+1	1011	11
x^8	x^2+1	0101	5
x^9	x^3+x	1010	10
x^{10}	x^2+x+1	0111	7
x^{11}	x^3+x^2+x	1110	14
x^{12}	x^3+x^2+x+1	1111	15
x^{13}	x^3+x^2+1	1101	13
x^{14}	x^3+1	1001	9
x^{15}	1	0001	1

Table 2: Enumeration of the elements of $GF(16)$

```
unsigned int q_x = 0210013;
unsigned int x_to_16 = 0200000;
unsigned short gflog[0200000];
unsigned short gfilog[0200000];

setup_tables()
{
  unsigned int binary_el, log;

  binary_el = 1;
  for (log = 0; log < 0177777; log++) {
    gflog[binary_el] = (short) log;
    gfilog[log] = (short) binary_el;
    b = b << 1;
    if (b & x_to_16) b = b ^ q_x;
  }
}
```

Figure 7: C Code for Generating the logarithm tables of $GF(2^{16})$