ANALYSES OF CODING AND COMPRESSION STRATEGIES FOR DATA STORAGE AND TRANSMISSION

Thesis by

Masahiro Sayano

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1992

Defended 15 May 1992

ACKNOWLEDGMENTS

There are many people who have helped me with my research and thesis; their forms of assistance ranged from academic and research advice, computer support, administrative assistance, and moral and material support. Almost everyone with whom I have come into contact in the last four years has helped me in this regard. However, there are a few to whom I am especially indebted.

Professor Rodney M. Goodman, my Research Advisor, has given me many and varied opportunities for education and research. Not only is he responsible for getting me started on my research topics, he has also given me the freedom to explore, the encouragement to advance, and the opportunities to present my work in various forums. Without his guidance I would not have had the chance to grow.

Professor Robert M. McEliece of Caltech, Professor Patrick G. Farrell of the University of Manchester, United Kingdom and Dr. Mario Blaum of the IBM Almaden Research Center have all selflessly given me advice and assistance on probability and coding theory. Their guidance was a key factor in my obtaining some of the results on my projects on memory reliability and error correction coding.

Dr. Kar-Ming Cheung of the Jet Propulsion Laboratory has given me opportunity and advice in the field of image compression. His advice, especially during the summer that I was working at JPL, and his generosity have helped immensely in the work I have done in this field.

I thank my family, who have provided moral and material support during these four years and the many years before.

Parts of this work were supported through grants from the National Science Foundation and the Jet Propulsion Laboratory.

ABSTRACT

Selected topics in error correction coding and data compression for data storage and transmission will be analyzed here. In particular, a model for the mean time to failure for computer memories protected by error correction coding, characteristics and applications of phased burst error correcting array codes, and locally adaptive vector quantization for image and data compression will be examined.

A model of the mean time to failure (MTTF) of semiconductor random access memories protected by single error correcting—double error detecting (SEC-DED) codes on the chip and with soft error scrubbing and multiple types of hard failures will be presented. Only a few assumptions and approximations will be made. This model will provide a more complete picture of the expected failure modes, reliability, and the mean time to failure of memory systems protected by on-chip error correction coding. Special cases will also be addressed, such as slow or fast scrubbing and dominance of hard or soft errors.

Characteristics of a family of phased burst error correcting array codes will be addressed. In particular, allowable and optimal code sizes will be examined. When used in non-binary applications, these codes retain their characteristics and can correct "approximate" errors with even higher rate: If the amount any q-ary symbol can be in error is bounded by some value, these codes can be designed to address this type of error with even fewer check symbols.

Improvements to a locally adaptive vector quantization compression strategy will be discussed. The basic strategy involves reorganization of the code book after each use so that the most recent codewords are moved to the front. With the various improvements covered in this work, the algorithm is capable of matching the performance of other more computationally intensive algorithms at a fraction of the computational complexity.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

As digital data become more the standard mode of communication, the need to analyze and further understand the means by which this communication can be better accomplished becomes paramount. This process is well under way as evidenced by the volume of research being done in this field. Data storage also falls within this regime, for data storage is the equivalent of data communication over time as opposed to data communication over distance. Data storage concerns, therefore, are closely linked to data communication concerns: reliable and rapid transmission of data. For data storage, these concerns are interpreted as a need for a reliable storage medium and efficient coding methods for data correction and compression.

The primary motivation for this work is the need for dense storage of large amounts of digital data on silicon. At present, practically all silicon memory systems in operational use are digital in nature. One of few exceptions is charged coupled devices (CCD) for imaging systems, but these are not meant for large scale semi-permanent information storage. Dense silicon storage systems will allow fast access to a large amount of data. Such ability is necessary for computer storage of various forms of data typically stored in arrays: sound and image data are two common examples.

The three topics explored in this work deal with the concerns of designers of such systems. The topics cover specific points in each of the three areas of storage media, coding, and applications. These topics are the study of reliability of semiconductor random access memories, properties of a family of phased burst error correcting codes, and a locally adaptive vector quantization algorithm for image compression, respectively.

Semiconductor memory reliability is of interest to many who utilize computer systems in which high reliability is a precious commodity. Reliability may be necessary due to having many parts (such as in a large computer system) or having a hostile environment (such as on spacecraft). As memory systems increase in size and complexity, the mean time between system failures can decrease beyond an acceptable point. Testing the reliability of these complex systems, however, can be exceedingly costly. The designer must therefore utilize models to predict behavior prior to construction.

Chapter 2 presents one model for semiconductor memory reliability. The model includes employment of error correction coding directly on each chip for on-the-fly correction. Features of the model include accounting for various memory failure types, periodic removal of soft errors (alpha particle induced errors) through scrubbing, and multiple block construction which large capacity memory chips employ. The model is shown to be accurate through computer simulation for a variety of parameters. Furthermore, supplementary models for special and limiting cases are also explored.

The error correction coding used here is the single error correcting—double error detecting (SEC-DED) code family. Different codes are more matched for different applications. One such case is when an extension of CCD-type storage cells is utilized to increase storage capacity: Multi-valued memory, where a dynamic random access memory cell stores a discrete charge value corresponding to several bits of information, provides a means of increasing storage capacity quickly and easily using existing technologies. Here, one code of choice is a phased burst error correcting code, since the errors which occur will be burst-like in nature and occurring at one of many predefined intervals.

Properties of a family of phased burst error correcting codes, based on the two-dimensional array code, are studied in Chapter 3. This code is not limited to use in multi-valued computer memories and can be useful in many different applications such as modulation and synchronization protection and line noise tolerance; however, the concentration of this chapter will center upon computer memories. In particular, the chapter covers width and height limitations of the array, optimal array sizes, and methods of coding for correction of approximate errors (error vectors which have components of small scalar values).

Dense storage is useful for storage of images which can routinely occupy 64Kbytes or more for each image. However, much of the information provided by the image is redundantly stored; removing these redundancies can reduce storage requirements and therefore increase storage capacity and access speed. Such compression strategies, however, must themselves be fast to avoid negating the benefits of increased access speed. Furthermore, since no prior knowledge of the images to be entered into the system is available, the compression algorithm must be adaptive. For the algorithm to be competitive, image quality and compression rate must be comparable to existing algorithms.

All of these requirements are met by the locally adaptive vector quantization (LAVQ) algorithm and its improvements presented in Chapter 4. The algorithm is improved through bit stripping, entropy coding, difference coding, nonlinear quantization, high speed search methods, image scanning techniques, filtering, and interpolation. Combined, these improvements give the ba-

sic LAVQ algorithm rate-distortion performance comparable to the basic Linde-Buzo-Gray (LBG) algorithm at a fraction of the computation time.

This work, therefore, covers three topics which are of concern for those dealing with storage of data. It covers the reliability of the physical storage device, means of correcting errors in data, and fast reduction of space requirements for image storage. The medium for which these tools are most applicable is silicon memory; however, the possible applications are broader than those suggested here.

CHAPTER 2
THE RELIABILITY OF SEMICONDUCTOR RANDOM ACCESS MEMORIES WITH ON-CHIP ERROR CORRECTION CODING

2.1. Introduction

Reliability of semiconductor memory is of interest to a number of fields as a result of its proliferation in an assortment of applications. These applications range from high-capacity workstations or mainframes with millions of words of memory to military, industrial, and space-based systems subject to harsh environments. The former requires high reliability of a large number of components in a benign environment; the latter requires high reliability from a smaller number of components in a harsher operating environment. Evaluating memory reliability is necessary in the design of these systems as a part of the study of reliability of the entire system and compliance with design criteria.

As a means of increasing reliability of memory, error correction coding can be applied. These codes can negate the effect of errors or failures by correcting for them on-the-fly, but at the cost of added bits needed for coding check bits. This reduces the effective capacity of the memory chips and is therefore best kept at a minimum; furthermore, since faults occur as a function of chip area, having more cells causes higher error rates. However, more check bits are needed for correcting larger amounts of error. Thus, there is a trade-off between fault tolerance and capacity. Most applications use single error correcting—double error detecting (SEC-DED) codes: More powerful codes tend to have rates which are much lower than the SEC-DED codes and therefore cancel the advantage of having higher density chips.

A typical semiconductor random access memory (RAM) chip is composed of a two-dimensional array of memory cells, organized to provide a single or multiple number of bits as output. Most chips are organized as single bit output ($N \times 1$), 4-bit "nibble" output, or 8-bit "byte" output. All chips are organized internally into a number of two-dimensional blocks with word lines along the rows and bit lines along the columns. When a bit in the block is accessed, the correspond-

ing word line is activated, allowing the entire row of memory cells to be read by the bit lines. One of these bits is then chosen by the column selection circuitry, and that bit is then outputted. (See Figure 1.) The outputs of one or more of these blocks, depending on the output width of the chip, are then chosen.

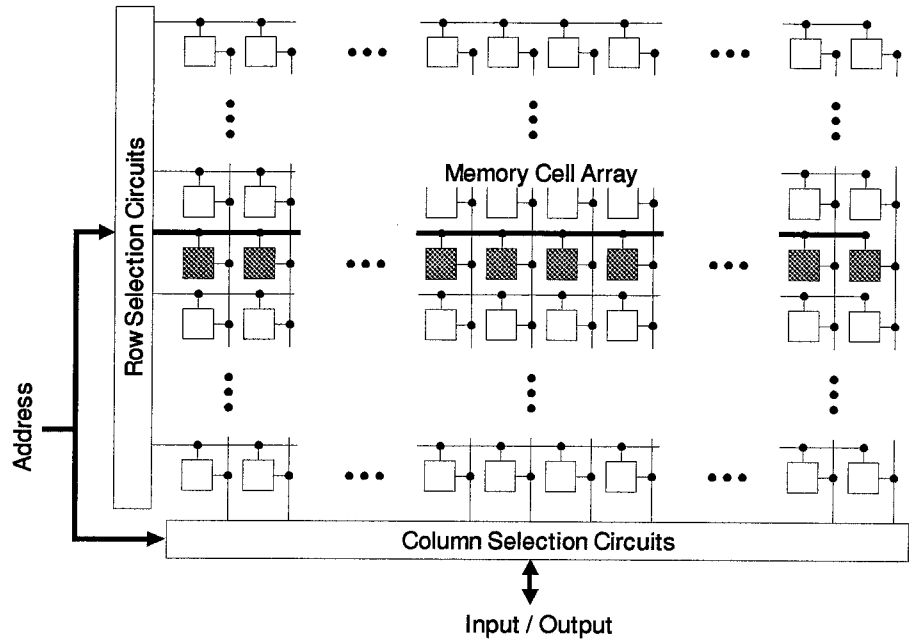

Figure 1: Random access memory structure. Figure shows one word line chosen, allowing access to entire row.

Memory chips are organized onto boards to create byte-wide or word-wide (16, 32, or 64 bits) memory systems. Typically, for SIMM-type (single in-line memory module) units, eight single bit output chips are aligned to provide an 8-bit byte output as shown in Figure 2. The bits from the same addresses on each chip compose a byte; thus, each chip provides one bit of the byte. On a larger system using single bit output chips, the board may be composed of many rows of chips. Figure 2 also shows a case where rows of $N \times 1$ chips are used to form a multi-page memory board. Often, in a large multi-page memory, the rows of the memory are encoded with an (n, k) error correcting code (information chips are shown white; parity chips are shown shaded). This is an example of board-level error correction coding [1,2]. Systems of this form are the most common; they are based on $N \times 1$ memory chips and most often use Hamming codes. The code on the board level is applied to the output byte or word and is therefore capable of correcting a single error in the word. Other techniques are used when the chips themselves have nibble-wide or byte-wide outputs [3].

Memory chips are subject to several types of failure modes. The two main classes of errors

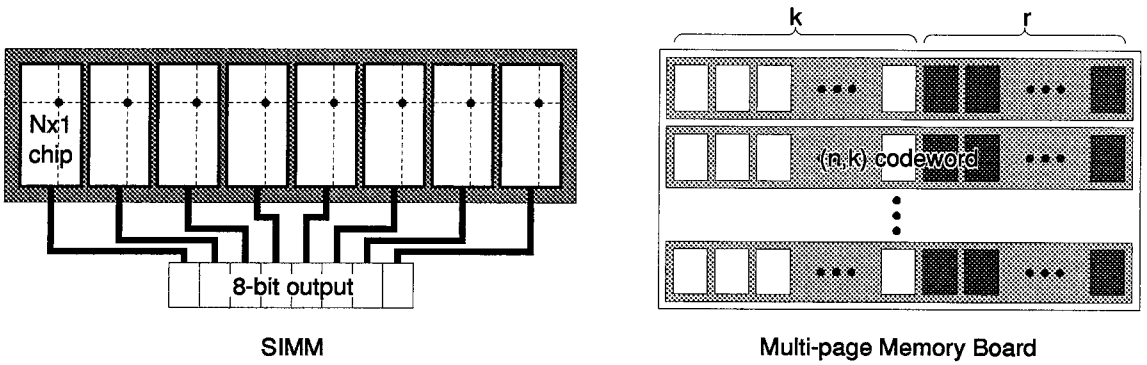

Figure 2: Board-level memory organization. $N \times 1$ memory chips used. SIMM module and multi-page board with ECC are shown. Each chip contributes one bit to the output byte or error correction codeword.

are hard and soft errors. Soft errors, induced most commonly by alpha particle radiation and noise, can affect the content of a cell temporarily by upsetting the charge stored in the cell; this occurs for both static and dynamic memory cell types. The effect is not permanent since no physical damage is done to the chip [4]. As memory cells decrease in size, they individually become more susceptible to this type of damage; furthermore, one alpha particle can affect more than one cell, resulting in clusters of errors [5], as shown in Figure 3.

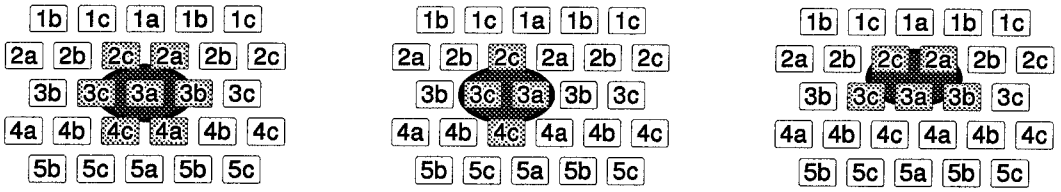

Figure 3: Cluster error patterns. Error patterns created by alpha particle strikes (soft errors). Affected cells are darkened. Cells are marked with numbers for rows and letters for codewords. In this case interleave depth is 3 (three codewords per row, shown as *a*, *b*, and *c*).

There are five common types of hard error or hardware failure. The most common type by far is a single cell hard failure, where a defect occurs in a single cell, making it no longer able to reliably hold data. There are also other less common failure modes. Failures of row selection circuitry (row failure) and column selection circuitry (column failure) cause an entire row or column, respectively, to be unreliable. Shorting of row select and column sense lines (row-column failure) can cause a row and a column to fail simultaneously. Finally, the entire chip can fail if a chip selection or power circuit fails [6]. These types are shown in Figure 4. Other techniques such as row and column replacement are used to increase the reliability of chips at the time of manufacture [7] and

may be better suited to deal with row, column, and row-column failures; these techniques are not addressed here. Also, failures of the addressing mechanisms are not dealt with explicitly, since these are prone to cause multiple-row or multiple-column failures [8].

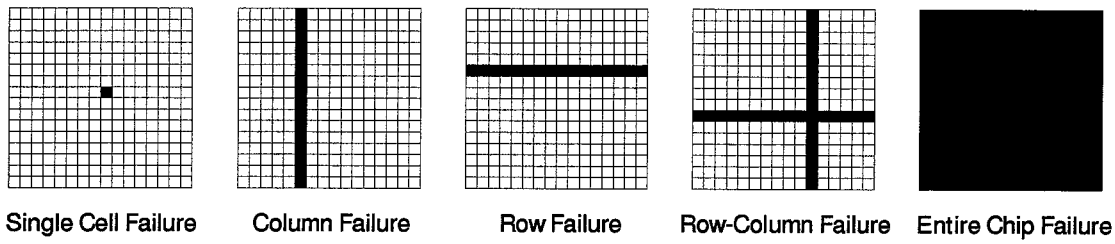

Figure 4: Some hard error types.

Additional hard error types may occur from layout strategies used on large chips. For example, long selection lines cause the chip to be slow and error-prone due to the large capacitance these lines have in relation to the cell capacitance or driving ability. Thus, large memory chips are broken into blocks, with each block having its own selection circuitry. There may be as few as one or as many as 128 blocks of this type; large chips may have more. This makes possible single and multiple block failures and full and partial row and column failures in addition to single cell, row-column, and entire chip failures as shown in Figure 5. The number of types of errors increases as more blocks are employed since there are more failure modes possible.

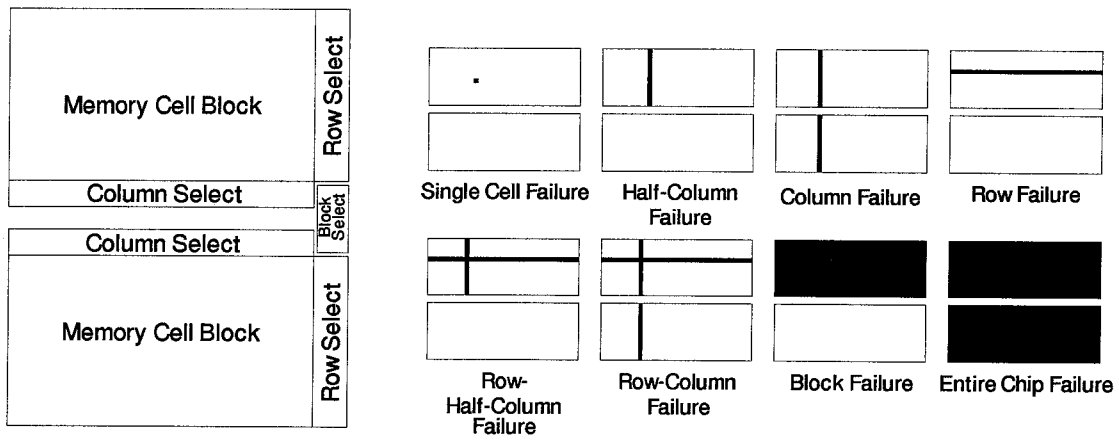

Figure 5: Two-block ram chip. Shown are some associated types of hard errors.

As memory capacity per chip increases, the effect of the above errors becomes too great for board level coding to handle. In particular soft errors can drastically reduce the effective mean time

to failure of an individual chip. It is thus natural to move to on-chip ECC (error correction coding) in order to get higher chip reliability. In addition, it may be necessary to combine chip and board coding to maintain the required reliability [9]. The most natural architecture for on-chip ECC is to place a single error correcting code along each row of the memory. These are most typically SEC (single error correcting) Hamming codes which are shortened to have an information length of some integer power of two. Thus, when a row is accessed, the entire codeword can be simultaneously read, then corrected if necessary, prior to outputting the single bit required by the column selection. If no more than a single error occurs along each row, the memory will remain functional and will reliably hold data. Furthermore, since soft errors can be removed by reading, correcting, and rewriting the data at regular intervals (soft error scrubbing), repeated soft errors on a codeword will not cause memory failure unless more than one error occurs before the memory can be scrubbed. Whether or not more than one bit is output does not conceptually change the function of the code.

Soft errors, however, can affect more than one cell at a time (*i.e.* they cause clusters of errors) as shown in Figure 3, and therefore there must be some means to handle clustered errors. One way is to use burst error correcting codes, but these tend to have higher redundancy and be more complex, resulting in more difficult and time-consuming encoding and decoding, than Hamming codes. These are not desirable characteristics. Instead, this effect is most commonly minimized by employing good layout format and spatially interleaved cells so that cells which belong to the same codeword are spaced apart. This allows the use of single error correcting codes but requires that each row employ more than one codeword [5,10]. Note also that by placing more than one codeword for each row and modeling alpha particles as induced clusters of errors, there arises the possibility that two soft error strikes, even if not in the same codeword, can cause failure. A simple model assumes that each alpha particle effectively affects only a single cell and that two alpha particles must be in the same row to affect the same codeword; this assumption will be used here.

Hard errors, in contrast to soft errors, cannot be removed, only corrected, by error correction coding. