The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

A SIMPLE AND FAST VECTOR SYMBOL REED-SOLOMON BURST ERROR DECODING METHOD

A Thesis in

Computer Science and Engineering

by

Christopher Chang

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2008

The thesis of Christopher Chang was reviewed and approved* by the following:

John J. Metzner
Professor of Computer Science and Engineering
Thesis Adviser

Guohong Cao
Associate Professor of Computer Science and Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Error correction and detection play an important role in data transmission and storage systems. With the increasing demand for higher data transfer rates, reliability and efficiency is a necessity. A commonly used error correcting method is Reed-Solomon decoding. It is particularly attractive when dealing with bursts of errors. However, decoding complexity is a factor to consider when choosing codes.

There exists a faster and rather simple method in which vector symbol decoding is used along with Reed-Solomon codes to correct errors with a probability $\geq 1 - n(n-k)2^{-r}$. This paper discusses and simulates this novel technique and shows that it does in fact correct at a close to perfect success rate. Three cases of errors are tested, two different types of bursts of errors along with a non-burst scenario. We will see that the procedure described in this paper can uniquely correct a larger range of errors with less decoding complexity.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would first and foremost like to thank my thesis advisor, Dr. John J. Metzner for his continued and unlimited patience, guidance and support. I am also very grateful to my committee member Dr. Guohong Cao who has also assisted me through my time in Penn State. Last but not least, I would like to thank Penn State's Department of Computer Science and Engineering for the time I spent there.

Chapter 1

Introduction

Error correction has become a necessity in recent and future information systems. This is especially true now due to the demand for higher digital data transmission speeds and larger storage systems. Any type of interference or corruption on the data channel could lead to errors on the receiving end. Data that is transmitted from location A should look exactly like the data received at location B. This data also needs to appear at the receiver at a reasonable speed. In order to guarantee this reliability and efficiency, there are several different types of techniques that have been created to deal with it.

One powerful and popular type of error correcting scheme is Reed-Solomon decoding. This method is especially impressive when dealing with bursts of errors. Bursts of errors occur when a consecutive set of symbols are received in error. Since Reed-Solomon codes count a single error when one or all bits of a symbol are incorrect, it is able to correct a greater number of bursts than the average block code. However, there are obvious overheads when encoding and decoding any type of data. Therefore, previous works in this area have constantly aimed to provide faster methods of decoding and a greater range of error correction. Unfortunately, increasing the speed and range of error correction usually comes with the price of having a slight failure rate.

This paper describes a technique that goes about lowering the decoding complexity and

increasing the error correction capability of burst error decoding. It will prove to be able to correct bursts of errors with a probability $\geq 1 - n(n-k)2^{-r}$. For those that are not corrected, vector symbol decoding can also be used to find the error location vector and correct it accordingly, with the overhead of greater decoding complexity.

This simple and fast technique uses vector symbol decoding along with Reed-Solomon codes to correct errors. It traps the error pattern, which is of length up to $n - k$ in a set of registers in which the error values are the syndrome components. When the errors are trapped, we should see $n - k - e$ zero syndromes, where e is the number of errors. Three different scenarios are tested: a burst of $n - k - 1$ errors, a burst of $n - k$ errors where at least one of the components is error-free and a case of a non-burst set of errors such that there are at least two errors spaced greater than $n - k$ spaces apart. The goal of this paper is to implement the idea mentioned and show that it can correct errors with a close to perfect success rate.

The paper is organized as follows. Chapter 2 provides information about a powerful decoding scheme, vector symbol decoding. Chapter 3 goes into some background information about Reed-Solomon encoding and decoding. In chapter 4, we will see a discussion on burst error correction and some examples of prior work others have done in this area. The next section, chapter 5, describes the methodology of the technique that is implemented in this paper. Chapter 6 lists information about the simulator, the tools and the

parameters that were used to build and test this method. Chapter 7 analyzes and evaluates the results that were attained through that simulator with a number of tables and graphs. In chapter 8, the paper goes on to discuss future work that can be done. The paper ends by concluding in the final chapter, chapter 9.

Chapter 2

Vector Symbol Decoding

Vector symbol decoding [11, 12, 14] allows for greater error correcting capabilities when applied to normal block code decoding schemes. Data is represented in entities known as vector symbols instead of a sequence of bits. In addition, this technique is capable of operating on non-binary codes. This property then, in turn, allows for greater increased data transmission rates.

2.1 Concatenated Coding

Concatenated codes, first discovered by Forney in 1966 [5], is a way of constructing long powerful codes from shorter codes. This is can be done by using a binary code as an inner code and a non-binary code as the outer code. It is often used in both digital data communication and storage systems due to its ability to achieve higher reliability with lower decoding complexity. In many applications, Reed-Solomon codes, described later in the paper, are used as the outer codes.

With a (n_1, k_1) binary code and a (n_2, k_2) non-binary code, a concatenated code can be formed by having the symbols from $GF(2^{k1})$ of the non-binary code represented by k_2 bytes of k_1 binary symbols each. The k_2 bytes of k_1 digits each are encoded depending on the type of

non-binary code that is being used. These k_1 digits are then encoded into the rules set for the binary code.

2.2 Vector Symbol Coding

Vector symbols are similar to concatenated codes in which the inner code is a binary code and the outer code is the vector symbol. If each vector has *r* binary digits each, then the outer code is generated using an *r*-bit interleaving of the (n, k) binary code. Let *H* be the parity check matrix for the binary code, then the vector symbol code has *n* symbols of *r* bits each which satisfy the matrix equation [0] = [H] * [C]. In this equation, *C* is an *n* x *r* matrix of binary digits, in which each row is a code symbol. The *H* and the *0* matrix are both of size *(n – k)* x *r*.

It should be noted that vector symbols thrive on linearly independent vectors. In [14] it is shown that the probability of independence on *r*-bit random vectors is highly dependent on the size of *r*. It mentions that the likelihood of an *n – k r*-bit random vector being dependent is approximately $2^{-[r-(n-k)]}$. Therefore, the size of a vector symbol should be a lot larger than *n – k*. A lower *r* could lead to linear dependent vectors that generate harder to correct errors. We will see this in the evaluation section when a lower vector symbol size results in a lower success rate.

2.3 Vector Symbol Decoding Steps

There are a number of operations to perform in vector symbol decoding which are described below. If we consider symbols over GF(2^1), we will be using Boolean matrices that contain only 0s and 1s. These fields have the property of closed modulo-2 addition and multiplication. However, this paper mainly uses symbols over GF(2^5). The following describes the steps involved in vector symbol decoding.

1. Multiply the parity check matrix *H* and the received vector *y* to attain the syndrome matrix *S*.
2. If the syndrome matrix *S* contains a non-zero element, then we know that the received vector *y* contains an error.
3. Use column operations on the syndrome matrix *S* to find the null indicators. Using these indicators we can find the null combinations from the row space of the parity check matrix *H*.
4. Take the logical OR of these null combinations to form the *error location vector*.
5. A zero in the *error location vector*, also known as the *elv*, indicates the position of the error(s).
6. Form an *n* x *n* sub-matrix of the parity check matrix *H*, where *n* is the number of zeroes found in the *error location vector*. This matrix corresponds to the *n* rows of *S* and error position columns to be used to solve for the error values.

2.3.1 Example

Let's assume a (7, 3) code with generator polynomial $g(X) = 1 + X^2 + X^3 + X^4$. This vector symbol with eight bits each has already completed the first two steps mentioned above and computed $[S] = [H] * y$ as shown in figure 2.1 below.

$$\begin{bmatrix} 10110101 \\ 01100010 \\ 01100010 \\ 00000000 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix}$$

Figure 2.1 - Vector symbol matrix [9]

We now would like to perform column operations on the syndrome matrix $[S]$ to find the null combinations. By adding column 1 to columns 3, 4, 6 and 8, we will end up with the shown in figure 2.2.

$$\begin{bmatrix} 1\,0\,0\,0\,0\,0\,0\,0 \\ 0\,1\,1\,0\,0\,0\,1\,0 \\ 0\,1\,1\,0\,0\,0\,1\,0 \\ 0\,0\,0\,0\,0\,0\,0\,0 \end{bmatrix}$$

Figure 2.2 - Syndrome matrix with column operations performed

We then want to add column 2 onto columns 3 and 7. The result is shown in figure 2.3.

$$\begin{bmatrix} 1\,0\,0\,0\,0\,0\,0\,0 \\ 0\,1\,0\,0\,0\,0\,0\,0 \\ 0\,1\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,0 \end{bmatrix}$$

Figure 2.3 – Final syndrome matrix after column operations

We can now figure out the null indicators from this final matrix, which are row 2 + row 3 and row number 4. With these indicators, we then find the null combinations from the parity check matrix *H*. By taking row 2 + row 3 and row 4 from *H* in figure 2.1, we get the following two vectors:

row 2 + row 3 : 0 1 0 1 1 1 0
row 4 : 0 0 0 1 1 0 1

Taking the logical OR of these two vectors would give the error locating vector as *0 1 0 1 1 1 1*. Looking at the location of the 0s in the *elv*, we know that the errors are located in positions 1 and 3.

Columns 1 and 3 of *H* tell us that the error values are in rows 1 and 2 of *S*.

$$\begin{bmatrix} 1\,0 \\ 0\,1 \end{bmatrix} \begin{bmatrix} e_1 \\ e_3 \end{bmatrix} = \begin{bmatrix} 1\,0\,1\,1\,0\,1\,0\,1 \\ 0\,1\,1\,0\,0\,0\,1\,0 \end{bmatrix}$$

Figure 2.4 – Error values

The error values are as follows then:

$$\mathbf{e_1 = 1\,0\,1\,1\,0\,1\,0\,1}$$
$$\mathbf{e_3 = 0\,1\,1\,0\,0\,0\,1\,0}$$

Chapter 3

Reed-Solomon Decoding

Reed-Solomon codes were discovered in 1960 by Irving S. Reed and Gustave Solomon. This class of error-correcting codes is very popular today. They are used in a variety of applications including optical media, broadband data transmission and television broadcasting systems. These non-binary cyclic codes are made up of *m*-bit sequences, which contain *k* data symbols to be encoded and *n* code symbols in the encoded sequence of bits. A symbol error occurs when at least one bit in the symbol is incorrect. The code will correct the symbol whether if it is one bit that is incorrect or all the bits received are error bits. No matter how many bits in a symbol are corrupted, it is only regarded as one single error. Due to this property, this gives Reed Solomon codes a very good burst correcting capability.

The code is known to have a *t* symbol-error correcting capability where $t = (n - k)/2$. This means that it can correct up to *t* errors in the received sequence and 2*t* erasures, which are errors with the characteristic that their location is known. Reed-Solomon codes also have the property of having the largest possible minimum distance at $n - k + 1$. In other words, it has the greatest number of bits in which two code words differ. As shown in figure 3.1 below, the total size of *k* bits worth of data is a codeword of *n* length with *2t* parity bits. Of course, the more parity bits there are the more CPU power it takes to encode and

decode the codeword. As the error correcting power of the code increases, the implementation grows more complex.

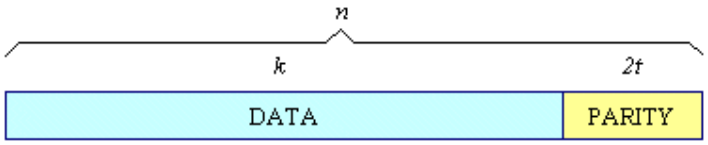

Figure 3.1 - Reed-Solomon codeword [1]

3.1 Example

A common Reed-Solomon code is (255, 223) with 8-bit symbols. This means that each codeword contains 255 bytes in length. 223 out of the 255 bytes make up the data and the 32 remaining bytes are used as parity check codes. In this example, *t* would equal 16, since *255-223 = 32* and *32/2 = 16*. That means that this code can correct up to 16 bytes of errors anywhere in the codeword and 32 bytes of erasures. In addition, it has a minimum distance of *255 - 223 + 1 = 33*.

3.2 Encoding

Reed Solomon encoding is done by first figuring out the generator polynomial *g(X)*. The degree of this polynomial depends on the number of parity check symbols in the code. In the *t* = 16 example above, the X^{32} polynomial degree would be the highest. For

simplicity, let's assume a (7, 3) double-error correcting Reed Solomon code. We notice that this code has $n - k = 4$ roots. Therefore the generator polynomial *g(X) is*,

$$\mathbf{g(X) = (X - \alpha)\ (X - \alpha^2)\ (X - \alpha^3)\ (X - \alpha^4)}$$

Multiplying this out and canceling accordingly would give us,

$$\mathbf{g(X) = \alpha^3 + \alpha^1 X + \alpha^0 X^2 + \alpha^3 X^3 + X^4}$$

We can check the correctness of the generator polynomial by setting $X = \alpha, \alpha^2, \alpha^3, \alpha^5$. Other alphas need not be checked since they are conjugates of the previous four. If

$$\mathbf{g(\alpha) = 0,\ g(\alpha^2) = 0,\ g(\alpha^3) = 0,\ g(\alpha^5) = 0}$$

Then we know that our g(X) is correct. Otherwise, we would have to re-compute it.

3.2.1 Steps to Encoding

1. Convert the data symbols into their respective alpha polynomial message *m(X)*.

   (eg. 110 011 101 would equal $\alpha^3 + \alpha^4 X + \alpha^6 X^2$)

2. Multiply *m(X)* by $X^{(n-k)}$ to obtain the product *b(X)*.

   (eg. $\alpha^3 + \alpha^4 X + \alpha^6 X^2 * X^4 = \alpha^3 X^4 + \alpha^4 X^5 + \alpha^6 X^6$)

3. Divide this new polynomial by *g(X)* to obtain *p(X)*.

   (eg. $[\alpha^3 X^4 + \alpha^4 X^5 + \alpha^6 X^6] / [\alpha^3 + \alpha^1 X + \alpha^0 X^2 + \alpha^3 X^3 + X^4]$)

4. Obtain the encoded message *u(X)* by adding *p(X)* to *b(X)*.

3.2.2 Encoding Using a FSR

Encoding can also be done using a feedback shift register with field element multipliers right before the XOR adders. The bits that result in the registers are the parity bits, *p(X)* which will be added to the message. Figure 3.2 shows the architecture of the feedback shift register that would be used for the example above. With Switch 1 closed, the message symbol sequence is fed in for the first *k* cycles with all the registers initialized to zero. At the same time, Switch 2 is in the down position to allow for the input to also be fed to the output. The input is then multiplied by the respective field element alphas. The result is either fed into a register (for the first register on the left) or added with the previous register's contents using XOR gates. This is done until the whole input sequence has been fed in (after *k* cycles). Switch 1 is then open and Switch 2 is switched to the up position to allow for the parity bits to be concatenated to the message.

The decoding method in this paper uses a similar technique to compute $\text{Rem}\{X^{(n-k)^*}m(X)\}$, where m(X) is the input. The registers in the technique denote the syndromes instead and trap the errors when at least one zero vector in a register is found. This will be shown later on in the paper.

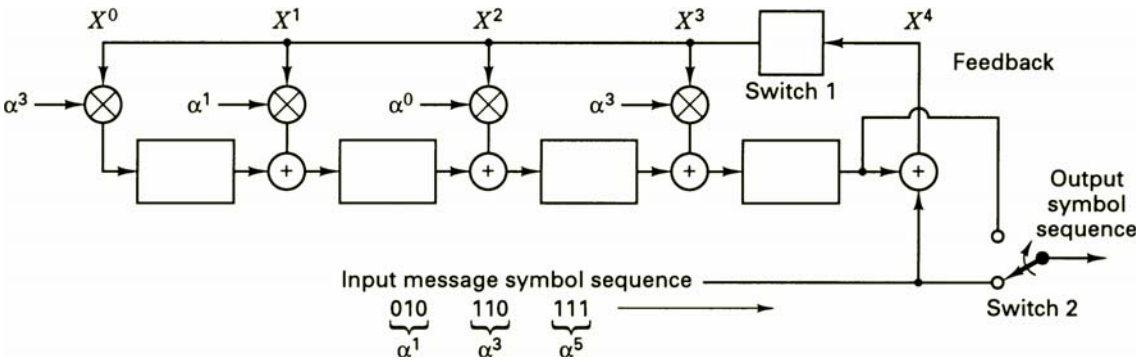

Figure 3.2 – example of a feedback shift register encoding a Reed Solomon codeword [16]

3.3 Decoding

Reed-Solomon decoders attempt to locate the position and magnitude of up to *t* errors or 2*t* erasures and correct them. Decoding a Reed-Solomon code requires that we compute the syndrome of the received vector *r(X)*. This tells us whether or not *r(X)* is a member of the codeword set. A zero syndrome will indicate that it is a part of the set. On the other hand, a nonzero value will signify the presence of errors in the received vector. Therefore, our *r(X)* will be the addition of *u(X)*, which is the transmitted vector, and *e(X)*, which is the error vector.

3.3.1 Steps to Decoding

1. Take $r(X) = u(X) + e(X)$ and convert the bits into their respective alphas as in step 1 of the encoding procedure.

2. Substitute the *2t* roots of the generator polynomial into the *X*. For the double-error correcting code we used in the encoding example, we would substitute the field elements α, α^2, α^3, α^4 into *r(X)* for the four syndrome equations. (Note that if the error polynomial *e(X)* is known, then by evaluating it at the roots of *g(X)*, we should also see the same values when substituting the *2t* roots into *r(X)*).
3. Determine if any of these equations result in a nonzero value. A nonzero value indicates an error in the received vector. Otherwise, if all syndromes are zero, we can assume that the received sequence is not corrupted.
4. If we find a nonzero value, we need to figure out the error location polynomial. We first note that there are *v* errors in locations $X^{j1}, X^{j2}, X^{j3}, \ldots, X^{jv}$ where *v* is unknown and *j* denotes the location of the error. The error values are symbolized by $e_{j1}, e_{j2}, \ldots, e_{jv}$. Therefore,

$$\mathbf{e(X) = e_{j1}X^{j1} + e_{j2}X^{j2} + e_{j3}X^{j3} + \ldots + e_{jv}X^{jv}}$$

We now define $\beta_l = \alpha^{jl}$ where $l = 1, 2, \ldots, v$. These are known as the error location numbers. We can then express the *2t* set of equations as follows:

$$\mathbf{S_1 = r(\alpha) = e_{j1}\beta_l + e_{j2}\beta_2 + \ldots + e_{jv}\beta_v}$$

$$\mathbf{S_2 = r(\alpha^2) = e_{j1}\beta_l^2 + e_{j2}\beta_2^2 + \ldots + e_{jv}\beta_v^2}$$

$$\mathbf{\ldots}$$

$$\mathbf{S_{2t} = r(\alpha^{2t}) = e_{j1}\beta_l^{2t} + e_{j2}\beta_2^{2t} + \ldots + e_{jv}\beta_v^{2t}}$$

5. The error location polynomial is defined as $\sigma(X) = 1 + \sigma_1 X + \sigma_2 X^2 + ... + \sigma_v X^v$. The roots of this equation are the inverses of the error-location numbers, defined as $\beta_1, \beta_2, \beta_3, ..., \beta_v$ and the σ are the coefficients of the error-locating polynomial. In order to find these numbers, we would want to find the solution that gives us the smallest degree, producing $\sigma(X)$ with the minimum number of errors. One method that can be used to solve for the equations and determine the error-location polynomial is Berlekamp's iterative algorithm [2]. However, there are other ways to go about finding it, such as Euclid's algorithm [8] which tends to be more widely used in practice because it is easier to implement. These algorithms are not described in detail due to their complexity and slight relevancy to this literature.
6. Once we figure out the error-locating polynomial, we can go about determining the error values. By substituting $1, \alpha, \alpha^2, \alpha^3, ..., \alpha^q$, where $q = 2^m$ from $GF(2^m)$, into the $\sigma(X)$ we found above, we can find the inverse of the roots. These values indicate the error-location numbers that we need to generate the error pattern polynomial $e(X)$. A popular algorithm used to find these values is the Chien search [8]. If we are dealing with binary symbols, then the error pattern polynomial denotes the errors since we are only concerned with 0s or 1s and we can flip them accordingly. However, in non-binary cases, we would need to find the error values by the following steps.

