Unequal Loss Protection of Hyperspectral Compressed Images on Reconfigurable Platforms

Todd Owen

A project report submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in Electrical Engineering

University of Washington

2003

Program Authorized to Offer Degree: Electrical Engineering

University of Washington

Abstract

Unequal Loss Protection of Compressed Hyperspectral Images on Reconfigurable Platforms

Todd Owen

Chair of the Supervisory Committee:
Associate Professor, Scott Hauck
Electrical Engineering

Satellite communications are often lossy due to weather and other atmospheric conditions. The lossy channel can pose a problem for data communications, causing data to be lost when conditions are not ideal. In order to protect against data loss, Forward Error Correction (FEC) codes can be added to the data sent. Traditional systems generally assign FEC at a constant rate, as the data are equal in importance. With the use in progressive image coders, variable FEC assignments have become a research interest [3,13].

Compressed hyperspectral images have the unique property that the later data refines the earlier data. Therefore, if only a fraction of the data are received, a valid image still results. This progressive bit stream provides an opportunity to assign FEC based upon the importance of the data. Generally, more FEC should be assigned to the earlier and more important data while less FEC is assigned to the later less important data.

Flexibility of FEC assignment and the ability to generate it in real time presents a good application for Field Programmable Gate Arrays (FPGAs). FPGAs offer high performance, a short design time, and can be reconfigured remotely. These characteristics make them good for satellite systems. Our work is part of a FPGA-based Hyperspectral Image Compression system for use in space.

Table of Contents

Page

List of Figures

Page

List of Tables

1 Introduction

Satellite communication channels are often lossy. Packets sent are lost due to changing atmospheric conditions and other interference. To combat the lossy channel, error correction codes are often added to the sent data. Currently many satellite systems assign forward error correction (FEC) codes at a constant rate. This technique provides a level of error correction that is optimized for one channel bit error rate (BER). More importantly this technique treats data as equally important throughout the transmission.

With the deployment of a Hyperspectral Image Compression system, the data sent from the satellite is no longer equal in importance. Hyperspectral Image Compression is progressive, meaning that all data received refines the data prior to it. If we lose the first half of the bit-stream, but recover the last half, we are still left with no viewable image. For this reason we want to add more FEC to the earlier parts of the bit-stream and lesser amounts of FEC to the later parts.

This work is part of a NASA sponsored investigation into the design and implementation of a space-bound FPGA-based Hyperspectral Image Compression algorithm that is optimized for a lossy channel. In prior work, the Set Partitioning in Hierarchical Trees (SPIHT) compression algorithm was implementation on a FPGA platform [8]. This paper describes a FPGA system that adds unequal amounts of forward error correction to a SPIHT bit-stream transmitted over a lossy communications channel.

2 Background

2.1 FPGAs

Computing systems come in many different forms. The most common type of computing system today uses a general-purpose processor such as an Intel Pentium™ processor. A general-purpose processor contains only a few basic instructions, but these instructions can be sequenced to perform a very wide variety of tasks. This flexibility comes at a price though. General-purpose processors use a lot of power and perform poorly when compared to a custom hardware system.

Custom hardware systems, generally known as Application-Specific Integrated Circuit (ASIC) can perform specific functions quickly and with low power. However, speed and power savings come at a great cost. The development of ASICs requires several time consuming and expensive steps. First, the circuit must be designed and validated. Next, the design must be fabricated. Fabrication involves creating the wafer masks, fabricating the chips, packaging the chips, and then testing the chips. Any modifications or errors in the chip design mean restarting the fabrication process by creating new wafer masks. These design considerations generally make ASIC designs prohibitively expensive for low volume runs or continual design changes.

Field Programmable Gate Arrays (FPGAs) offer an attractive alternative to ASICs or general-purpose computing systems. FPGAs are flexible, fast, and cheap. FPGAs can be reprogrammed without a new fabrication run. This flexibility comes with a price: FPGAs are generally not as fast as ASICs. However, an FPGAs inherent parallelism can give significant performance gains when compared to a general-purpose computing system. Additionally, FPGAs are mass-produced, making them a low cost option.

FPGAs are an array of logic gates that can be programmed to perform a variety of functions. FPGAs consist of programmable logic structures that are distributed throughout the chip. These logic structures are then connected with a programmable

routing interconnect. This programmability allows FPGAs to be used in many different applications.

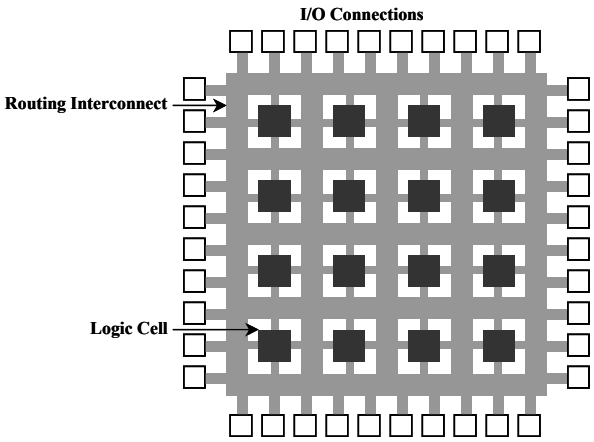

Figure 1: Island style FPGA structure [8].

An FPGA's low cost and reprogamability offer an ideal solution for a satellite system. An ASIC solution offers better performance and power savings, but lacks the ability to be modified after the satellite is launched. A general computing system offers great flexibility, but suffers from performance and power issues. Thus, FPGAs are an ideal choice for satellite systems.

2.2 SPIHT

In prior work, the Set Partitioning in Hierarchical Trees (SPIHT) image compression algorithm was implemented in hardware. SPIHT is a wavelet-based image compression coder [1] that is an extension of Shapiro's Embedded Zerotree Wavelet method [2]. SPIHT works by doing a series of wavelet transforms on an image. These wavelets are then put together in an ordered fashion, allowing the image to be reconstructed by a decoder. SPIHT was selected because it offers characteristics such as:

- Good image quality with a high PSNR.
- Fast coding and decoding.
- A fully progressive bit-stream.
- Can be used for lossless compression.
- Ability to code for exact bit rate or PSNR.

SPIHT begins by doing a series of wavelet transforms on an image. A discrete wavelet transform is used as the image is made up of individual pixels. First, the discrete wavelet transform computes a high and low pass subband in the horizontal direction. Next, this is repeated in the vertical direction. This creates four quadrants, representing the high and low subbands in each direction. After each subband is computed, the results are down-sampled by two. This makes each subband half the size of the original input.

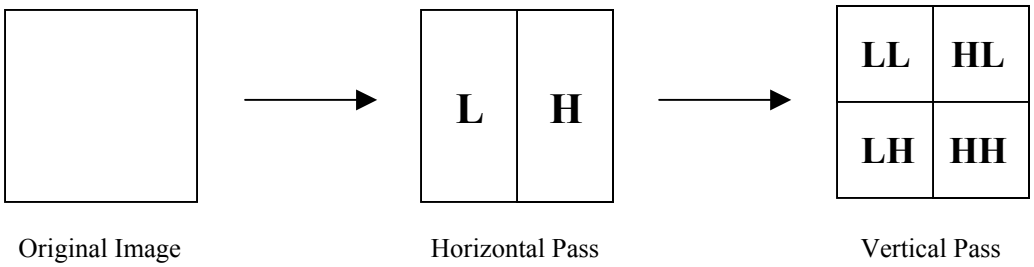

Figure 2: First level wavelet construction.

The four quadrants created by the horizontal and vertical transformations correspond to distinct frequencies. The four quadrants are labeled LL, LH, HL, and HH. LL contains the lowest frequencies, while LH, HL, and HH contain higher frequencies. Once the four quadrants are created, the horizontal and vertical passes will be repeated on the LL quadrant. This process will continue until the procedure is complete.

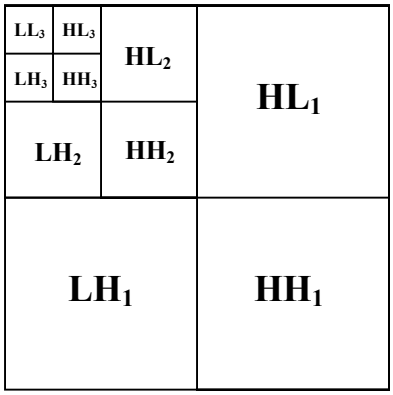

Figure 3: Three level wavelet transform.

Once the wavelet coefficients have been generated they are divided into Spatial Orientation Trees. Each node in a tree corresponds to an individual pixel. The children of each node are the pixels in the same spatial location but at the next finer wavelet scale.

SPIHT transmits information about whether a pixel is significant or not. This is done by using a threshold. If a pixel is above the current threshold it can be processed. Because the pixels are arranged in a tree-like structure, if a pixel is below the current threshold, then all of its children are also below it and therefore insignificant.

Information regarding whether a pixel is above the threshold is held in three lists: the list of insignificant pixels (LIP), the list of insignificant sets (LIS), and the list of significant pixels (LSP). The LIP contains pixels that are being processed, but are not above the threshold. The LIS contains pixels that are being processed, but none of their descendants are above the current threshold and are not being processed. Finally, the LSP contains pixels that are above the threshold and their value is being transmitted.

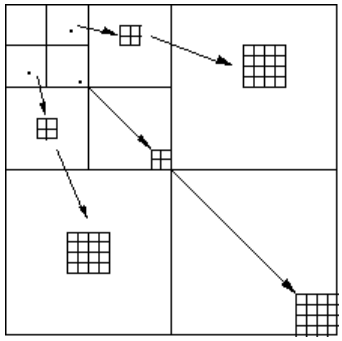

Figure 4: Spatial Orientation Trees [8].

The algorithm works by computing the LIP, LIS, and LSP lists and then adjusting the threshold. By proceeding in this manner the most significant data will always precede less significant data. Additionally, this creates a natural progressive data stream which can be cut off at any point and still give a valid image. For more information about SPIHT see reference [1].

2.3 Finite Fields

In this project we have constructed the hardware for an error detecting and correcting code. Error correction codes work on the principles of Finite Fields, also known as Galois Fields. Finite Fields are the work of French mathematician Evariste Galois. The night of 30 May 1832 he scribbled down the fundamentals of what is now known as Galois theory. Later that night he fought a duel with Perscheux d'Herbinville allegedly over a woman. Galois was fatally wounded in the duel and died the next day [12]. The following overview of Finite Fields is adapted from [14].

In mathematics numbers are often part of a field. Numbers in a field have the property that all arithmetic operations (addition, subtraction, multiplication, division) in the field have a result in the field. The only exception is division by zero. Complex numbers are an example of a field. However, integers are not an example of a field. Multiplicative inverses do not exist with integers.

More strictly, a field is a set of elements such that the following rules are satisfied for any elements x, y, and z in a set.

1) $x + y = y + x$
2) $x + (y + z) = (x + y) + z$
3) $x(y + z) = xy + xz$
4) 0 must exist such that $x + 0 = 0 + x = x$
5) 1 must exist such that $x1 = 1x = x$
6) $–x$ must exist such that $(-x) + x = x + (-x) = 0$
7) x^{-1} must exist such that $x^{-1}x = xx^{-1} = 1$ $(x \neq 0)$

A finite field is a field with a finite number of elements in it. The number of elements in the field is called the order of the field. A field of order q is denoted by GF(q), where GF stands for Galois Field. GF(q) exists only when q can be expressed as $q = p^m$, where p is a prime and m is a positive integer. $GF(p^m)$ exists for every p and m, and there is only one field with p^m elements.

The prime number p is called the characteristic of the field. If a finite field has the characteristic p, then for any element β in the field

$$p\beta = \beta + \beta + \beta + \ldots + \beta = 0 \text{ (p times)}$$

Elements in the field can be expressed in a multitude of different ways. They can be represented by powers of a number or with a polynomial. In each representation, we will use a term α with the following properties:

$$\alpha^{q-1} = 1$$
$$\alpha^i \neq 1,\ 1 \leq i \leq q - 2$$

An element that satisfies these conditions of a finite field is called a primitive element. A root of an equation f(x) = 0 is called a zero of f(x). A primitive element of $GF(p^m)$ is a

zero of a primitive polynomial of order m. Therefore, because α is a primitive element, $f(\alpha) = 0$.

The first way to represent elements is called the power representation. Using this method, elements in GF(q) are expressed as

$$0, 1, \alpha, \alpha^2, \ldots, \alpha^{q-2}$$

We can also write the element 1 as α^0. Therefore all elements of a finite field except 0 can be written as powers of α. Multiplication using the power representation can then be performed by

$$\alpha^i \alpha^j = \alpha^{(i+j) \bmod (q-1)}$$

Because $f(\alpha) = 0$, an element of $GF(p^m)$ can be expressed as a polynomial in α where the coefficients a_i are elements of GF(p):

$$\alpha^j = a_0 + a_1\alpha + a_2\alpha^2 + \ldots + a_{m-1}\alpha^{m-1}$$

For example, using $GF(2^2)$, we use the primitive $f(x) = 1 + x + x^2$. Next we can plug in α into f(x) giving:

$$f(\alpha) = 1 + \alpha + \alpha^2 = 0$$
$$\text{solving gives: } \alpha^2 = 1 + \alpha$$

Now the elements of $GF(2^2)$ can be represented by polynomials giving:

$$0 = 0$$
$$1 = 1$$
$$\alpha = \alpha$$
$$\alpha^2 = 1 + \alpha$$

Using the following polynomial representation we can represent addition as

$$(a_0 + a_1\alpha + a_2\alpha^2 + \ldots + a_{m-1}\alpha^{m-1}) + (b_0 + b_1\alpha + b_2\alpha^2 + \ldots + b_{m-1}\alpha^{m-1})$$
$$= (a_0 + b_0) + (a_1 + b_1)\alpha + (a_2 + b_2)\alpha^2 + \ldots + (a_{m-1} + b_{m-1})\alpha^{m-1}$$

$a_i + b_i$ is performed under modulo-2 arithmetic which is the same as an XOR function between the a_i and b_i. Table 1 and 2 gives the addition and multiplication tables for GF(2) and GF(2^2).

Table 1: Addition and Multiplication Tables for GF(2).

+	0	1
0	0	1
1	1	0

x	0	1
0	0	0
1	0	1

Table 2: Addition and Multiplication Tables for GF(2^2). f(x) = 1 + x + x^3

+	0	1	α	α^2
0	0	1	α	α^2
1	1	0	α^2	α
α	α	α^2	0	1
α^2	α^2	α	1	0

x	0	1	α	α^2
0	0	0	0	0
1	0	1	α	α^2
α	0	α	α^2	1
α^2	0	α^2	1	α

We can also extend the polynomial representation to multiplication. Doing so gives

$$((a_0 + a_1\alpha + a_2\alpha^2 + \ldots + a_{m-1}\alpha^{m-1}) \text{ x } (b_0 + b_1\alpha + b_2\alpha^2 + \ldots + b_{m-1}\alpha^{m-1})) \bmod f(x)$$

Numbers in finite fields can also be represented as a vector of numbers. Using this representation in GF(2^m) the vector will be 1s and 0s. Each 0 or 1 corresponds to the coefficient in the polynomial. Table 3 below shows the elements of GF(2^4) in all three representations.

Table 3: Elements of GF(2^4) where α is a zero of f(x) = 1 + x + x^4

Power Representation	Polynomial Representation	Vector Representation
0	0	0000
1	1	1000
α	α	0100
α^2	α^2	0010
α^3	α^3	0001
α^4	$1 + \alpha$	1100
α^5	$\alpha + \alpha^2$	0110
α^6	$\alpha^2 + \alpha^3$	0011
α^7	$1 + \alpha + \alpha^3$	1101
α^8	$1 + \alpha^2$	1010
α^9	$\alpha + \alpha^3$	0101
α^{10}	$1 + \alpha + \alpha^2$	1110
α^{11}	$\alpha + \alpha^2 + \alpha^3$	0111
α^{12}	$1 + \alpha + \alpha^2 + \alpha^3$	1111
α^{13}	$1 + \alpha^2 + \alpha^3$	1011
α^{14}	$1 + \alpha^3$	1001

The use of the vector representation of numbers in the Galois field is well suited to computers. When in GF(2^m) all the coefficients are either 1 or 0 and are of fixed precision. All operations now have a result that can be represented and precision is not an issue. Also, addition is simple to implement, using only a XOR operation.

2.4 Reed-Solomon Codes

Reed-Solomon codes are the result of work by Irving Reed and Gustave Solomon [18]. These codes have been around since 1960 and are a form of forward error correcting codes (FEC). While they have been around for a long time, only in the last 20 years have efficient hardware implementations existed.

Reed-Solomon codes work by adding extra, redundant information to the original data. If an error occurs during a transmission, then the redundant information can aid in the recovery from the error. The amount of errors that can be fixed is dependent on the amount of redundant information added.

Reed-Solomon codes are used in a variety of applications including:

- Storage devices
- Wireless communications
- Digital television
- Satellite communications
- Broadband modems

Reed-Solomon codes have certain properties that make them highly useful in the real world. RS codes are a systematic linear block code. The block nature of the code means the data are split up into fixed length blocks. Each block is then divided up into m-bit symbols. Symbols are a fixed size, often 8 bits for computing ease. The linear nature of the code means that every possible m-bit symbol is valid. For example, if symbols were 8 bits, we would not need to worry if the data were binary or ASCII. The systematic nature of the code means that the data consists of the original data plus extra redundant data attached to it.

RS codes are usually specified as RS(n,k) with m-bit symbols. A common code found in CD players is the RS(255,233) code with 8-bit symbols. The n refers to how large the block is. A block has a maximum size of 2^m-1. If the block is any smaller than 2^m-1 it is known as a shortened RS code. The k refers to how much real data are in the block. The value n-k is normally referred to as 2t. These are the redundant symbols added to correct errors. Under normal conditions, a RS code can correct up to t errors using 2t symbols. These errors can occur anywhere in the block and still be recovered. Figure 5 below gives shows the makeup of a RS(n,k) code.

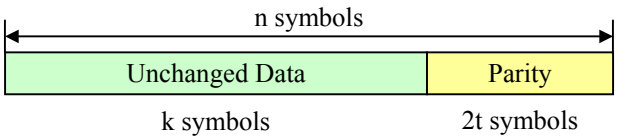

Figure 5: RS(n,k) code.

There exists a special case where we only need t symbols to correct t errors. In order to correct an error, it must be first located and then fixed. In a normal RS block, t symbols are used to locate the error and an additional t symbols are used to then correct the error. If we have a mechanism for locating the error, then we only need t symbols to fix the error [17]. Often a cycle redundancy check (CRC) code will be used for this purpose. For the purposes of explaining RS codes, the 2t version will be used. However, the t version will be used in the ULP framework.

Reed-Solomon codes are especially good at correcting a burst of errors. Because the data are grouped into symbols, we only care if there is an error in a symbol. Regardless of how many errors occur in one symbol, the bit errors get generalized into one symbol error.

RS codes can also be shortened. A shortened RS code is where the block size is something smaller than 2^m-1. Shortened RS codes are created with the same circuits as their full sized counterparts. In reality they are still of length 2^m-1. The difference lies in that the missing symbols are considered to be zeros.

Reed-Solomon codes are generated using a special polynomial. All codewords generated are exactly divisible by the generator polynomial because they were created from it. The generator polynomial, g(x), is created from some variable x and powers of the primitive element α. The generator polynomial has the form:

$$g(x) = (x - \alpha^i)(x - \alpha^{i+1})(x - \alpha^{i+2})\ldots(x - \alpha^{i+2t})$$

Using the generator polynomial, codewords, c(x), are created. By multiplying the next information block, i(x), by the generator polynomial a codeword is made. The information block i(x) will be discussed shortly.

$$c(x) = g(x)i(x)$$

Often the generator polynomial is shown in an expanded form. The following is an example for RS(255,249)

$$g(x) = (x - \alpha^0)(x - \alpha^1)(x - \alpha^2)(x - \alpha^3)(x - \alpha^4)(x - \alpha^5)$$
$$g(x) = x^6 + g_5x^5 + g_4x^4 + g_3x^3 + g_2x^2 + g_1x^1 + g_0$$

From the example, we see that the original terms are expanded and simplified. The g coefficients (g_5, g_4, g_3, g_2, g_1, g_0) are constants made up of additions and multiplications of α^0, α^1, α^2, α^3, α^4, and α^5 and can be computed rather easily.

Once the g coefficients are known, codewords can be created rather easily. The multiplications and additions required to make a codeword can be implemented using a Linear Feedback Shift Register (LFSR) circuit with length 2t. In the LFSR design, each register is m bits wide. Because a LFSR can be used to create codewords, Reed-Solomon codes are cyclic.

Figure 6 below shows the basic architecture of the Reed-Solomon encoder circuit. The multiplier coefficients g0 to g(2t-1) are the coefficients of the RS generator polynomial. The coefficients are fixed for a particular RS(n,k) code. Looking at the cyclic encoder, we can tell that the information blocks i(x) are the result of the input symbols added to the result in the last (b_{2t+1}) register.

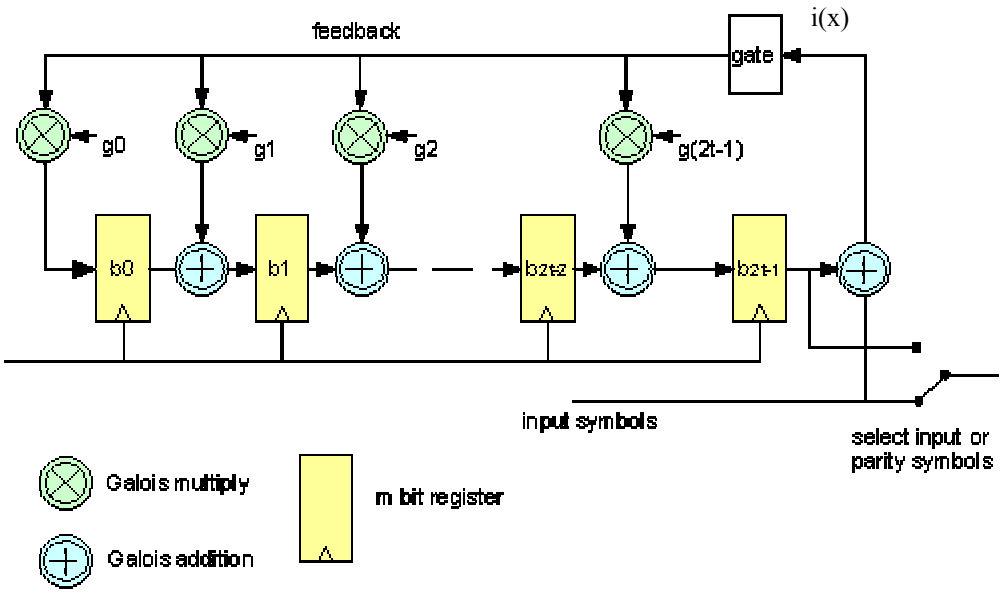

Figure 6: Reed-Solomon Encoder Circuit.

Because RS codes are systematic, a block of data can be read into the circuit and then outputted without alteration. After all k data symbols have been clocked into the circuit, the parity symbol calculation is finished, and the resulting parity codes can be clocked out.

The encoder works by initially setting all registers to zero. Then, on each successive clock cycle, the symbol in each register is added to the product of the feedback symbol and the fixed coefficient for that tap. The result is then stored in the next register. The symbol in the last register is added to the incoming data and then sent to the Galois multipliers as the feedback symbol. Once the circuit has been clocked k times, the incoming data and feedback symbols are set to zero. At this point the symbols held in the registers are the parity symbols. The circuit is then clocked 2t times allowing all parity symbols to be clocked out of the circuit.

3 Prior Work

This work builds upon efforts by Thomas Fry and Alexander Mohr. Fry modified the SPIHT algorithm to work efficiently on FPGAs [8]. Mohr created an Unequal Loss Protection framework for use with progressive bit streams [3].

3.1 SPIHT for FPGAs

In order to make SPIHT efficient on FPGAs, Fry adapted SPIHT to use variable fixed point numbers and Morton Scan ordering. FPGAs have traditionally not employed the use of floating point numbers for some of the following reasons:

- Floating-point numbers require variable shifts based on the exponential description and variable shifters in FPGAs perform poorly.
- Floating-point numbers consume enormous hardware resources on a limited resource FPGA.
- Floating-point numbers are often unnecessary for a known data set.

Using a variable fixed-point number to represent a floating-point number had an effect on the image quality. As we can see from Figure 7, fixed-point numbers with a length of 16 performed nearly identical to the original SPIHT. However, as the fixed variable length was decreased, so was SPIHT's ability to produce a high quality image.

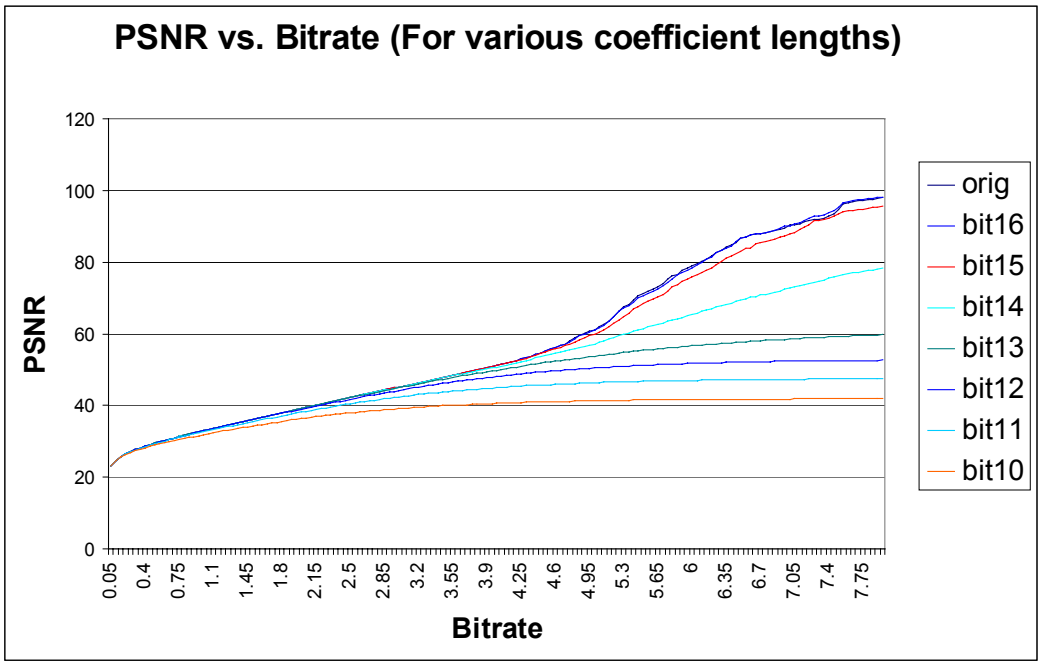

Figure 7: PSNR vs. Bit rate for fixed point SPIHT [8].

Secondly, the SPIHT data was modified so that the ordering was fixed. In the normal SPIHT algorithm every image has a unique list order determined by the image's wavelet coefficient values. These lists are normally referred to as the LIP, LIS, and LSP lists. In order to create a fully parallel version of SPIHT and take advantage of the inherent parallelism of a hardware implementation, the ordering of these lists was fixed. By fixing the order, all lists can be computed in a parallel fashion, greatly increasing performance. The ordering of the Fixed Order SPIHT is based upon the Morton Scan ordering which is discussed in Algazi et al. [6]. As Figure 8 below shows, the fixed ordering slightly decreases image quality for a given bit rate.

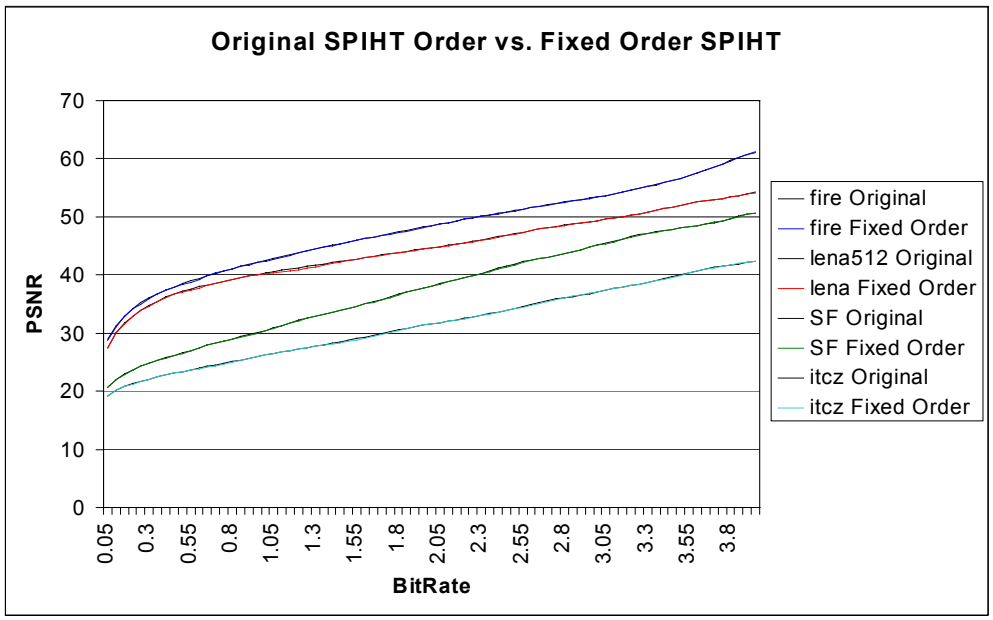

Figure 8: PSNR plot for original SPIHT vs. fixed order SPIHT [8].

3.2 ULPsim

ULPsim (Unequal Loss Protection Simulator) is the result of work done by Mohr, Riskin, and Ladner [8]. ULPsim aims to achieve a graceful degradation of image quality as packet losses increase. The Unequal Loss Protection framework is designed to work on a progressive bit-stream where the earlier data are more important than the later. Therefore, the algorithm seeks to assign more FEC to the earlier data and gracefully decrease the amount of FEC as the bit-stream progresses. This differs from conventional error protection where a constant amount of FEC is assigned during transmission over a lossy network.

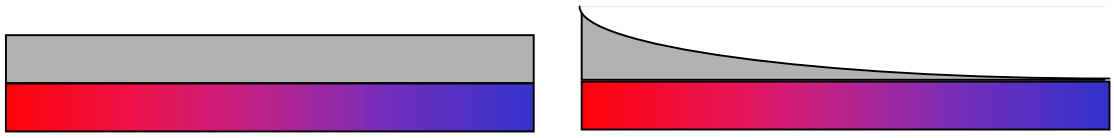

Figure 9: Equal and Unequal Loss Protection Models.

The ULP framework considers the effect of each symbol on image quality when assigning FEC. This is done by splitting up each packet into a number of message

fragments. A message fragments is a symbol in a RS code and therefore they are each m bits. Each message fragment in a packet contains either data or FEC. Because the ULP works on channels with packet erasures, RS block codes are spread out across the number of packets to be sent at one time. Each RS block is called a stream and contains all message fragments needed to construct a message. Figure 10 below shows an example of a ULP packet description. Each row is a stream and each column is a packet. Numbers 1-32 are data, the symbol F is FEC and the symbol C is a CRC.

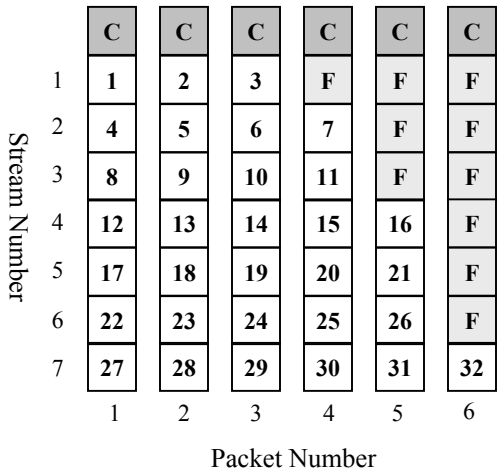

Figure 10: ULP packet description.

Figure 10 is one way to send 32 bytes of data (we are assuming each message fragment is one byte). One can note that the earlier streams have more FEC assigned to them than the later ones. Figure 11 goes on to show the case where one packet is lost.

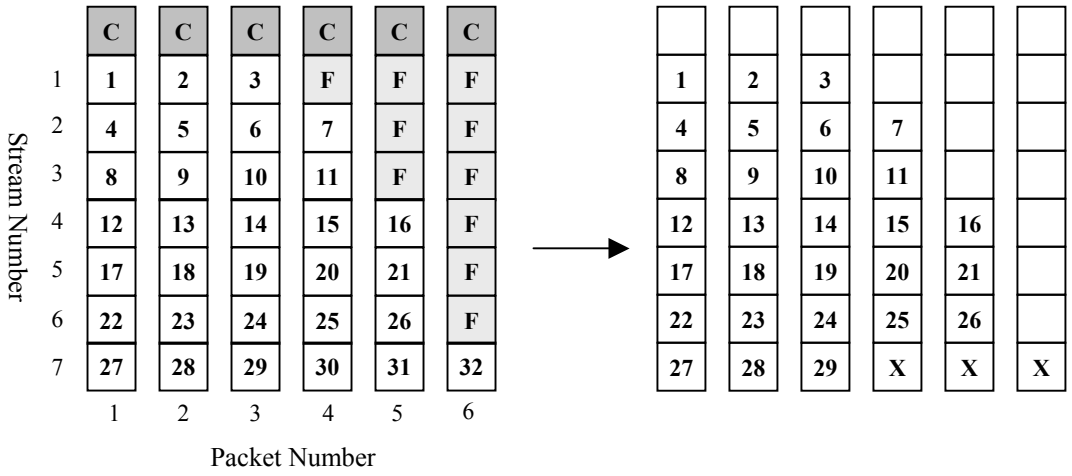

Figure 11: ULP packet loss example.

Figure 11 shows the case where packet number four is lost. In order to recover from the packet loss, each stream must have at least one FEC symbol. Streams 1-6 have one or more FEC symbols added to them; therefore they can recover from the packet loss. Stream 7 contains no FEC symbols. Therefore all message fragments that occur after the lost packet in the stream are lost. This results in losing fragments 30, 31, and 32.

3.2.1 Formal ULP Framework

The ULP framework can be represented more formally [8]. The data to be sent are broken up into messages. Each message M can be broken into N smaller fragments. N is the block size of the RS code and the length of the stream. We now define two more variables, m_i and f_i. The term m_i equals the number of data symbols assigned to stream *i.* Next, we set $f_i = \text{N-}m_i$, where f_i equals the number of FEC symbols assigned to stream *i*. Following the notation in [8], we can define the redundancy assignment, a L-dimensional FEC vector as

$$\overline{f} = (f_1, f_2, \ldots, f_L) \text{ [8]}$$

Entries in the redundancy assignment vector $\overline{f}$ contain the amount of FEC assigned to each stream. For an assignment $\overline{f}$, each message M is divided into a number of fragments $M_i(\overline{f})$. $M_i(\overline{f})$ is defined to be the sequence of data symbols for the *i*th stream. A prefix of M, containing the first j fragments for the redundancy vector $\overline{f}$ is defined in [8] as:

$$M(j, \overline{f}) = M_1(\overline{f})\, M_2(\overline{f}) \ldots M_j(\overline{f}) \text{ [8]}$$

The prefix $M(2, \overline{f})$ corresponds to receiving the first 2 packets of an image encoded with a redundancy vector $\overline{f}$. As more packets are received, the image quality will slowly increase. The quantity $g_i(\overline{f})$ is the amount by which the image quality increases when the receiver decodes fragment *i*. This assumes that all fragments prior to *i* have already been decoded. In the case of $g_1(\overline{f})$, it is defined as the difference in image quality between zero and one message fragment received. If PSNR is used to determine image quality, than $g_i(\overline{f})$ as mentioned in [8], is defined as the incremental PSNR of stream *i* given by:

$$g_i(\overline{f}) = \text{PSNR}[M(i, \overline{f})] - \text{PSNR}[M(i\text{-}1, \overline{f})] \text{ [8]}$$

The ULP framework uses progressive data, therefore we require $f_i \geq f_{i+1}$; $i = 1,2,\ldots,L\text{-}1$. The higher stream numbers have less FEC, so FEC assigned to streams is non-increasing with *i*. If $M_i(\overline{f})$ can be decoded, than $M_1(\overline{f})$, $M_2(\overline{f})$, …, $M_{i-1}(\overline{f})$ can also be decoded. Having more FEC in stream *i*+1 than in stream *i* offers no benefit, as a loss of more than f_i packets results in both streams being unrecoverable.

3.2.2 Channel Profile

To determine the FEC vector $\overline{f}$, a channel loss profile is used. The channel loss profile gives the probability of how many packets will be in error given how many are received. This estimate is given by a probability mass function (PMF) p_n; n = 0,1, … , N, such that p_n is the probability that n packets are lost. The PMF estimator can be based on a variety of channel models (uniform, binomial, Zipf, Poisson, exponential, Gilbert-Elliott).

Using the PMF, an estimate of the received image quality can be made. This is done by first calculating the cumulative distribution function $c(k)$ defined in [8] as

$$c(k) = \sum_{n=0}^{k} p_n \quad [8]$$

The cumulative distribution c($\overline{f_i}$) gives the probability that a receiver can decode stream $_i$. Now the expected PSNR of a received message can be defined as a function of the redundancy vector $\overline{f}$ and the number of terms in it L.

$$G(\overline{f}) = \sum_{i=1}^{L} c(r_i) g_i(\overline{f}) \quad [8]$$

3.2.3 ULP Algorithm

The ULP algorithm aims to find a good FEC assignment. Finding the globally optimal assignment of FEC data for each stream appears to be computationally prohibitive [3]. Therefore the ULP framework uses a local search hill-climbing algorithm that makes limited assumptions about the data. Two assumptions are made. First, the constraint of $f_i \geq f_{i+1}$ is made. Secondly, it is assumed that a single byte missing from the bit-stream causes all later bytes to become useless.

The algorithm starts out with all streams containing data symbols only. Thus, m_i = N and f_i = 0; i = 1, 2, … , L. The algorithm works by iteratively examining FEC assignments. During each iteration, the algorithm examines a number of possible assignments equal to 2QL, where Q is the search distance and L is the number of streams. The search distance is defined as the maximum number of FEC symbols that can be added or subtracted to a stream in one iteration. The larger the search distance, the more likely the global optimum will be found. However, a larger Q results in a longer run time. Next, $G(f)$ is determined after adding or subtracting 1 to Q symbols of FEC data to each stream. The f corresponding to the highest $G(f)$ is chosen and the allocation of the FEC data to all affected streams is updated. This process is repeated until none of the cases examined improves the expected image quality. This process is detailed more fully in the psudocode in Figure 12.

```
best[*] := (N, N, …, N)
Until best[*] = last[*] Do:
    last[*] := best[*]
    Foreach stream s from 1 to L:
        Foreach search_value from –Q to +Q
            temp[*] := last[*]
            temp[s] := temp[s] + search_value
            If temp[s] < 0 or temp[s] > N then continue to next search_value
            If search_value > 0 then for all i > s
                Do temp[i] := max(temp[s],temp[i])
            Else for all i < s
                Do temp[i] := min(temp[s],temp[i])
            End if
            Calculate expected image_quality
            If image_quality(temp[*]) > image_quality(best[*]) then
                best[*] := temp[*]
            End if
        End foreach
    End foreach
End until
```

Figure 12: Psudocode for the Unequal Loss Protection algorithm [8].

4 ULP Description and Design Considerations

In order to fully take advantage of a hardware implementation of an unequal loss protection system, the software implementation will need to be adjusted to fit the application and hardware resources.

4.1 Grouping of Packets

In the software implementation of ULP framework the user specifies a variety of inputs. One of these is the RS block size (n). Because the RS codes are interleaved among the packets, by specifying n, the user is also specifying how many packets to send. If the user specifies 128 symbols per block, then they are sending 128 packets.

In our FPGA system we want to send all the SPIHT data, not just n packets. If we were to send all the SPIHT data with the software ULP system, the packet size would be very large, or we would have very large Reed-Solomon blocks. With this in mind we will introduce the concept of a group. Each group contains n packets and a fixed number of streams. By sending multiple groups we can keep both the block length of the RS code and the size of each packet short, helping conserve hardware resources. Figure 13 shows 68 data symbols being sent in two packet groups. RS block codes span 7 symbols and each packet holds 7 data symbols. Streams 1-7 are in the first group, while 8-14 are in the second group.

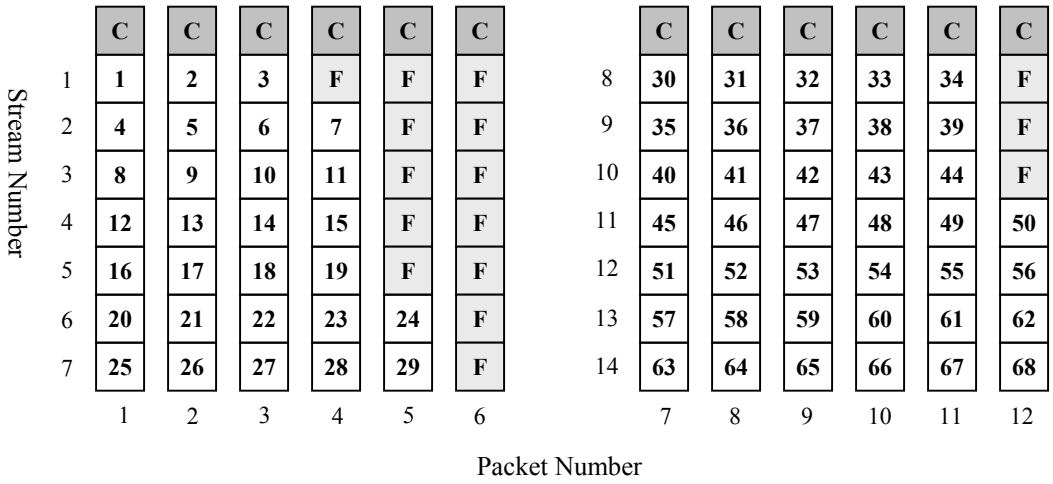

Figure 13: Data symbols 1-68 are sent in two packet groups.

4.2 Binomial Channel Model and fixed Bit Error Rate

The Unequal Loss Protection framework operates in a modular fashion where a different packet loss model can be used in the simulation. For our target platform we assumed that the channel has no memory and each error is an individual event. This assumption was made based on advice from NASA. Additionally, NASA engineers noted that they normally have undesirable channel conditions when the bit error rates (BER) are between 10^{-4} and 10^{-5}. With this in mind, we took the worst case and designed for a BER of 10^{-4}.

The ULP framework was designed for a packet loss channel, working with a packet loss rate (PLR). Therefore we needed to convert the BER to a PLR. This is done by first converting the BER to a packet error rate (PER). All packet errors are then considered to be losses in the channel. The PER will be dependent on the symbol size (m) of the RS code and how many symbols are in each packet (d). The PER is calculated by first finding the probability that all the bits in the packet are received correctly. Next this value is subtracted from 1.0, giving us the PER.

$$PER = 1 - (1\text{-}BER)^{m*d}$$

$$PLR = PER$$

It will be shown later that our system uses a packet size of 69 symbols, a symbol size of 8 bits, and a BER of 10^{-4}, resulting in a PLR of 5%.

4.3 Fixed RS(n,k) sequence

The ULP framework aims to map a set of RS codes to an image, providing graceful image degradation for a channel with known conditions. However, the framework requires an image quality vs. bit rate plot, which changes for each individual image. Generating a unique set of RS codes for an image will be slow and prohibitively expensive in hardware.

In order to have a solution that is equivalent to the software version we need two things. First, we need to be able to assign the RS(n,k) levels on the fly in hardware. Second, we need to be able to have a RS encoding circuit for each level. Additionally, the receiver will need to know the RS(n,k) value for each stream.

As mentioned before, the ULP framework requires an image quality vs. bit rate plot. Because each image has a unique plot, we can either generate a plot for each image or fix the RS code levels for the hardware encoder. Generating the image quality vs. bit rate plot requires implementing a SPIHT decoder in hardware and then running the decoder N times, where N is how many points in the image quality plot we want. It may be possible to get this information during the SPIHT encoding process in the future. Next, the ULP algorithm needs to be implemented in hardware. Doing all of this in hardware would require a large amount of resources and significantly slow down the system.

Instead of implementing the ULP algorithm in hardware, we will send a known order of RS codes. This drastically reduces the hardware complexity and runtime for the ULP module. However, this arrangement will create configurations that are not optimal for the data stream.

4.4 Optimal Packet Length

Our ULP system will be sending a lot of packets. Because of this it is important that the optimal packet size be chosen. In practice, setting the packet size will be dependent on the transmission protocol, bit error rate, and the amount of overhead per packet. A small packet size can create an inefficient solution due to fixed amount of overhead per packet. A large packet size is more susceptible to the bit error rate of the channel as more bits will be in the packet. For our purposes, we will assume the following:

- Packets lost are not re-transmitted
- The BER does not vary with time and is always 10^{-4}
- Each packet has a 32-bit CRC attached to it

With these assumptions, the optimal packet size [11] P_{opt} to be used is given by:

$$P_{opt} = \frac{h}{2}\left[\sqrt{1 - \frac{4}{h\ln(1-p)}} - 1\right]$$

h = overhead bits used for control, error detection, and framing (flags)
p = bit error rate

Using this equation we get a packet size of 550 bits, which is roughly 69 bytes.

4.5 Bandwidth Allocation Estimation

ULP works by trying to provide a near optimal set of RS(n,k) codes given a packet loss rate, an image quality plot, and a fixed amount of bits to send. Our last modification will be to try and give a near optimal estimate for the amount of data needed in the transmission. If too much is given, then ULPsim will allocate more FEC than needed.

Each image has a different looking image quality vs. bit rate plot. Some images will reach their peak image quality (approximately zero MSE) much sooner than others. If we allocate the same amount of space to all of these images then we can get very different RS(n,k) assignments. If an image is quick to reach its peak image quality, then more of the space allocated for the transmission will go to FEC. If the image does not reach its peak image quality in the space allocated for the transmission, then ULP will only add FEC that the channel model requires.

Figure 14 shows the packet loss rate needed to render an image un-reconstructible (all image quality is lost) for a given set of images. All images are treated independently of each other. In this example we used 69 byte packets, RS(128,k) codes, and a BER of 10^{-4}. A total of 2840 streams of data were sent over 42 packet groups.

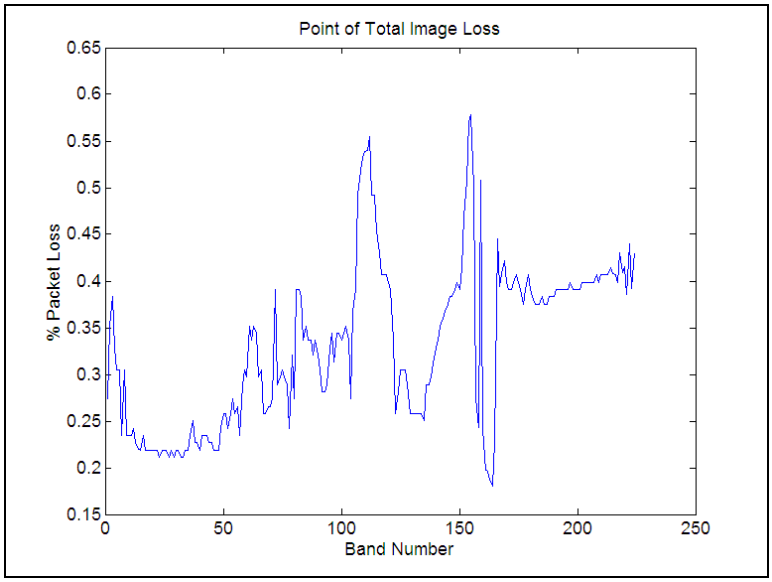

Figure 14: Packet loss rate resulting in complete image loss (224 different images).

As we can see from Figure 14, the point of total image loss varies from 20% to nearly 60% over all the bands. This large variation makes it nearly impossible to pick a decent static ULP assignment.

To correct this behavior we propose the idea of estimating the bandwidth needed to send a perfectly reconstructed image with the minimal FEC. Estimating the necessary bandwidth should give static ULP assignments that are similar to each other. When estimating the necessary bandwidth, we want enough room to allow the best image quality (approximately 0 MSE) with little or no packet loss.

From our experiments we have determined that adding an additional 27% to the number of bytes required for a nearly perfect reconstruction gives us a fairly constant packet loss rate for total image loss. Additionally, this additional 27% allows enough bytes to give very good image quality with little to no packet loss. Figure 15 below plots the packets loss rate for total image loss using this technique.

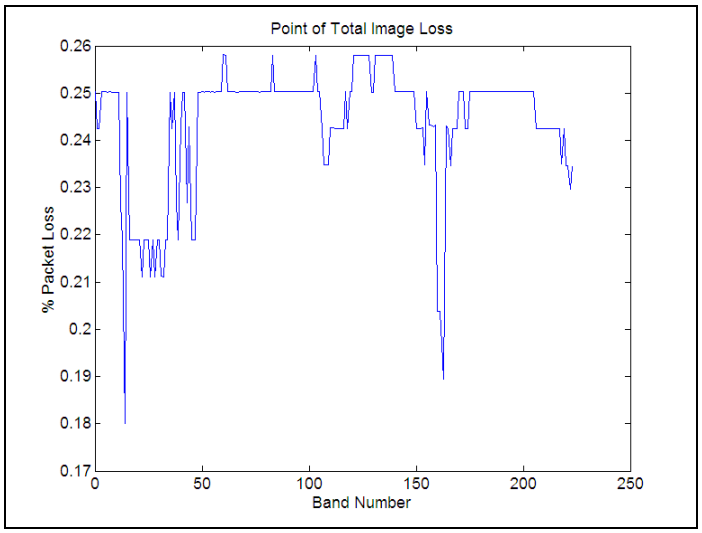

Figure 15: PLR required for total loss of the image using bandwidth estimation (224 images).

4.6 Design of Static ULP

Sending a fixed sequence of RS codes (Static ULP) as opposed to dynamically assigning them to an image (Dynamic ULP) will create a sub optimal ULP assignment. However, if this sub optimal assignment can produce results that are fairly close to the optimal, then static assignments can be justified.

Our ULP FPGA system has been designed using data from the Cuprite SPIHT image set. The Cuprite data set is a 224-band image taken from a NASA satellite. This image set allows us to work on data that are similar to what our hardware system will use.

Using the Cuprite data set, image quality vs. bit rate plots were generated using the fixed-point SPIHT decoder. To get a first estimate of our static ULP assignment we generated an average (arithmetic mean) image quality vs. bit rate plot. Our goal in taking the average is to try and minimize the deviation from the optimal assignment for each band in the image set. Figure 16 below shows the average Mean Square Error (MSE) vs. bit rate plot across all bands for the Cuprite dataset.

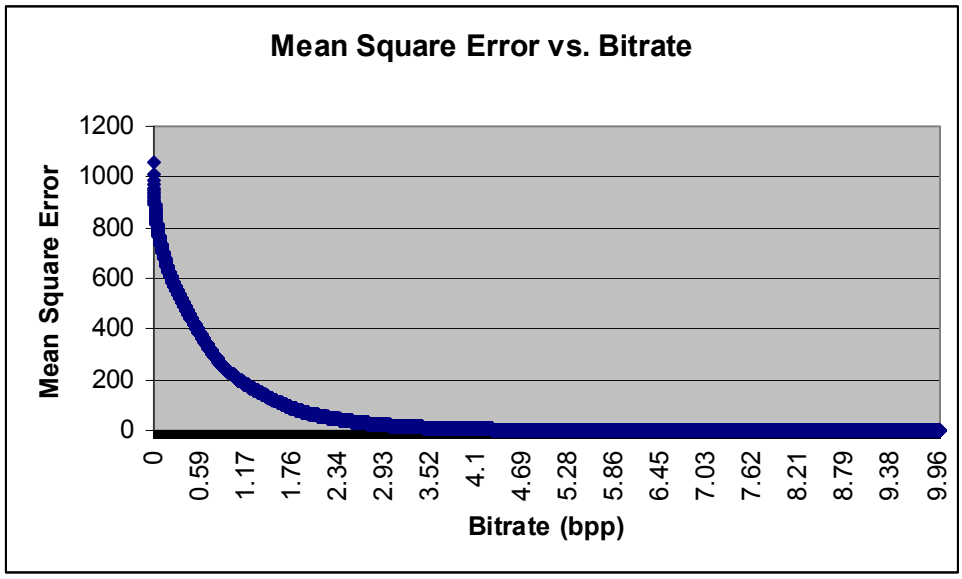

Figure 16: Plot of the average mean square error for a given bit rate (Bands0-223, Cuprite dataset).

For our ULP implementation we have chosen 128 packets per block to send. This produces codes of RS(128,k). This block size is hardware friendly and gives a nice range of RS(n,k) levels. Additionally, we are sending packets with a length of 69 bytes. After running ULPsim with this configuration and the data plot in Figure 16, we get the ULP assignment in Table 4.

Table 4: Static ULP assignment.

RS Level	Stream Start	Stream End	RS(n,k)
1	1	2	RS(128,100)
2	3	6	RS(128,101)
3	7	56	RS(128,102)
4	57	366	RS(128,103)
5	367	768	RS(128,104)
6	769	1150	RS(128,105)
7	1151	1468	RS(128,106)
8	1469	1768	RS(128,107)
9	1769	2062	RS(128,108)
10	2063	2358	RS(128,109)
11	2359	2530	RS(128,110)
12	2531	2650	RS(128,111)
13	2651	2746	RS(128,112)
14	2747	2908	RS(128,113)
15	2909	2968	RS(128,114)
16	2969	3068	RS(128,115)

Further, Figures 17 and 18 graph the number of correctable symbols and data symbols against the stream number.

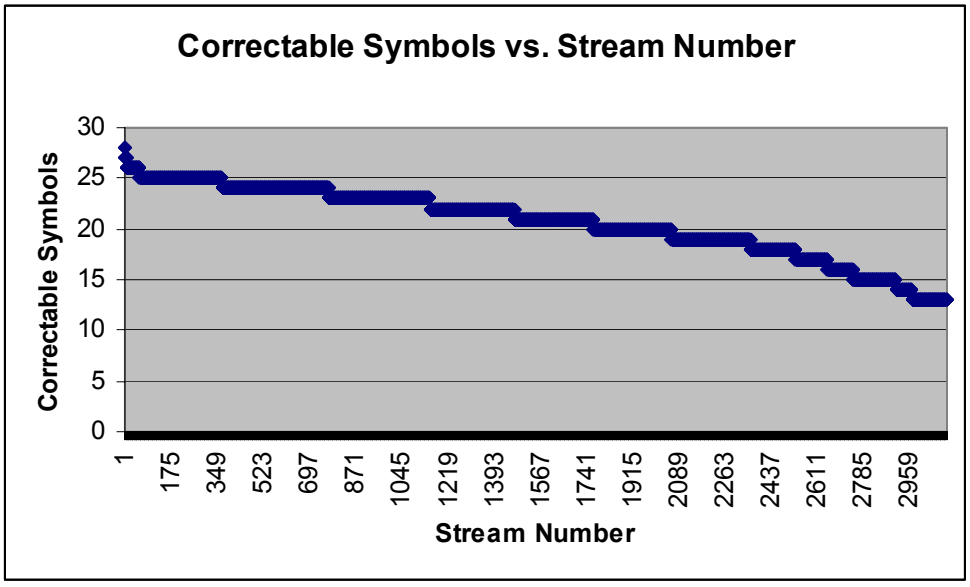

Figure 17: Number of correctable errors in each stream for the static ULP assignment.

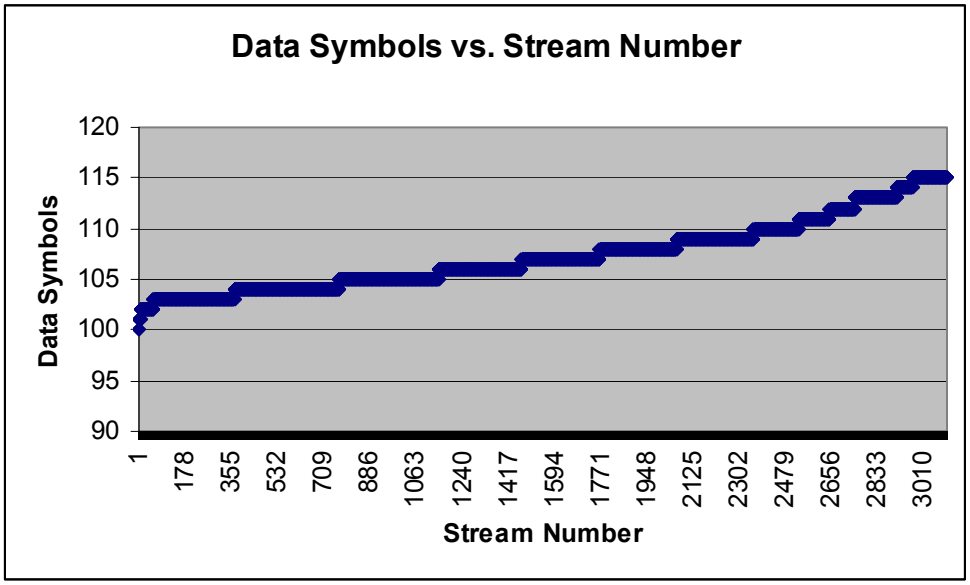

Figure 18: Number of data symbols in each stream for the static ULP assignment.

5 Architecture

5.1 Target Platform

Our target platform is the Wildstar FPGA processor board made by Annapolis Micro Systems [9]. A block diagram of the board can be seen below in Figure 19. The board contains three FPGAs: PE0, PE1, and PE2. It can operate at frequencies up to 133 MHz. 48 Mbytes of memory are available through 12 individual memory ports that are either 32 or 64 bits wide. This gives the board a memory throughput of up to 8.5 Gbytes/sec. There are four shared memory blocks, which can be accessed through a crossbar. By switching the crossbar, we can move processed data onto another FPGA in a pipelined fashion.

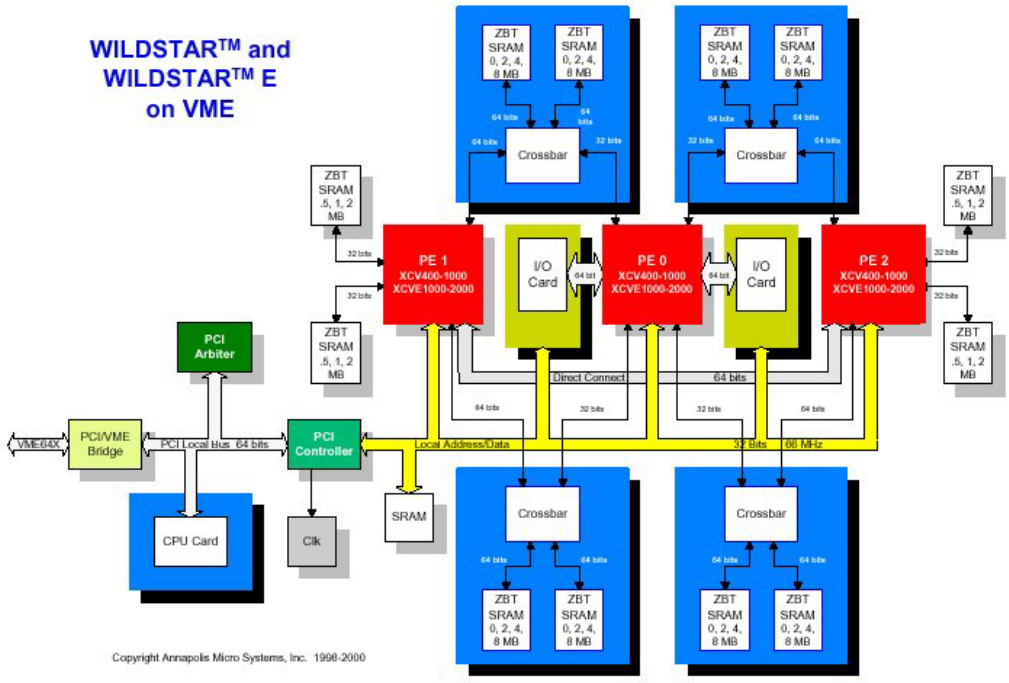

Figure 19: Annapolis Micro Systems Wildstar block diagram.

Our Wildstar board is equipped with 3 Xilinx Virtex 2000E FPGAs. Each Virtex 2000E allows for 2 million gate descriptions [10]. Each FPGA has 160 on-chip, dual ported, asynchronous blockrams. Each blockram stores 4096 bits of data and is accessible in 1,

2, 4, 8, or 16 bit wide words. Because each blockram is dual ported, they function well as FIFOs.

The Wildstar board connects to a host computer via a PCI bus. Commands are sent to the FPGA board via a host program residing on the computer. This connection is the limiting factor on performance. However, as our system is designed to be in a satellite, this limitation would not be present in a real system.

5.2 Design Overview

Our ULP system is designed to work with the SPIHT system created previously. The previous system fully utilized the Wildstar board. Because all three FPGAs were used, the ULP system cannot be easily integrated into the existing SPIHT system. Figure 20 below shows the design overview of the existing SPIHT system.

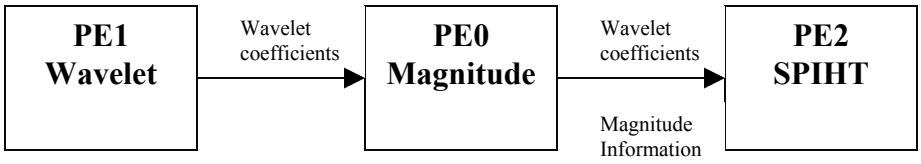

Figure 20: Overview of previously created SPIHT system.

The enhanced SPIHT system will take the SPIHT data written from the previous system and load it to a shared memory accessible by PE1. Once loaded, the ULP encoding will be run on PE1. Once finished, the encoded data will be written back to the host system. Figure 21 shows the design overview of the ULP FPGA system.

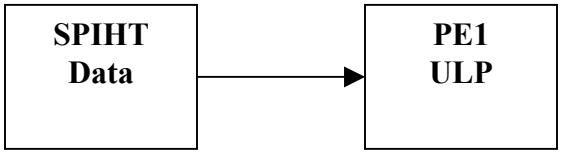

Figure 21: Design overview of ULP FPGA system.

5.3 Modified RS Encoder Circuit

The ULP phase needs to encode the data at several different RS(n,k) levels. The simplest way to do this is to have a modified RS circuit. In a normal RS encoder circuit the Galois multipliers multiply the feedback symbols with a single, fixed polynomial. Instead of using a fixed, static multiplier, we can use a variable multiplier. This allows us to reuse the RS encoder circuit for multiple RS(n,k) values.

Next, because we have multiple RS(n,k) values using the same encoder circuit we will need to adjust which data are added to the incoming raw data. This adjustment will be based upon which RS(n,k) code is currently being encoded. Remembering from our previous discussion of Reed-Solomon codes, different values of k require different lengths of the LSFR circuit. By simply inserting a multiplexer into the circuit we can accomplish this task. Figure 22 below shows a model of the ULP encoder circuit.

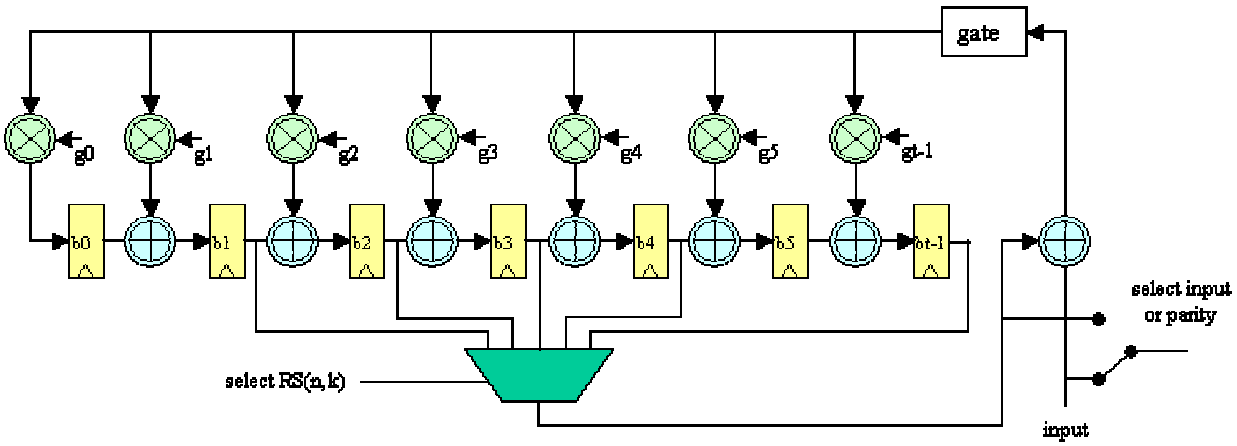

Figure 22: Modified RS(n,k) encoder

5.4 Static vs. Variable Galois Multipliers

Instead of using a variable Galois multiplier in the RS encoder, a number of static (fixed) multipliers can also be used. If the number of different RS(n,k) codes in the ULP assignment is small, then static multipliers can possibly offer greater performance and a

small encoder circuit. On average a scalar Galois multiplier requires ½m^2-m XOR gates, where m is GF(2^m) [15].

There are a number of different variable Galois bit parallel multiplication architectures [15]. In our case we used the Mastrovito multiplier architecture [15], which uses at least $2m^2$ –1 gates (AND + XOR). The number of gates will actually depend on which primitive polynomial the field is constructed from. In our case, GF(2^8), the multiplier uses 84 XOR and 64 AND gates.

We can use the gate counts to determine whether static or variable multipliers should be considered. Using static multiplier gate complexity as (# levels)*(½m^2-m) and variable multiplier gate complexity as 148 gates, we find that designs with fewer than 6 levels should consider static multipliers. Figure 23 shows our theoretical gate numbers. It should be noted that these are only estimates. FPGAs use LUTs and not gates, therefore gate count may not mirror LUT usage and overall delay. In the static designs, we have omitted the gates needed for muxing multiple static multipliers. In designs using variable multipliers, we have omitted the flip-flops needed to store the appropriate values for each RS(n,k) level.

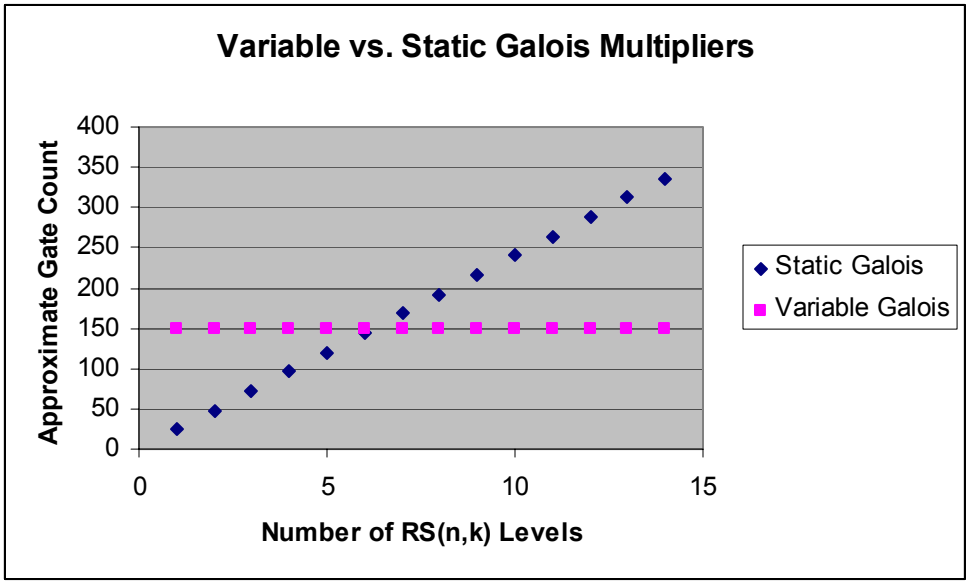

Figure 23: Approximate gate comparison for a Galois multiplier in a modified RS encoder circuit.

5.5 Parallel RS(n,k) generation

Our SPIHT FPGA system is bounded by the wavelet magnitude generation phase. Its throughput is 100 mega pixels per second. Our system works on 8-bit data, so our ULP throughput needs to be at least 100 mega-bytes per second.

It is important to note that the input and output throughput of the ULP system will be different. Because RS encoders add redundant data to the bit stream, the true bit rate of the system will decrease. The true bit rate will be given by:

$$\text{True Bit Rate} = (\text{raw data throughput})\frac{k}{n}$$

In order to sustain 100 Mbytes/sec, multiple RS encoders will need to be running in parallel. Without multiple encoders, a single RS encoder would need to be running at speeds over 100 MHz using 8-bit symbols. By running RS encoders in parallel we can increase our throughput and decrease our required system clock frequency. For our system we chose to run 8 encoders in parallel. By doing this we are easily able to make our goal of a 100 Mbytes/sec true bit rate.

Running RS encoders in parallel on SPIHT data adds a large amount of complexity to the FPGA design. ULP works with SPIHT's progressive bit stream to generate RS codes. Therefore running multiple RS encoders in parallel requires that the SPIHT data are split up into sections for each encoder. Failing to do so means that the SPIHT data will not be fully interleaved in the packets sent. This will reduce the effectiveness of the ULP framework. Figure 24 demonstrates this.

C	C	C	C	C	C
1	3	5	F	F	F
2	4	6	7	F	F
8	10	12	14	F	F
9	11	13	15	16	F
17	19	21	23	25	F
18	20	22	24	26	F
27	29	31	33	35	37
28	30	32	34	36	38

C	C	C	C	C	C
1	2	3	F	F	F
4	5	6	7	F	F
8	9	10	11	F	F
12	13	14	15	16	F
17	18	19	20	21	F
22	23	24	25	26	F
27	28	29	30	31	32
33	34	35	36	37	38

Figure 24: Parallel RS(n,k) generation without (left) and with (right) data sectioning.

Figure 24 shows an example of two RS encoders working in parallel. If data sectioning is not used (left example), the data are fetched from memory and then written to one FIFO. The data are then taken out from the FIFO and split up between the encoders. By doing this we have started to destroy the interleaving of the RS block data. We have also not completely followed the ULP framework. Certain bytes may be in RS blocks with less or more FEC than the ULP assignment called for. For example, in Figure 24, byte 2 is protected with 2 FEC bytes instead of 3 as the ULP assignment called for. The right example in Figure 24 shows the correct ULP assignment.

In order to correctly assign FEC when running RS encoders in parallel we need to distribute sections of the SPIHT bit stream to the appropriate RS encoder. This distribution scheme will cause the parallel RS encoders to stall for a small amount of time. However, once a RS encoder has some data to start encoding, it can begin while its data are replenished in the background.

Memory is initially put into a doublewide data buffer. Therefore, a read from memory is valid for 2 clock cycles. Attached to the data buffer are two FIFO loaders that each control loading for eight FIFOs. The FIFOs act as individual memories for each RS encoder. With the data from memory being progressive, normally only one FIFO loader will be active at a given time. However, when the data buffer contains the end of the data

for RS encoder x and the start for RS encoder x+1, both will access the buffer at the same time. Figure 25 gives a block diagram of the system.

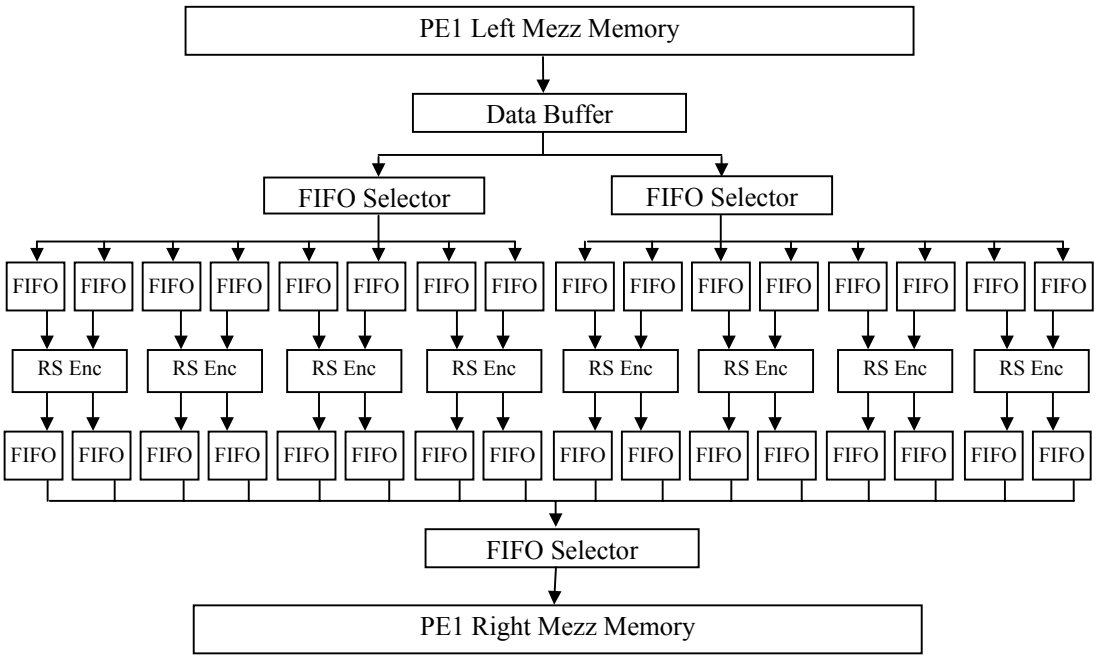

Figure 25: Block diagram of parallel Reed-Solomon generation system.

The data from the RS encoders are then sent to an individual FIFO. An additional FIFO selector then sequentially pulls out the Reed-Solomon block codes. This data are then written to a shared memory.

5.6 Speedup using C-slow retiming

Additional performance was obtained through the use of C-slow retiming. C-slow retiming is a pipelining technique for circuits with feedback paths. It aims to increase the throughput of the circuit at the cost of clock cycle delay.

C-slow retiming involves adding C additional registers in the design stage [7]. If a circuit were 3-slowed, every register in the circuit would be replace with 3 registers. After the registers are added, the circuit is retimed. Retiming involves moving registers in an attempt to shorten the critical path delay. Registers are moved as long as they do not

affect the functionality of the circuit. Figure 26 gives an example of a circuit being 2-slowed. Its critical path goes from 5 to 2 after C-slowing.

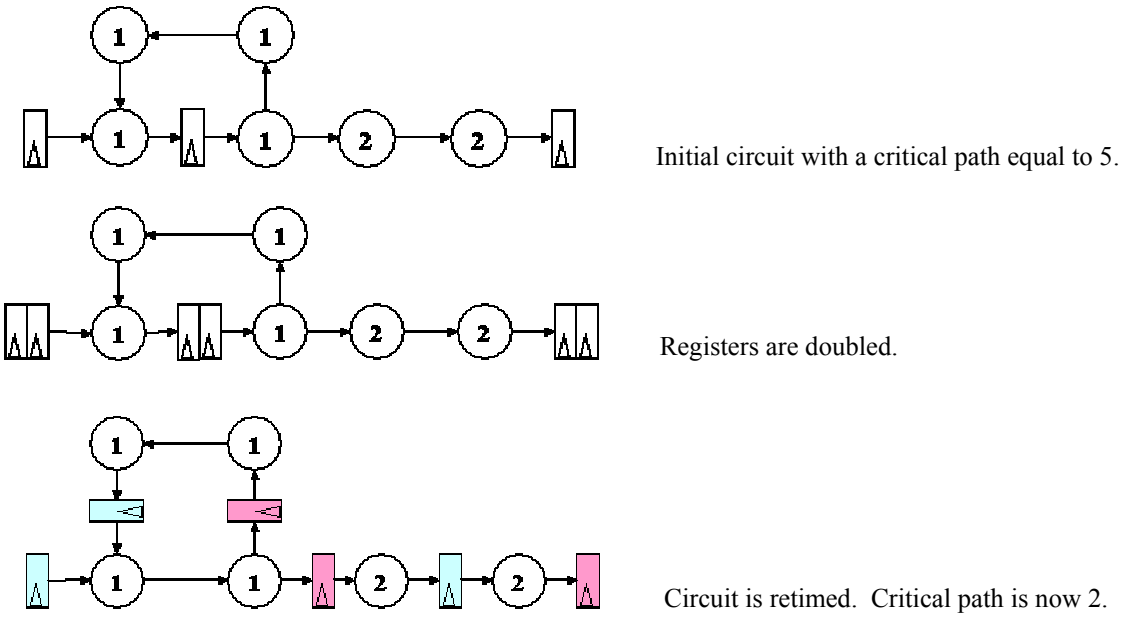

Figure 26: Example of a circuit being 2-slowed.

C-slow retiming only works if the circuit can use independent data sets. In other words, if the data on each successive clock cycle is unrelated to the previous clock cycle, then C-slow retiming can be used. We are in effect operating on C different datasets per clock cycle.

A disadvantage of C-slow retiming is that the delay per clock cycle increases by the number of registers that are added. However, as long as the critical path can be decreased, the latency of the circuit will stay the same.

For our design we chose to add an additional set of registers, effectively creating a 2-slow circuit. This had the greatest benefit on the system clock. This is why Figure 25 shows each RS encoder using two different FIFOs for reads and two different FIFOs for writes. 2-slowing essentially turns a RS encoder into two encoders.

6 Design Results

6.1 FPGA Implementation Numbers

The ULP FPGA implementation was designed using VHDL with models for the FPGA board provided by Annapolis Micro Systems. These models were used to access the PCI bus and memory ports. Simulations were done using ModelSim SE 5.6d from Mentor Graphics. Netlist generation was done using Synplify 7.1 from Synplicity. Place and route was done using Xilinx Foundation Series 3.1i tool set. Lastly, the utility peutil (supplied by Annapolis Micro Systems) was used to generate the FPGA configuration.

The static ULP system has 8 RS encoders operating in parallel. Each encoder is 2-slowed, effectively doubling the clock rate of each circuit. So while there are effectively 16 different encoders, only 8 will be operational at one time. 2-slowing each RS encoder circuit allows the system clock to run at 26 MHz. This gives us a raw throughput rate of 208 MB/sec.

The RS(n,k) assignments vary from RS(128,100) to RS(128,115). Our true bit rate will be limited by the amount of redundancy added and any overhead the system has. Accounting for overhead and redundancy added to the bit stream, our true bit rate is effectively 171 MB/sec. This exceeds our desired throughput of 100 MB/sec. Table 5 shows the speed and runtime specifications of the enhanced SPIHT system.

Table 5: Performance numbers of enhanced SPIHT system.

Phase	Clock Cycles per 512x512 Image	Clock Cycles per Pixel	Clock Rate	Throughput	FPGA Area
Wavelet	182465	3/4	75 MHz	100 MPixels/sec	62%
Magnitude	131132	1/2	73 Mhz	146 MPixels/sec	24%
SPIHT	65793	1/4	56 Mhz	224 MPixels/sec	98%
ULP	39855	1/8	26 Mhz	171 MPixels/sec	74%

6.2 Performance of Static ULP

The static ULP assignment will now be compared against dynamic ULP assignment, equal loss protection (ELP) set for each individual image (dynamic ELP), and SPIHT data sent without error correction. Our system used the Cuprite dataset with images compressed at 9.9 bits per pixel. All comparisons will be made to the individual bands within it.

We want to look at how the static assignment performs on the average MSE data that was used to create it. From Figure 27 below we can see that we should expect a near perfect image between packet loss rates of 0-11%. After a PLR of 11% we see a graceful degradation of image quality until a PLR of 21%. At a PLR of 21%, FEC no longer protects the image and we have an unprotected bit stream[1]. In this case, 5504 packets are sent with our BER of 10^{-4} corresponding to a PLR of 5%.

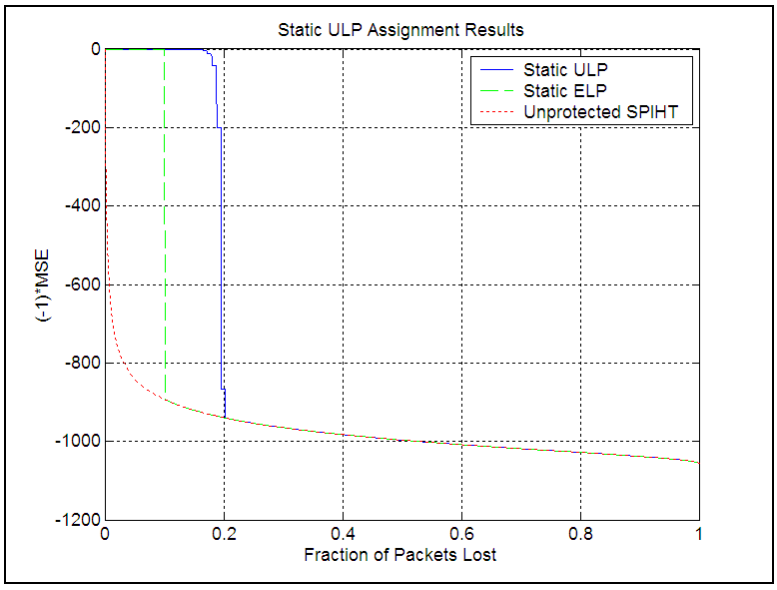

Figure 27: Static ULP assignment performance on data used to produce it.

[1] Unprotected SPIHT values determined by randomly generating bit errors in a Monte Carlo simulation with 10,000 iterations.

To analyze the static ULP over all the SPIHT bands we will calculate the MSE difference between the static and dynamic assignments for each band and then average them. Figure 28 gives the graph of the average MSE difference at a given packet loss rate. A positive MSE difference indicates that static ULP is doing worse than the dynamic assignment. What we see is a large spike shortly after a PLR of 20%. Going back to Figure 27, we see that the static ULP assignment does not protect against a PLR above 20%, so this is to be expected.

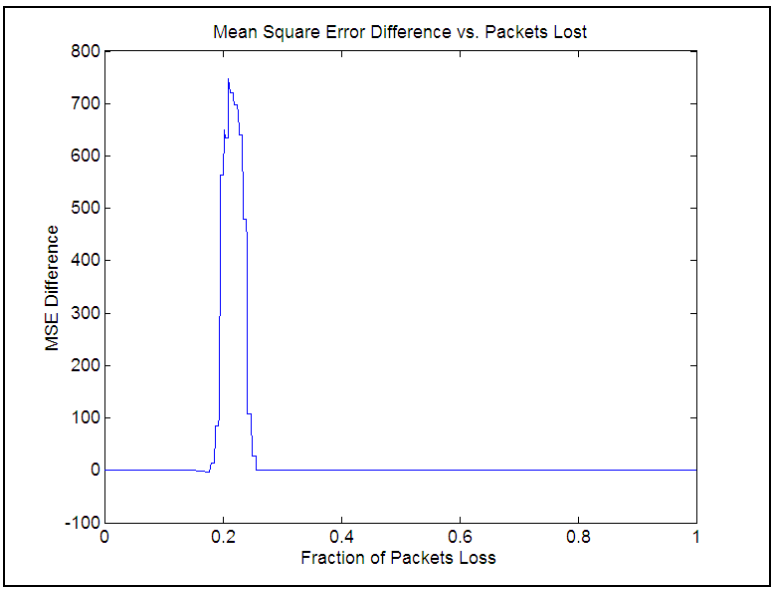

Figure 28: Average MSE difference between Static ULP and Dynamic ULP assignments across all bands in the Cuprite dataset.

If we look closely at the MSE difference between packet loss rates of 10-20% we can see that the maximum MSE difference is around 85, which occurs at rate of approximately 19%. After this point, static ULP has broken down and is no longer useful. Figure 29 verifies this behavior, giving an enlarged region of Figure 28 for the rates 0-0.2.

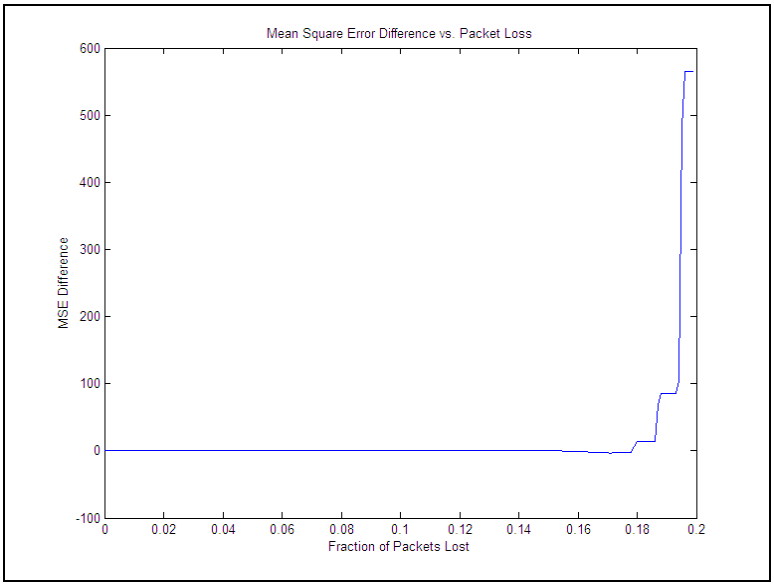

Figure 29: Close up of MSE difference from all the bands in the Cuprite dataset.

We are interested in the MSE difference between the rates of 10-20% as this is the region where a graceful degradation of image quality will occur. From Figure 29, we can see that there are three main points where static ULP differs from dynamic ULP. Between rates 0-0.178 static ULP differs at most with dynamic ULP by 3 MSE. Between rates 0.18-0.186 they differ by approximately 14.2 MSE. And between rates 0.187-0.190 they differ by 85.5 MSE.

Next we can take a closer look at the MSE difference by looking at the distributions of the MSE difference. Figures 30, 31, 32, 33, and 34 show the histograms for packet loss rates of 0.15, 0.16, 0.17, 0.18, and 0.19 respectively. In all figures the histogram is centered on a MSE difference close to zero. At a PLR of 0.15 the points spread out approximately 0.5 MSE from zero. At a PLR of 0.16, this spread increases to 1.8 MSE. At 0.17, the spread is approximately 5 MSE. Packet loss rates of 0.18 and 0.19 are much worse with the spread at 50 and 500 respectively.

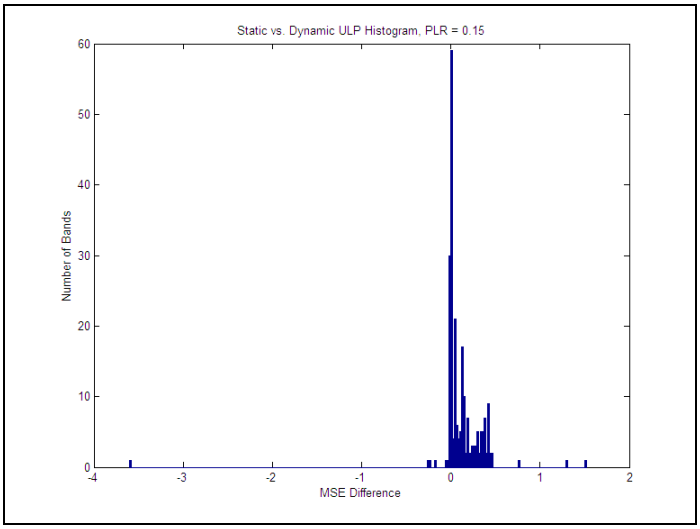

Figure 30: MSE difference histogram of Cuprite dataset for Static and Dynamic ULP (PLR=0.15).

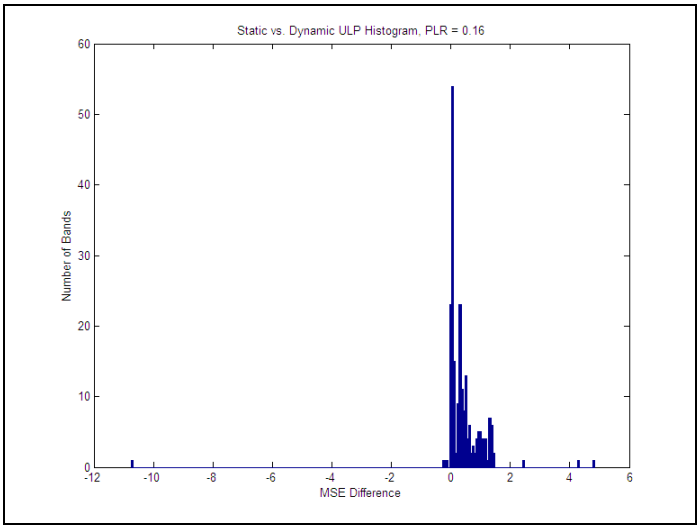

Figure 31: MSE difference histogram of Cuprite dataset for Static and Dynamic ULP (PLR=0.16).

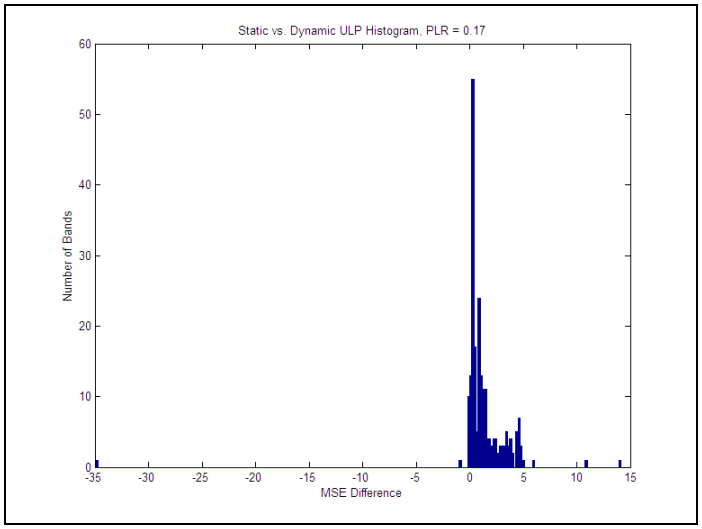

Figure 32: MSE difference histogram of Cuprite dataset for Static and Dynamic ULP (PLR=0.17).

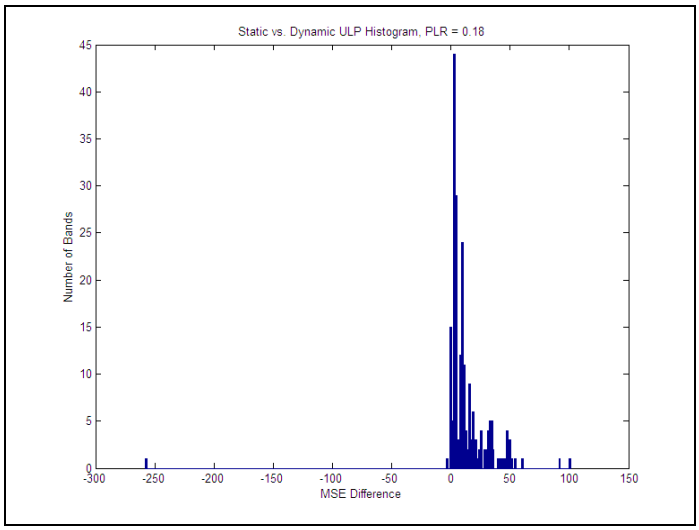

Figure 33: MSE difference histogram of Cuprite dataset for Static and Dynamic ULP (PLR=0.18).

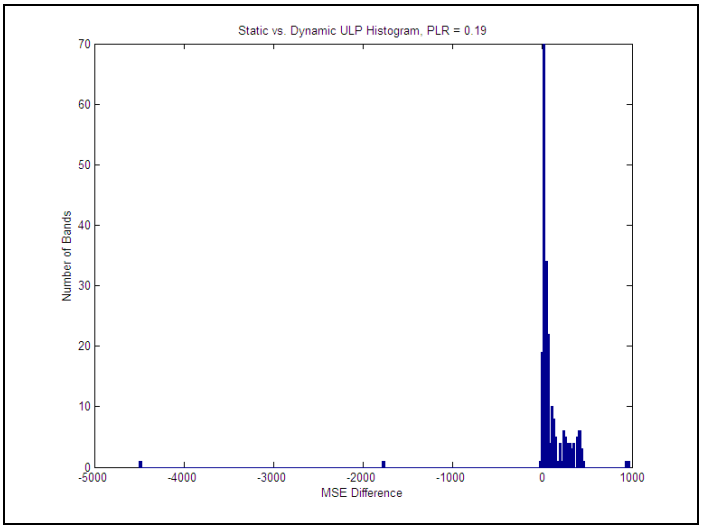

Figure 34: MSE difference histogram of Cuprite dataset for Static and Dynamic ULP (PLR=0.19).

From Figures 30, 31, 32, 33, and 34 we notice that the histograms are less centered on a MSE difference of zero as the PLR increases. Additionally, as the PLR increases, the MSE differences are continually increasing and spreading out. Therefore, with increasing packet loss, the individual characteristics each image has begin to take over and significantly degrade the static ULP assignment.

Next we will compare the static ULP assignment with a dynamic ELP assignment. We notice that the static ULP performs better than dynamic ELP up to a PLR of 0.185. This is shown by a negative MSE, which peaks at –192 MSE. The negative step curve indicates that the static ULP is giving a graceful degradation of image quality.

After a PLR of 0.185, a positive MSE is seen. At this point, dynamic ELP is giving better performance than static ULP. Dynamic ELP peaks at a PLR of 0.205, giving 700 MSE better on average. Figure 35 plots the MSE difference between the static ULP and dynamic ELP across all bands. The reason behind the dual peak graph is that there are cases where an image will have FEC added to it at a rate greater than what static ULP assigns.

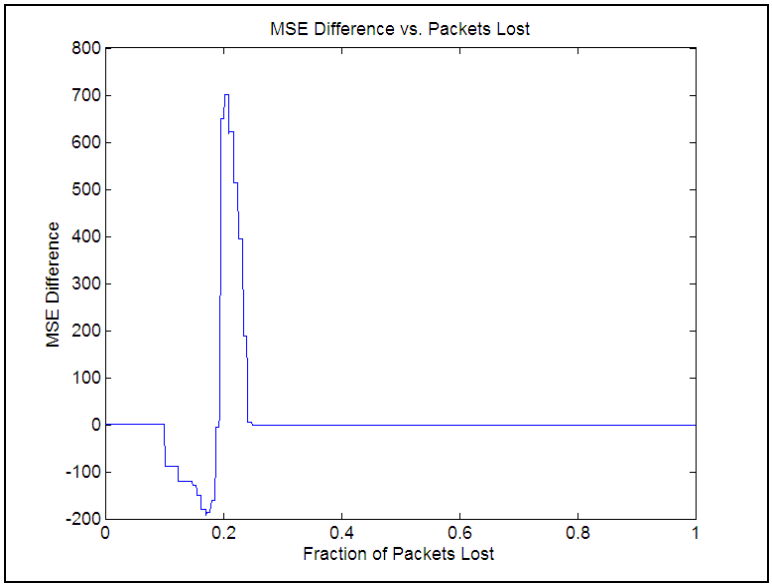

Figure 35: Average MSE difference between Static ULP and Dynamic ELP assignments across all bands in the Cuprite dataset.

Looking more in detail at each individual band for a given packet loss rate we see that the data has a higher deviation than the comparison between static and dynamic ULP. As the PLR increases we see small peaks around 1000 and 2000 MSE. However, the strongest peak is still centered on 0 MSE indicating that the static ULP assignment is not performing much better than the dynamic ELP assignments. Figures 36, 37, 38, and 39 show the histograms for packet loss rates of 0.18, 0.19, 0.20, and 0.21.

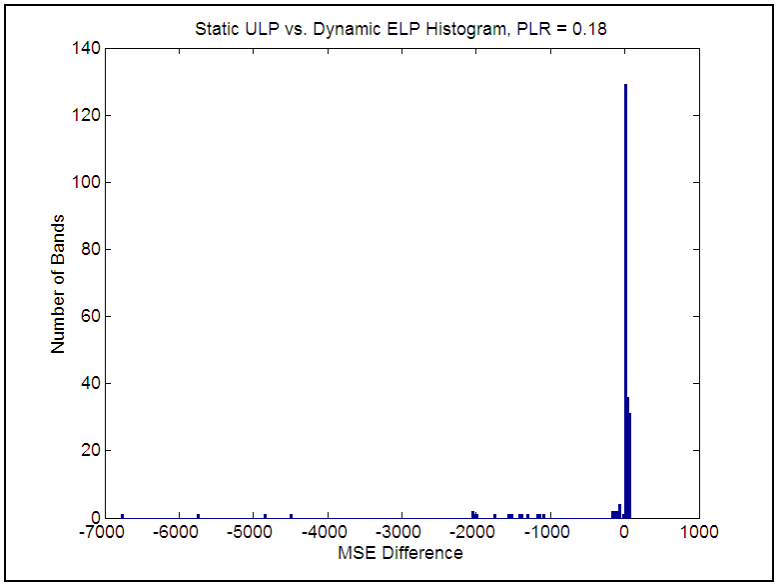

Figure 36: MSE difference histogram of Cuprite dataset for Static ULP and Dynamic ELP (PLR=0.18).

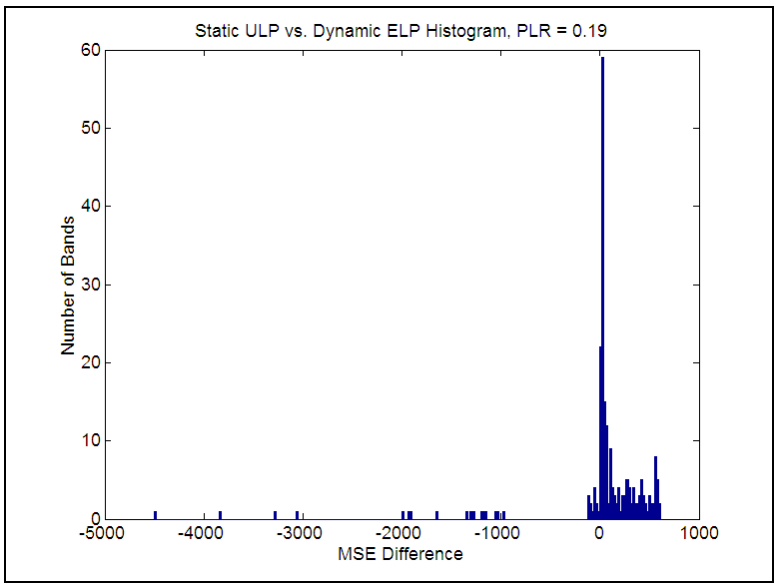

Figure 37: MSE difference histogram of Cuprite dataset for Static ULP and Dynamic ELP (PLR=0.19).

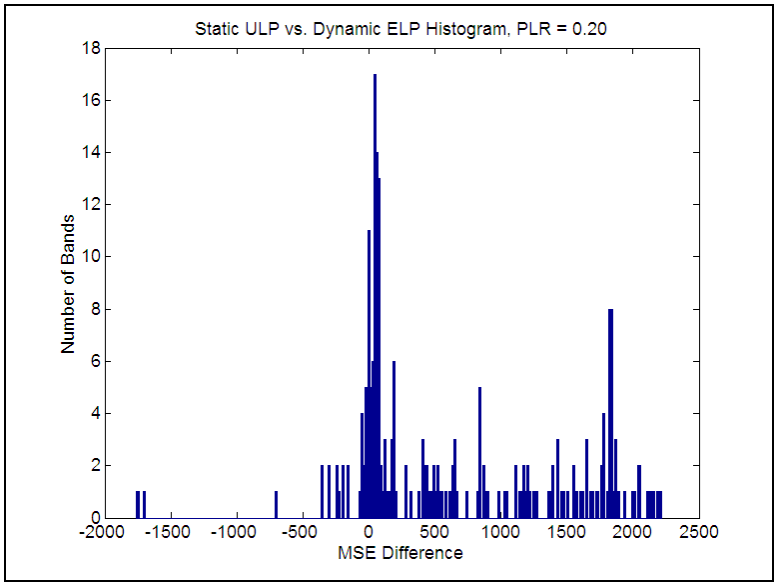

Figure 38: MSE difference histogram of Cuprite dataset for Static ULP and Dynamic ELP (PLR=0.20).

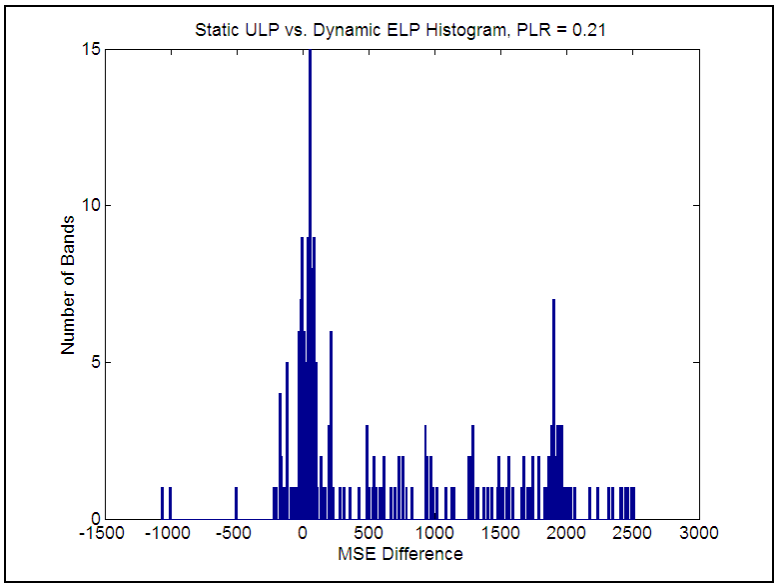

Figure 39: MSE difference histogram of Cuprite dataset for Static ULP and Dynamic ELP (PLR=0.21).

From Figures 36, 37, 38, and 39 we notice that the histograms are less centered on a MSE difference of zero as the PLR increases. Additionally, as the PLR increases, the MSE differences are continually increasing and spreading out. This behavior is very similar to the histogram comparison of static and dynamic ULP. Again, similar to dynamic ULP, the individual characteristics each image has begin to take over and significantly degrade the static ULP assignment.

7 Conclusions and Future Work

In this work we have demonstrated viable Reed-Solomon encoder architectures implemented on a Xilinx Virtex 2000E FPGA. The circuits are designed for a progressive bit stream and match the slowest stage of the SPIHT FPGA system. These RS encoders work together to provide a good degradation of image quality as the fraction of packets lost increases. Our architecture performs nearly as well as a dynamic unequal loss protection framework at packet loss rates up to 18%.

Additional static assignments could be looked at. We used the arithmetic mean for the Cuprite dataset. Using additional datasets might prove useful. Additionally, the static ULP assignment could be based upon a number of things including:

- Geometric mean
- Arithmetic-geometric mean
- Worst possible MSE

Future work can be done to increase the throughput of the system. The following design changes could boost performance:

- Additional C-slow levels
- Composite field Galois multipliers
- Memory based RS encoder configuration
- Additional RS encoders in parallel

Additional C-slow levels can add performance if the critical path remains in the Galois multipliers. Composite field multipliers have shown good performance [16], and could also boost performance. Storing the values for the RS generator function in memory and then loading them to a register could also provide a performance boost. Currently the system calculates the RS generator function based upon the amount of redundancy required. Finally, filling the remaining FPGA area with RS encoders can add additional throughput.

Our data-compression work is part of an ongoing development effort funded by NASA.

8 References

[1] A. Said and W. A. Pearlman, "A new, fast, and efficient image codec based on set partitioning in hierarchical trees," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 6, pp. 243-250, June 1996.

[2] J. M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," *IEEE Trans. Signal Processing*, vol. 41, pp. 3445-3462, Dec. 1993.

[3] A. Mohr, E. A. Riskin, R. E. Ladner "Unequal Loss Protection: Graceful degradation of Image Quality over Packet Erasure Channels Through Forward Error Correction," *IEEE Journal on Selected Areas in Communications*, vol. 18, pp. 819-828, June 2000.

[4] R. H. Katz, *Contemporary Logic Design*, The Benjamin/Cummings Publishing Company, Inc. Redwood City, CA. pp. 524-538, 1994.

[5] M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies, "Image coding using wavelet transform," *IEEE Trans. Image Processing*, vol. 1, pp.205-220, Apr. 1992.

[6] V. R. Algazi, R. R. Estes, "Analysis based coding of image transform and subband coefficients," *Applications of Digital Image Processing XVIII*, vol. 2564 of *SPIE Proceedings*, pp.11-21, 1995.

[7] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *Proceedings of the ACM International Conference on Supercomputing*, June 1990.

[8] T. W. Fry, *Hyperspectral Image Compression on Reconfigurable Platforms*, Master Thesis, University of Washington, Seattle, Washington, 2001.

[9] Annapolis Microsystems. *Wildstar Reference Manual*, Maryland: Annapolis Microsystems, 2000.

[10] Xilinx, Inc., *The Programmable Logic Data Book*, California: Xilinx, Inc., 2000.

[11] M. Schwartz, *Telecommunication Networks: Protocols, Modeling and Analysis*, Addison-Westley, Reading, MA, 1987.

[12] The MacTutor History of Mathematics Archive – Evariste Galois. School of Mathematics and Statistics, University of St Andrews, Scotland, http://www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Galois.html.

[13] P. G. Sherwood and K. Zeger, "Progressive Image Coding for Noisy Channels," *IEEE Signal Processing Letters*, vol. 4, no. 7, July 1997.

[14] H. Imai, *Essentials of Error-Control Coding Techniques*, Academic Press, San Diego, CA, 1990.

[15] E. Mastrovito, *VLSI Architectures for Computation in Galois Fields*. PhD thesis, Linköping University, Dept. Elect. Eng., Linköping, Sweden, 1991.

[16] C. Paar, "Comparison of Arithmetic Architectures for Reed-Solomon Decoders in Reconfigurable Hardware," *In Proceedings for FCCM*, 1997.

[17] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *ACM Computer Communication Review*, vol. 27, pp. 24-36, Apr. 1997.

[18] I. S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol 8, pp. 300-304, June 1960.