7. To determine the error values, we first designate $\beta_1, \beta_2, \beta_3, \ldots, \beta_v$ to equal the alphas that we found in $e(X)$, where v signifies the number of errors that we have found. We can then use the syndrome equations that we found in step four to solve for the error values,

$$\mathbf{S_1 = r(\alpha) = e_{j1}\beta_l + e_{j2}\beta_2 + \ldots + e_{jv}\beta_v}$$

$$\mathbf{S_2 = r(\alpha^2) = e_{j1}\beta_l^2 + e_{j2}\beta_2^2 + \ldots + e_{jv}\beta_v^2}$$

$$\mathbf{\ldots}$$

$$\mathbf{S_{2t} = r(\alpha^{2t}) = e_{j1}\beta_l^{2t} + e_{j2}\beta_2^{2t} + \ldots + e_{jv}\beta_v^{2t}}$$

8. Since now we know both S and β, we can go about solving the equations for e. Depending on the v number of errors we found in the error patter polynomial, we can figure out the error values using any v number of syndrome equations listed above. One way to do this is by writing out the matrices. For example, for the (7, 3) case above, if we chose to use S_1 and S_2, we can form the following matrix,

$$\begin{bmatrix} \beta_1 & \beta_2 \\ \beta_1^2 & \beta_2^2 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \end{bmatrix}$$

9. By inserting the respective values into that matrix, we can figure out e_1 and e_2 since they are the only unknowns. To solve for the e's, we can invert the above matrix to bring the β's to the right side of the equation. Once we have computed the values for

the *e*'s, we can correct the received polynomial by adding it to the received polynomial *r(X)* to obtain the codeword *u(X)*.

Chapter 4

Burst Error Decoding

Bursts of errors arise when a continuous set of symbols are received in error. This happens when disturbances or interference in the communication channels occur. As mentioned in the previous section, Reed-Solomon codes allow for very good burst correcting capabilities. The Reiger bound given for a Reed-Solomon code is $(n - k) / 2$, meaning it can correct up to that many errors only. There have been techniques that were developed to increase this bound. This section starts by introducing a few definitions of bursts followed by some information on prior work that has been done in this area.

4.1 Physical Burst

A burst in a block code is defined as a vector of consecutive errors. The length of the burst, l, begins from the first error position i, such that $1 \leq i \leq n$, and ends in the last error position $i + l - 1 \leq n$ [10]. However, the burst of errors need not necessarily be all errors as shown in figure 4.1. For example, if position $i + 2$ is correct, the length of the burst is still l. The case in which all the symbols in the burst are errors is known as a *full error burst* [10].

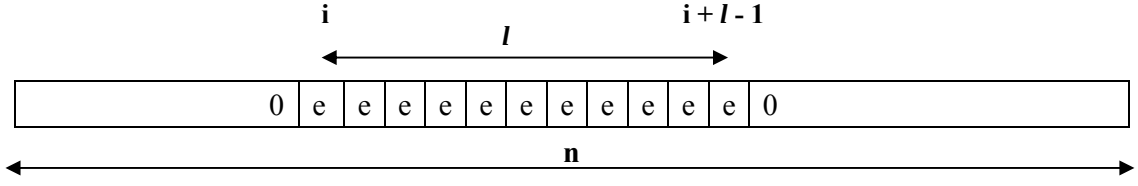

Figure 4.1 – Physical Burst

4.2 Cyclic Burst

A cyclic burst can be characterized as having the *n* positions of the code arranged around a circle as shown in figure 4.2. The burst length, *l*, is defined as the number of positions that cover the burst in the arc. Cyclic bursts that start at the end of the codeword and wrap around to the beginning of the codeword, however, are not considered the physical bursts described above. An error pattern such as *110000010* would be considered a cyclic burst of length 4 but a physical burst of length 8. Note that a *full error cyclic burst* is the same as a *full error burst*.

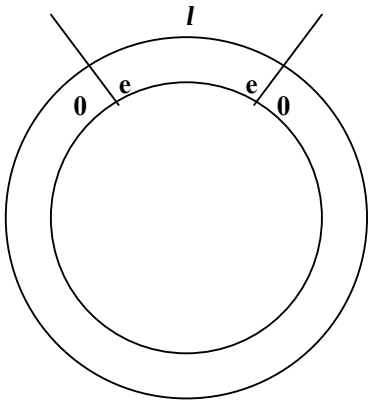

Figure 4.2 - Cyclic Burst

4.2.1 Error Trapping

If the burst of errors in a received vector are confined to the $n - k$ high-order positions, X^k, X^{k+1}, …, X^{n-1} and are cyclically shifted into a linear feed-back shift register $n - k$ times, the resulting syndrome will display the error pattern. Since the syndrome $s^{(n-k)}(X)$ of the

received vector $r^{(n-k)}(X)$ is equal to $e^{(n-k)}$ mod g(X), then $s^{(n-k)}(X) = e^{(n-k)}(X)$ which is the error pattern. This is computed when the r(X) is shifted *n - k* times since the circuit is computing $X^{(n-k)}$*r(X) mod g(X), in which the errors are trapped as the syndrome values. Due to shifting acting as multiplication, *n – k* shifts will bring the errors into the registers.

If the burst of errors are not in the *n – k* high-order parity digit positions, but are located in *l* consecutive positions, then after *i* cyclic shifts, the errors will appear as the syndrome s(X). We will see the first *l* high-order digits of $s^{(i)}(X)$ as the errors and the *n – k – l* low-order digits of $s^{(i)}(X)$ being zeros.

4.3 Reiger Bound

A code that is capable of correcting all errors in a burst of up to length *l*, but not all errors in a burst of length *l + 1*, is known as an *l-burst-error-correcting code* [8]. This means that it can correct a *full error burst* of length *l*. It is proved in [8] that the number of parity-check bits required for an *l-burst-error-correcting code* needs to be at least 2*l*. In other words,

$$\mathbf{n - k \geq 2}\boldsymbol{l}$$

From this equation, we can see the Reiger bound, which says that the burst error correcting capability of an *(n, k)* code is,

$$\boldsymbol{l} \leq \lfloor \mathbf{(n - k) / 2} \rfloor$$

4.4 Feed-Back Shift Register

Gallager's approach uses the error trapping idea described above in which the shortest burst is found with a given syndrome [7], allowing it to detect and correct bursts greater than the Reiger bound. This method, for binary codes, can find bursts that are of length less than $n - k$. It requires two passes to find the errors, the first is taken to locate the shortest burst and the second is used to trap it. Unfortunately, the drawback is that as the length of the burst increases above the Reiger bound, the probability of a burst of shorter length with the same syndrome increases as well, causing the decoding to fail. In cases where the length of the burst is $< n - k$, there could be two bursts of with the same syndrome. If the longer burst ended up being the correct one, then Gallager's method would make an incorrect decision since it was set to choose the shorter of the two. Therefore, for binary codes, since it is impossible for all bursts of length larger than $(n - k)/2$ to have different syndromes, we rarely see binary burst error correcting codes attempt to correct errors greater than the Reiger bound. Non-binary codes, on the other hand, are a different story as they allow for a better correction rate.

In [13], the non-binary case of burst errors is examined in which specific classes of vector symbol cyclic codes are shown to be able to correct up to $n - k - 1$ errors, with the restriction that they are linearly independent. However at the maximum number of errors, it only succeeded less than fifty percent of the time. Decoding is also done with a feed-back

shift register like that of the error trapping technique. The modulo-two XOR adders are, instead, replaced with r-bit vector XORs since they are working in a field greater than GF(2). The method shown later will make a similar modification.

4.5 Interleaving

One common method that has been used to increase the Reiger bound is code interleaving. This is done by taking a *(n, k)* code and combining it with *c* code words from the original code with alternating symbols. The *c* code words are arranged into *c* rows of a three dimensional array and transmitted column by column. Effectively, this would increase the error-correcting capability of the code by *c* since a burst of errors will affect no more than one digit in each row. However, this comes at the cost of a longer latency. In a cyclic interleaved code, the syndrome computation can also be done by replacing each register stage with *c* stages in the decoder for the original code.

Fossorier shows in [6] that code interleaving allows for the correction of bursts up to length $c(n - k) - m$, where *c* is the depth of the interleaved code (*cn, ck)*. It is also shown to be successful $\geq 1 - c*n*2^{-m}$ percent of the time. This means that interleaving a code actually increases the efficiency and success of burst error correction.

Interleaving had initially been done so that each component has a lower number of errors, therefore allowing a greater range of correction. However, the technique described

in this paper utilizes the concentration of bursts contained in a vector symbol to allow for a greater error correcting capability.

4.6 Reed-Solomon Decoding

Due to the capability of Reed-Solomon codes for correcting bursts of errors, there have been a number of algorithms designed to take advantage of this fact. One method was discovered by J. Chen and P. Owsley is described in [3], which deals with non-binary cases. It is said to be able to correct bursts of up to $n - k - 2$ as opposed to the $n - k - 1$ correcting capability of the technique described in this paper. However, this method is quite complex as it requires several Galois field operations in fields greater than $GF(2^1)$ and solving for variables. There is also a probability of failure associated with it. As the burst length increases, so does the probability of a "miscorrect." In the technique that will be described later in this paper, the probability of success increases as the number of bits increase.

In response to the algorithm developed in [3], E. Dawson and A. Khodkar had produced a more efficient algorithm in [4] to correct bursts greater than the Reiger bound by reducing the formation of the key equations and the calculation of the roots. Although this algorithm did lower the amount of operations, the method in this thesis has an even lower decoding complexity by using a feed-back shift register.

Chapter 5

Methodology

As described above, Reed-Solomon burst errors can be detected and corrected in several different ways. Here, a simpler method is presented by using feed-back shift registers and the error trapping technique [10]. As stated in [10], it is possible to correct up to $n - k - 1$ burst errors. The input is a vector symbol code for a Reed-Solomon code over the field of $GF(2^m)$. This vector symbol has $r = j * m$ bits each, in which there are j $GF(2^m)$ components. The circuit is fed these symbols of r bits each which contain $e \leq n - k - 1$ r-bit error vectors. We assume that the errors are composed of independent random patterns of 0s and 1s.

Once the complete received vector has been shifted in, the circuit beings to search for zeroes. The e errors values will appear as the syndromes in the other registers when at least one zero is found in a register. There will be $n - k - e$ zero entries and the remaining e non-zero entries will be the exact error vectors. The testing for zeroes can actually begin right after the burst if the codeword is the all zero vector and there are no more nonzero inputs. However, in real systems, the codeword is not always zero, so testing must be done after the whole sequence has been shifted in. It is said in [10] that n-k-1 errors can be decoded with a probability of less than or equal to $1 - n(n-k)2^{-r}$. The goal of this paper is to prove that it does run with that probability.

5.1 Finding g(X) for the (31, 24) Reed-Solomon code

When the vector symbols are fed in, they are multiplied by the respective field element corresponding to the generator polynomial. The *r*-bit vector that is input is split into *m*-bit components to allow for the multiplication of the field elements from $GF(2^m)$. In the simulations that were done, elements from $GF(2^5)$ are used for the (31, 24) Reed-Solomon code. Therefore, each symbol is divided into 5 bit components, multiplied by the corresponding alpha from $GF(2^5)$ and combined back to form the original r-bit vector. In order to figure out the alphas at each stage, we need to find the g(X) for the Reed-Solomon code that we are using. The generator polynomial was found by multiplying out the *31 – 24 = 7* roots shown below. Tables for the Galois Fields of 2^m from m = 3 to m = 10, can be found in [8].

$$\mathbf{g(X) = (X - \alpha) (X - \alpha^2) (X - \alpha^3) (X - \alpha^4) (X - \alpha^5) (X - \alpha^6) (X - \alpha^7)}$$

$$\mathbf{= X^7 + (\alpha^7 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha)\, X^6 + (\alpha^{13} + \alpha^{12} + \alpha^9 + \alpha^8 + \alpha^7 + \alpha^4 + \alpha^3)\, X^5}$$

$$\mathbf{+ (\alpha^{18} + \alpha^{17} + \alpha^{15} + \alpha^{12} + \alpha^9 + \alpha^7 + \alpha^6)\, X^4 + (\alpha^{22} + \alpha^{21} + \alpha^{19} + \alpha^{16} + \alpha^{13} + \alpha^{11} + \alpha^{10})\, X^3}$$

$$\mathbf{+ (\alpha^{25} + \alpha^{24} + \alpha^{21} + \alpha^{20} + \alpha^{19} + \alpha^{16} + \alpha^{15})\, X^2 + (\alpha^{27} + \alpha^{26} + \alpha^{25} + \alpha^{24} + \alpha^{23} + \alpha^{22} + \alpha^{21})\, X}$$

$$\mathbf{+ \alpha^{28}}$$

$$\mathbf{g(X) = X^7 + \alpha^5 X^6 + \alpha^{29} X^5 + \alpha^5 X^4 + \alpha^9 X^3 + \alpha^{10} X^2 + \alpha^{25} X + \alpha^{28}}$$

The corresponding alpha's bit representations are listed below.

$\boldsymbol{\alpha^{28}}$	**01101**
$\boldsymbol{\alpha^{25}}$	**10011**
$\boldsymbol{\alpha^{10}}$	**10001**
$\boldsymbol{\alpha^{9}}$	**01011**
$\boldsymbol{\alpha^{5}}$	**10100**
$\boldsymbol{\alpha^{29}}$	**10010**
$\boldsymbol{\alpha^{5}}$	**10100**

Chapter 6

Simulator

The simulator is written in C++ mainly using Microsoft Visual Studio. It uses the Galois Field library [15] and a variety of standard library files including the bitset API. Simulations were run on a desktop machine with a 2.0 GHz Sempron processor and 1.5 GB of memory. Runs took on the order of 15 minutes to an hour each.

6.1 Architecture

Figure 6.1 shows the basic architecture of the feed-back shift register implemented in the simulator. The *r*-bit registers begin cleared with zeroes in them. As each of the 31 vector symbols is shifted in, the *r* bits of the symbol are split into *j* 5-bit components from GF(2^5). The 31 is due to *n* symbols from the (31, 24) Reed-Solomon code. These error symbols are stored in a vector of bitsets for the testing phase.

The multipliers shown in the figure are 5-bit field multipliers. The input was originally split for this reason, so that the *j* sequence of 5 bits can be multiplied by the bit representation of the alphas. The *j* products are then combined together to form the original size *r*-bit vector since the additions are computed as *r*-bit vector XORs for each shift.

Once the whole sequence of vector symbols are shifted in, testing begins. The simulator inputs a sequence of 31 zero vectors since exclusive ors of zeros do nothing to the

state of the vector. After each shift, if a zero vector is found in any of the registers, the simulator tests to see if the remaining registers contain the errors that were input to begin with. A false positive would be one in which a zero is found but the errors trapped do not match the errors that were input. Success only comes when anytime a zero is found during the testing phase, the burst of errors are guaranteed to be trapped in the rest of the registers. Otherwise, a single false zero in a run would count as a failure.

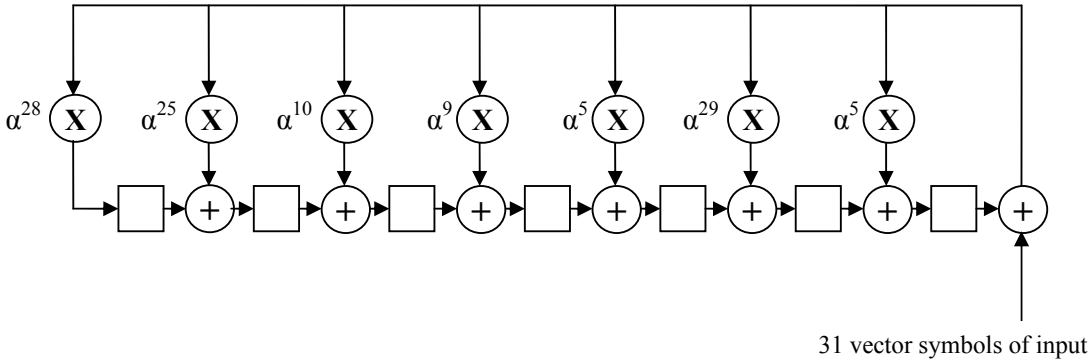

Figure 6.1 Architecture of simulator

6.2 Components

The application consists of several parts: a feeder, a random vector generator, a register controller, xor gates, multiplication gates, a field element handler, a bitset controller, a register tester, and a statistics handler. Their respective functions are listed below.

1. Feeder – inputs the vector created from the random vector generator. The location of the errors is set by an array of 31 binary digits, where a 1 signifies an error and a 0

signifies a correct symbol.

2. Random vector generator – creates a bitset of size NUMBITS and fills it with 1s or 0s based on PROB. The generation of an all zero vector is excluded since that would result in no errors at all. Therefore, the generator only creates a nonzero bitset. Tossing away zero vectors increase the actual probability of 1s. However, for generality, a set number of probabilities are used, namely 12.5%, 25%, 32.5% and 50%.
3. Register controller – manages the previous and current states of all the registers. At each shift, the controller keeps track of the data in the previous shift and the current shift. These registers are initialized to a bitset of size NUMBITS using bitset_reset().
4. XOR gates – computes a bit-wise xor of size NUMBITS of two vectors.
5. Multiplication gates – field element multiplication from $GF(2^5)$. 5-bit Galois field multiplications from the split input and the bit representation of the alpha. This is done using the GF.Mult() operator from the Galois field library. Since the operator took in decimals, binary to decimal and decimal to binary functions were required.
6. Field element handler – responsible for the Galois field elements on which the Galois field library operates on. Initialized and maintained the primitive polynomial of $GF(2^5)$, the generator polynomial for the (31, 24) Reed-Solomon code, and the Galois

field symbol datatypes.

7. Bitset controller – this component performed the splitting and combining of the bitset vectors. It also converted bitsets into decimals and vice versa to compute Galois field operations.
8. Register tester – tests for zeroes in the registers for 31 shifts and calculates the success or failure rate of the decoding method. A run is considered successful when for *every* shift where a zero is found, the values in the registers match the error values that were input. If, for any shift of any run, a zero is found with incorrect error values in the registers, the run is deemed a failure. This is regardless of whether or not a correct set of error values was found earlier in the run. The register tester is activated only after the whole input has been fed in.
9. Statistics handler – stores and prints the success and failure rates along with other useful information. This includes the number of times a zero was found in both a successful and failed run.

6.3 Parameters

There are a number of parameters that can be set for the simulator as described below:

1. CAS – used to denote whether or not the test for zeroes should be done during the input or after the input. Due to the simulator using the all zero codeword, it is

possible to start testing right after the nonzero inputs. In real systems however, it is done after although correctly trapped errors do show up if it is checked after the burst during the input for the zero codeword case. The simulations however, were mainly done with this parameter set to 1, which activates the register tester after the whole input has been fed in.

2. NUMCHECKS – used in conjunction with CAS, which tells the simulator how many shifts the application should cycle through with no input for the testing of the zeros. This parameter is set to 31 for all the simulations that were run due to the number of vector symbols that were input.
3. NUMBITS – number of bits for each vector symbol that was to be fed in. This is the *r* variable in our computations. This parameter was set to a multiple of $m = 5$ from 5 to 30.
4. NUMCOMPS – number of 5-bit components for each field multiplication. This is the *j* number of components. It is set according to the number of bits in each vector symbol.
5. PROB – probability of the number of 1s in the error vectors. This was set ranging from 5% to 50%. However, only a certain number of probabilities are recorded in the results listed in this paper. Note that the since the all zero vector is disregarded, the probability of ones is actually higher. This true probability will be mentioned

in the next chapter.

6. NUMRUNS – the number of runs to simulate with the set parameters. Each test included a 31 vector symbol input and an additional 31 shifts for the input. Each of the scenarios was run 100,000 times on the most part. A run is considered 31 shifts for the input and another 31 shifts for the searching of zeros.
7. G_X – defines the generator polynomial for a (31, 24) Reed-Solomon code. This is set to the generator polynomial that was found in the previous section. The bit representation of the alphas was used to store the g(X).

6.4 Test Cases

There were a total three different types of events that were experimented. The 1s in the vector denote the location of an error symbol and the 0s signify a correct symbol in the 31 symbol vector:

1. A burst of errors in any location less than or equal to $n - k - 1$, which is 6 in our case.

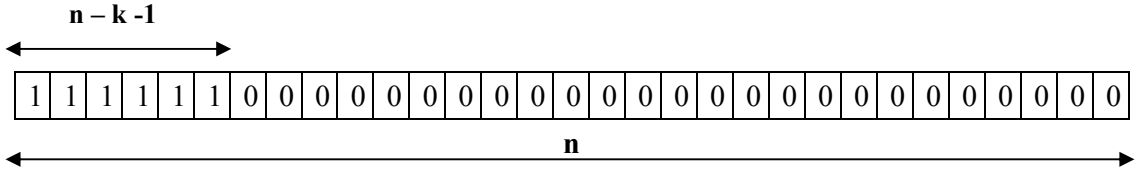

Figure 6.2 – Burst of Errors

2. A burst of length $n - k$ where at least one of the bits is error-free.

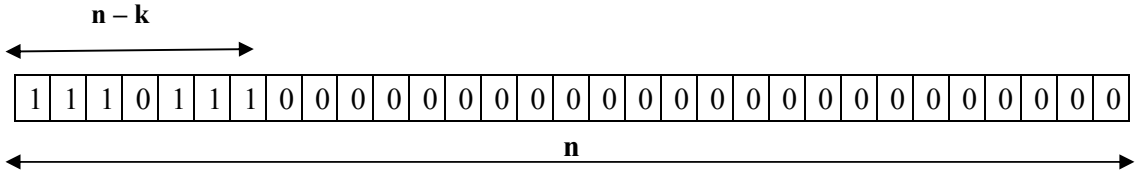

Figure 6.3 – Burst of Errors with One EFC

3. At least 2 errors that occur greater than $n - k - 1$ positions apart.

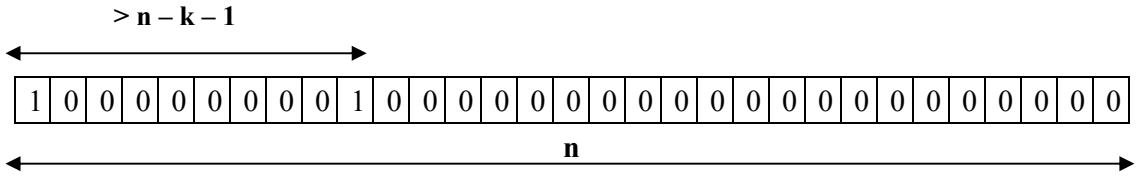

Figure 6.4 – Non-Burst of Errors

Chapter 7

Evaluation

For each of the three different types of tests described in Chapter 6, there were a number of tests run, each with a different *NUMBITS* and *PROB*. This method is said to be able to correct errors at a probability $\leq 1 - n(n-k)2^{-r}$. However, this error correction probability only applies to the case of 50% probability of 1s in the error vector. The results obtained from running the simulations were very close to this suggested probability mentioned in [10], even at differing percentages of 1s in the vectors. The tables in this section show the success rates and failure rates that were taken from the simulations.

Each of the tests listed was run 100,000 times in $GF(2^5)$ with PROB set to 0.125, 0.25, 0.375, and 0.50 using the all zero codeword from the (31, 24) Reed-Solomon code. The check for zeroes in the register were all done after the whole input had been shifted in. It should be noted that the probabilities that are used are approximates of the actual likelihood of 1s in the error vector since complete zero vectors are omitted. Due to using the zero codeword, the generation of a zero would signify no errors. Therefore, the actual probability of 1s in the vector should be $p / [1 - (1 - p)^r]$, where *p* is the *PROB* parameter and *r* is the *NUMBITS* parameter. For simplicity and consistency, multiples of 1/8 was used instead to denote the probabilities. In the few tests that were made using the actual

probability of 1s, the difference in accuracy did not significantly affect the results.

7.1 Case 1: Burst of 6 Errors

Table 7.1 shows the percentage of success of an event where a full burst of six errors are found in the vector. The results shown below are for errors in first six positions of the input. However, a burst of six errors can be anywhere in the 31 symbols. Due to the cyclic nature of the codes, the location of the burst of errors is independent from the success rate. Therefore, any location will give analogous results.

The probability of success shown and proved in [10] is calculated for each r in the tables below. The findings show that the success rate is very close, if not the same to the suggested probability. Due to the union bound on the failure rate mentioned in [10], it is expected that the actual simulation results would be a little bit higher. Note that as the average percentage of ones in the errors decrease, the success rate also decreases. This is to be expected as well due to there being a higher possibility of a false zero trapping incorrect errors.

Although $r = 5$ gives a 0% success rate, the simulator was able to find a few successful runs. This can probably be because the odds of the random error patterns matching another set of random bits in the register increases as the number of possibilities, or bits, decreases. In other words, there is a higher likelihood for the feed-back shift register to

generate a similar set of syndromes. In addition, I believe the similarity in success rates across the four probabilities of $r = 5$ are due to the fact that there is very little room for the generation of significantly different error vectors in each of the runs.

As mentioned earlier, vector symbol decoding is weaker when it comes in contact with linearly dependent vectors, and with a smaller r, the possibility of linear independence decreases. We notice that since $n - k$ is equal to 7 in our case, the r's that are closer to 7 (eg. 5 and 10) show a low probability of success. On the other hand, those that are significantly higher than $n - k$ display a close to perfect rate. This is one of the main reasons for the decrease in success rate since linear dependence of nonzero vectors could create accidental zeroes in the registers. In most cases, we will see two such solutions with one of them being correct, which can be verified with built-in detection. However, our simulations will count these cases as failures as it does not have this feature.

Tables 7.1 – Case 1 Results: Burst of 6 Errors, n = 31, k = 24

$r = 30$, $1 - n(n-k)2^{-r} = 99.99998\%$

Probability of 1s	Success Rate
1/8	99.836%
1/4	100%
3/8	100%
1/2	100%

Table 7.1a

$r = 25$, $1 - n(n-k)2^{-r} = 99.99935\%$

Probability of 1s	Success Rate
1/8	99.499%
1/4	99.991%
3/8	99.999%
1/2	100%

Table 7.1b

$r = 20$, $1 - n(n-k)2^{-r} = 99.97931\%$

Probability of 1s	Success Rate
1/8	98.428%
1/4	99.922%
3/8	99.984%
1/2	99.988%

Table 7.1c

$r = 15$, $1 - n(n-k)2^{-r} = 99.33777\%$

Probability of 1s	Success Rate
1/8	94.468%
1/4	99.021%
3/8	99.404%
1/2	99.455%

Table 7.1d

$r = 10$, $1 - n(n-k)2^{-r} = 78.80859\%$

Probability of 1s	Success Rate
1/8	72.417%
1/4	81.230%
3/8	83.857%
1/2	84.785%

Table 7.1e

$r = 5$, $1 - n(n-k)2^{-r} = 0\%$

Probability of 1s	Success Rate
1/8	0.311%
1/4	0.322%
3/8	0.319%
1/2	0.357%

Table 7.1f

7.2 Case 2: Burst of Length 7 with one Error Free Component

The second series of runs were done with a burst of seven errors in which at least one of the seven components is error free. The numbers in the tables below represent an error in the 4^{th} position of the input vector. However, running simulations with the error-free component in any position resulted in similar numbers. Once again the location of the burst in the whole input is irrelevant to the success rate. When a zero was found, the error vectors were seen to be wrapped around the beginning and end of the registers due to the cyclic properties of the code. The error free component would appear as a zero in the register when the errors were trapped. Therefore, we would actually see the whole burst of seven errors where their locations in the registers depended on the positioning of the errors and the error free symbol. This is similar to the length 6 burst case in which a successful zero would be found either in the first register or last register.

The tables list results that are very similar to those that were shown above with a burst of 6 errors. They also, in turn, match the $1 - n(n-k)2^{-r}$ success probability. The success

rate of this sequence of tests versus the previous tests seems to give a slightly smaller success rate. I believe this is because due to that extra error-free component in the burst, it has a slightly higher chance of generating a false zero. Once again we see that the $r = 5$ case resulted in a few successful runs. We can draw the same conclusions here as case 1 above.

Tables 7.2 – Case 2 Results: Burst of Length 7 with one EFC, n = 31, k = 24

$r = 30$, $1 - n(n-k)2^{-r} = 99.99998\%$

Probability of 1s	Success Rate
1/8	99.847%
1/4	99.999%
3/8	100%
1/2	100%

Table 7.2a

$r = 25$, $1 - n(n-k)2^{-r} = 99.99935\%$

Probability of 1s	Success Rate
1/8	99.487%
1/4	99.991%
3/8	99.999%
1/2	100%

Table 7.2b

$r = 20$, $1 - n(n-k)2^{-r} = 99.97931\%$

Probability of 1s	Success Rate
1/8	98.430%
1/4	99.946%
3/8	99.983%
1/2	99.980%

Table 7.2c

$r = 15$, $1 - n(n-k)2^{-r} = 99.33777\%$

Probability of 1s	Success Rate
1/8	94.402%
1/4	98.948%
3/8	99.386%
1/2	99.472%

Table 7.2d

$r = 10$, $1 - n(n-k)2^{-r} = 78.80859\%$

Probability of 1s	Success Rate
1/8	71.566%
1/4	81.191%
3/8	83.231%
1/2	83.994%

Table 7.2e

$r = 5$, $1 - n(n-k)2^{-r} = 0\%$

Probability of 1s	Success Rate
1/8	0.310%
1/4	0.312%
3/8	0.314%
1/2	0.301%

Table 7.2f

7.2.1 Case with At Least Two Error Free Components

We had mentioned the case of only one error free symbol in the burst of length 7 above. However, there is something we can do when we see more than one error free component in the error vector. Note that when more than one of the components in a burst of length $n - k$ is error free, we will see the generation of shifts in which only one zero is

found. This would of course be incorrect since the correct set has more than one zero vector symbol. Therefore, a strategy that can be used to find the true burst on failed runs is to pick the shift in which we see the most zeros.

7.3 Case 3: Non-Burst Errors

The method described in this paper is primarily designed for decoding bursts of errors. However, we wanted to test the strength of it by seeing how it would fare against a non-burst set of errors. This next and final test was done to confirm the previous results and make sure that it doesn't give false zeroes in non-burst cases. In this scenario, two errors were placed in positions at least 6 spaces apart. This particular series tests had errors placed in locations 1 and 10 of the input. There is no way the errors can be trapped due to the size of the register, which is a reason why the feed-back shift register method can only correct full bursts of up to $n - k - 1$. The extra register is needed to detect the shift in which the errors are trapped.

Instead of measuring the success rate, we measured the appearances of false zeroes. A failure was noted whenever at least one zero vector was found in a run. In fact, any zero found would be considered a false zero. A zero in this case would correspond to a failure due to the errors not being able to be trapped. Therefore, the tables below measure the percentage of zeroes that were found when decoding this sequence of errors, in which we can

relate to the failure rate. We note that the percentage of zeroes found are very similar to the failure rate of the method, $n(n-k)2^{-r}$. In addition, we also see the failure rate rise significantly when the bit count falls below 15, as in the previous two cases. This is what we hoped for because we do not want a non-burst set of errors confusing the decoder.

In the $r = 5$ case, we see that this technique finds a zero 100% of the time, which is exactly the amount computed from the suggested probability. This is different from the previous results we have found for $r = 5$. I believe this is because anytime a zero is found, it is considered a failure, as opposed to the first two cases in which a zero might accidently lead to the errors that were input. It is also safe to say that the dependence of the vectors also play a role in this test case.

Tables 7.3 – Case 3 Results: Non-Burst Errors, n = 31, k = 24

r = 30

Probability of 1s	Zeros Percentage
1/8	0.242%
1/4	0%
3/8	0%
1/2	0%

Table 7.3a

r = 25

Probability of 1s	Zeros Percentage
1/8	0.802%
1/4	0.014%
3/8	0.001%
1/2	0%

Table 7.3b

r = 20

Probability of 1s	Zeros Percentage
1/8	2.432%
1/4	0.115%
3/8	0.005%
1/2	0.003%

Table 7.3c

r = 15

Probability of 1s	Zeros Percentage
1/8	7.563%
1/4	0.967%
3/8	0.125%
1/2	0.100%

Table 7.3d

r = 10

Probability of 1s	Zeros Percentage
1/8	24.283%
1/4	9.340%
3/8	3.886%
1/2	2.99%

Table 7.3e

r = 5

Probability of 1s	Zeros Percentage
1/8	100%
1/4	100%
3/8	100%
1/2	100%

Table 7.3f

7.4 Graphs

Figure 7.1 and figure 7.2 plot the success percentage against the probability of 1s for the first two test cases. Since not all the cases for different *r*'s succeed with a close to perfect percentage, only cases from *r* = 15 to *r* = 30 are plotted in the two graphs mentioned. The next set of graphs, figure 7.3 and 7.4 plot the cases from *r* = 15 to *r* = 5. We notice that in both circumstances, a greater number of ones lead to a greater percentage of success. It can also be concluded that the success rate drops significantly as the average percentage of 1s decrease from 25% to 12.5%. We can notice a greater reduction degree from 20 bits to 15 bits as well. As mentioned earlier, we can relate the increase in failures at lower bits to the linear dependence of the vectors. With a higher degree of linear dependence, we would see an increase in the rate of false zeros in our runs. Lowering the number of bits we have in each vector would decrease the linear independence of the vectors. Therefore, we see that an *r* higher than the *n* – *k* length will generate a lot better results. However, even at 15 and 10 bits, we have an average of 98% and 80% success rate, respectively. Therefore, we can

say that this method can correct bursts of length $n - k - 1$, close to 100% of the time when the number of bits is ≥ 20. This property is attractive due to the growing size of data packets in today's digital data communication. From the graphs below, we can see that this method does very well as long as there are around at least a quarter of ones in the error vector.

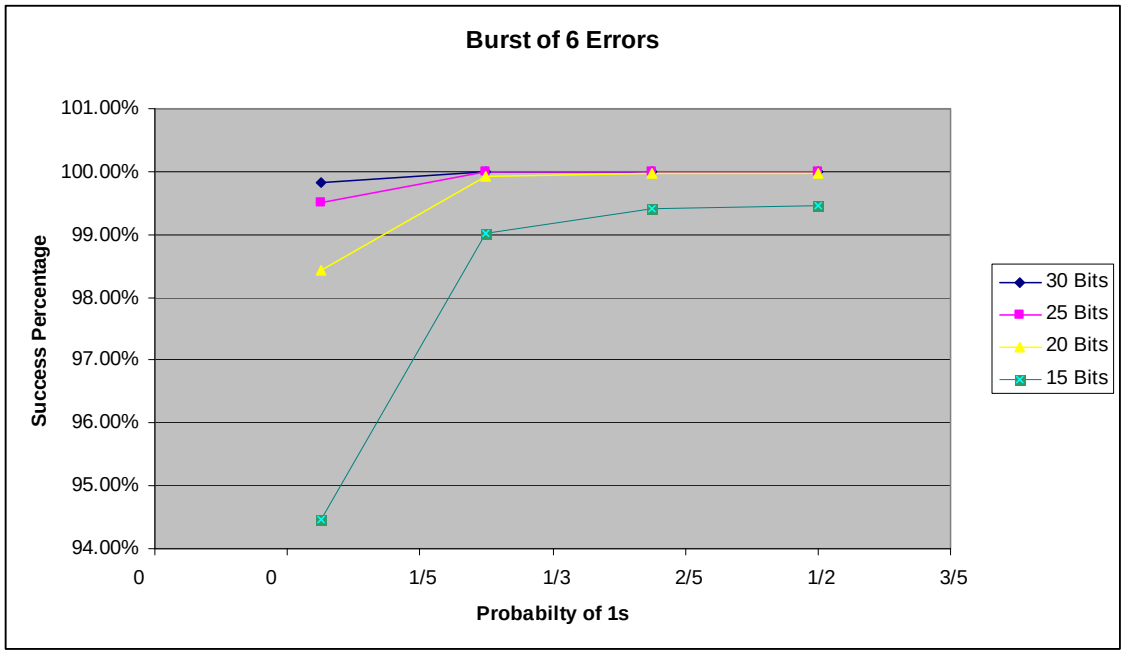

Figure 7.1 – Success Percentage vs. Probability of 1s (30, 25, 20, 15)

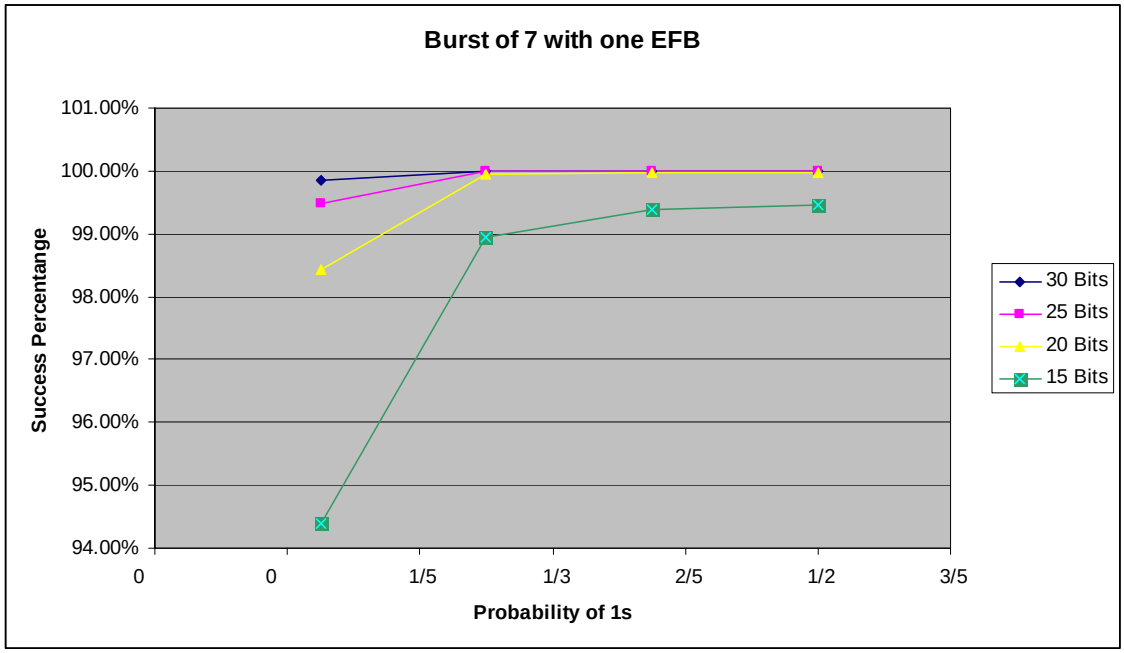

Figure 7.2 - Success Percentage vs. Probability of 1s (30, 25, 20, 15)

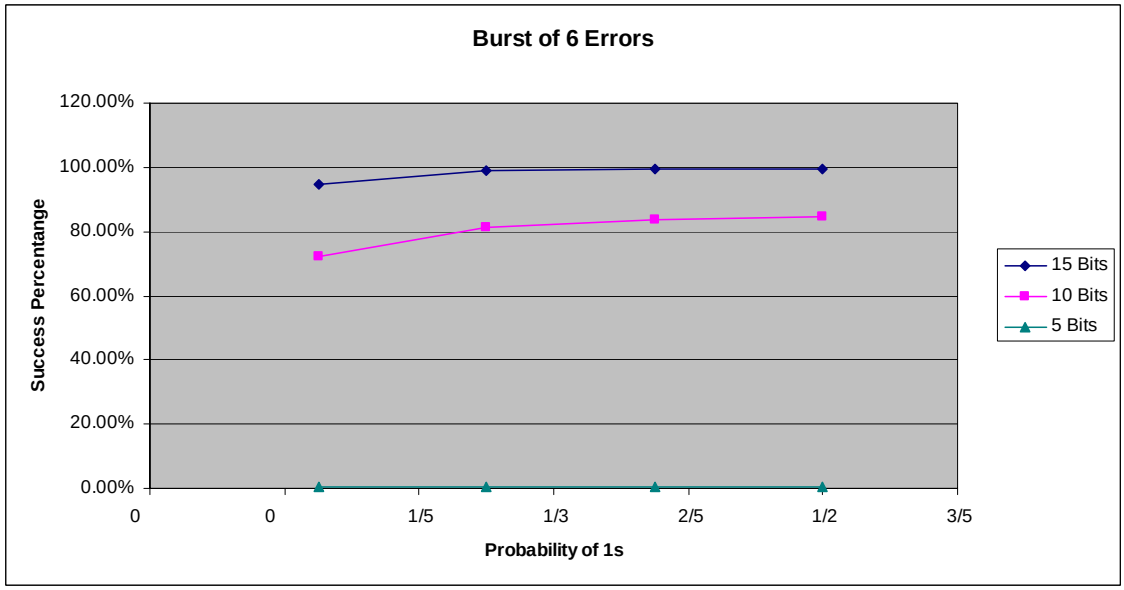

Figure 7.3 - Success Percentage vs. Probability of 1s (15, 10, 5)

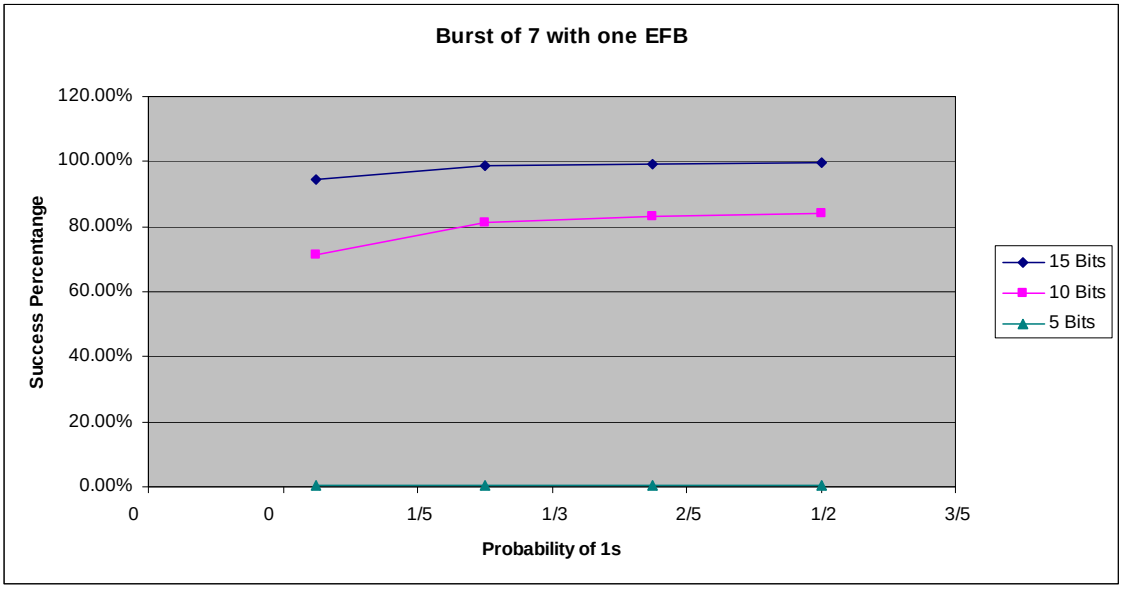

Figure 7.4 - Success Percentage vs. Probability of 1s (15, 10, 5)

Figure 7.5 and 7.6 are graphs displaying the percentage of failure to the probability of 1s, again one with $r = 30$ to $r = 15$ and $r = 15$ to $r = 30$. This is an inverse of the other four graphs which show the success rate instead. We can see that the failure rate approaches zero percent as the number of bits increase. It also has a similar slope at around 25% of 1s in error vector. We can make similar conclusions with this set of graphs since we see a more significant increase in the failure rate from the 25% mark to the 12.5% mark than from 50% to 25% as well. Once again, we can notice a greater change in success rate as we dip below 20 bits. The main difference is the 0% success rate at $r = 5$, in which the reason was explained earlier in this paper.

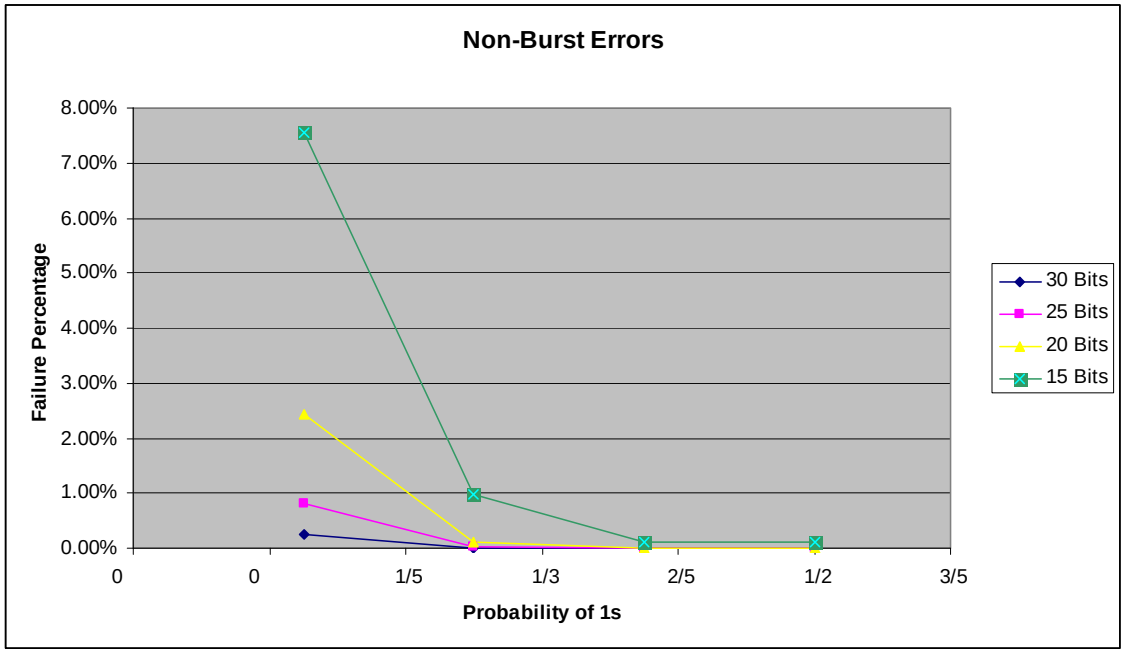

Figure 7.5 – Failure Percentage vs. Probability of 1s (30, 25, 20, 15)

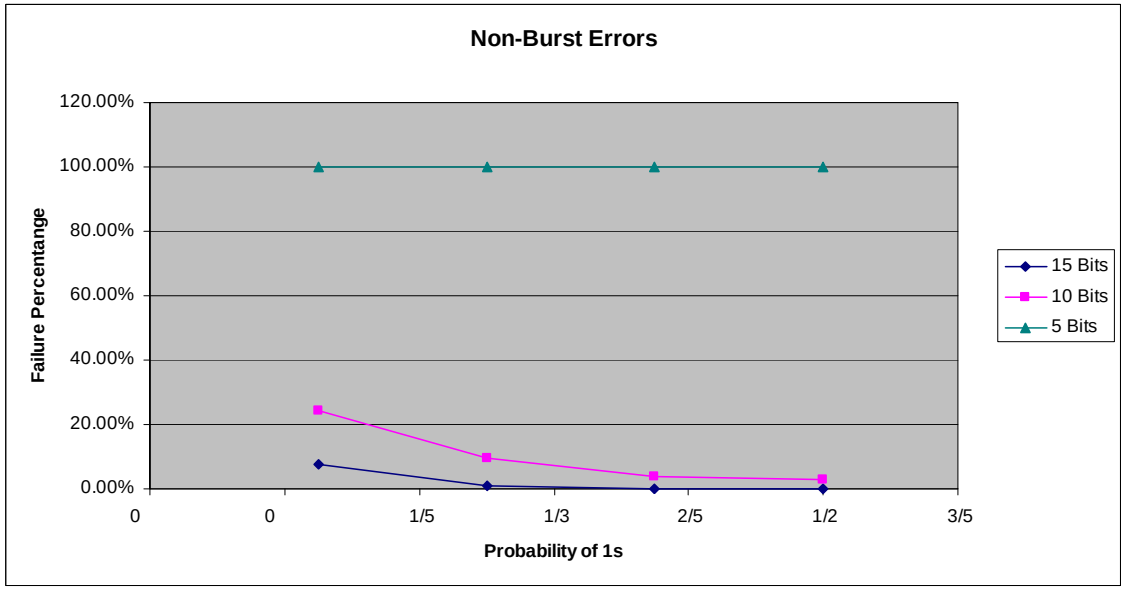

Figure 7.6 – Failure Percentage vs. Probability of 1s (15, 10, 5)

The final figure, figure 7.7 portrays the effectiveness of the number of bits in the vector. It plots the results we got for the 50% probability of 1s cases, along with the suggested probability. As shown in the graph, we can see that all three of the plotted lines are nearly identical to each other. At $r = 15$ and higher, we see a 99% or greater chance of succeeding in decoding a burst of length $n - k - 1$. In the lower r's, we see an increasing rate of failure instead. With our $n - k$ equal to 7, these numbers are to be expected. The more bits we have in the vector symbol, the higher likelihood the vectors are independent and the more likely we are to succeed in decoding and correcting the error. Therefore, we can conclude that there is less room for failure due to a greater number of bits. This is true from r in the denominator of the probability mentioned in [10].

It should be noted that although we found that some of the cases do not generate a flawless success rate, we can still use the vector symbol decoding technique, mentioned earlier in this paper, to find the error location vector and the error values [10]. However, doing that would obviously increase the overhead of decoding.

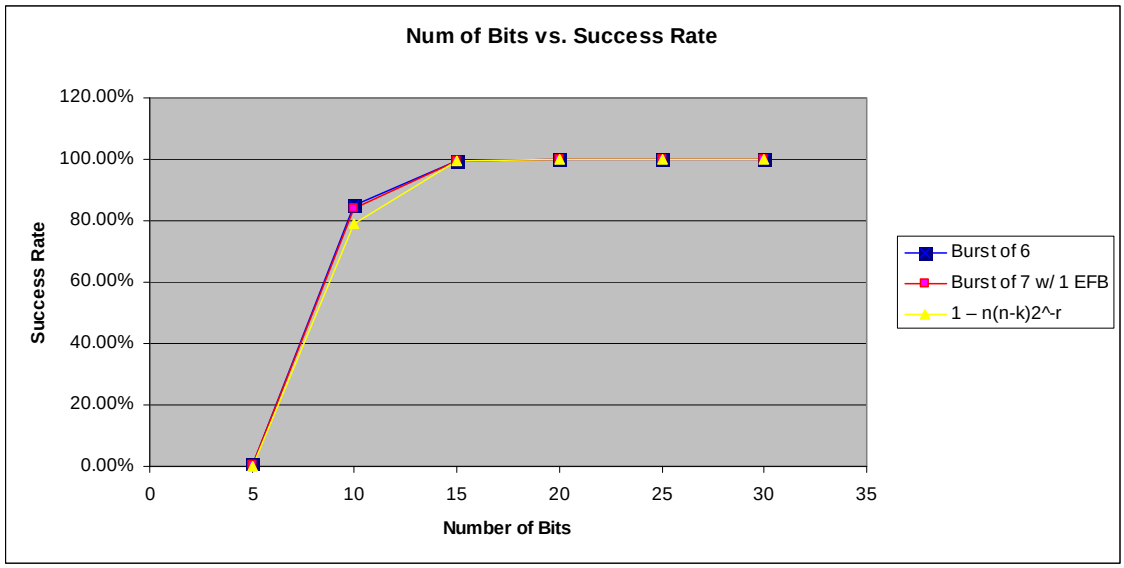

Figure 7.7 – Success Rate vs. Number of Bits

Chapter 8

Future Work

The results from this simulations show that this method is very powerful. Being able to correct bursts close to twice the Reiger bound with a minor rate of failure is very useful. Although this technique does not work as well for vector symbols under 15 bits, anything larger than that succeeds at a close to perfect probability.

Due to the simplicity, I believe a software or hardware application of this circuit should not be too hard and pretty feasible. Since the average size of each data transfer has been increasing as of late, an implementation of this would be a great contribution. In systems that have data sizes ranging from small to large, this scheme can be used in conjunction with another decoder to just handle the larger data sizes. This decoder should be able to fare pretty well with real data.

The case of more than two error-free symbols in a burst of length $n - k$ is also something that can be looked at in the future. As mentioned earlier, when we decode this scenario, we might see shifts in which only one zero is found in the registers. In order to find the true set of errors, we could pick the shift that contains the most zeros. There could be further evaluation done in this area, to possibly increase the success rate of the method described in this paper.

Another technique to consider is to use binary codes, instead of Reed-Solomon codes,

along with vector symbol decoding to correct bursts of errors. It is proved in [10] that a vector symbol code with a minimum distance greater than 2 can correct full bursts of up to length $n - k - 1$. However, this had the restriction that the errors were linearly independent. If the vectors were not completely independent, then we could look for the position that has the greatest number of zeroes to detect the errors. The results from this could then be compared to our findings from the technique used in this paper. Although it is expected that it might be somewhat lower than our results, it would be interesting to see how well it actually performs.

Chapter 9

Conclusion

Error detection and correction plays a major role in the transmission and storage of data. Bursts of errors are cases in which errors gather around a certain part of the code word. This generally happens due to disturbances in a communication channel and could last as long as the interference persists. In addition to the overhead required to transfer data from one point to another, the extra step of guaranteeing the integrity of this data is also a factor in the transfer speed and latency. Therefore, it is vital that we have efficient and reliable decoding methods with the increasing demand for speed and size. By increasing the speed of the decoders, we can increase the speed of a transfer as well.

A simple and fast technique was described in this paper utilizing the power of Reed-Solomon codes along with vector symbol decoding. It makes use of a simple feed-back shift register to correct bursts of errors of up to $n - k - 1$ with a success rate of $1 - n(n-k)2^{-r}$. As compared to other works that use Reed-Solomon codes for burst error correction, decoding complexity in this approach is much smaller and not to mention, the correction range is greater [10].

We found that an approach like this succeeds very often. With at least a 15-bit vector symbol, we are able to uniquely correct 99.5% of the time. Therefore, anything larger than 15 bits will give an even greater rate of success. Although there are higher

margins of error for a lower bit size and lower probability of 1s, these are merely some of the worst case scenarios and could be related to the linear dependence of the vector symbols. In addition, due to the rise in the use of digital data communication, there has been an increase in the size of data transfers. This algorithm could benefit the speed and reliability of these transfers. Despite the small number of failures, vector symbol decoding can still be used to find the error location vector and error values as shown in Chapter 2. The only drawback is that this would require more complex operations.

References

[1] 4i2i Communications Ltd. Reed-Solomon Codes.http://www.4i2i.com/reed_solomon_codes.htm, 2004.

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

[3] J. Chen, P. Owsley. "A burst-error-correcting algorithm for Reed-Solomon codes," *IEEE Transactions on Information Theory*, 1807 - 1812, November 1992.

[4] E. Dawson, A. Khodkar, "Burst-error correcting algorithm for Reed-Solomon codes," *IEE Electronic Letters*, 848 – 849, May 1995.

[5] G. D. Forney, Jr. *Concatenated Codes*, MIT Press, Cambridge 1966.

[6] M. Fossorier. "Universal burst error correction," *IEEE International Symposium on Information Theory*, 1969-1972, Seattle, July 2006.

[7] R.G. Gallager. *Information Theory and Reliable Communication*, 291 – 297, New York, 1968.

[8] S. Lin, D. J. Costello Jr. *Error Control Coding: Fundamentals and Applications*. Second Edition. Prentice-Hall, 2004.

[9] J. J. Metzner. "CSE 554 – Error Correcting Codes," *Pennsylvania State University*, Spring 2007.

[10] J. J. Metzner. “On correcting bursts (and random errors) in vector symbol (n, k) cyclic codes,” *IEEE Transactions on Information Theory*, 1795 - 1807, April 2008.

[11] J. J. Metzner. “Vector symbol decoding with erasures, errors and symbol list decisions,” *IEEE International Symposium on Information Theory*, 34, 2002.

[12] J. J. Metzner. “Vector symbol decoding with list inner symbol decisions,” *IEEE International Symposium on Information Theory*, 481, June 2000.

[13] J. J. Metzner, Y. T. Cha. “On simple correction of bursts up to nearly twice the guaranteed correction bound,” *Proceedings of the 1994 Princeton Conference on Information Systems and Sciences*. 38-47, New Jersey, March 1994.

[14] J. J. Metzner, E. J. Kapturowski. “A general decoding technique applicable to replicated file disagreement location and concatenated code decoding,” *IEEE Transactions on Information Theory*, 911-917, July 1990.

[15] A. Partow. Galois field arithmetic library. http://www.partow.net/projects/galois/. 2006.

[16] B. Sklar. *Digital Communications: Fundamentals and Applications.* Second Edition. Prentice-Hall, 2001.

[17] S. B. Wicker. *Error Control Systems for Digital Communication and Storage.* Prentice-Hall, 1995.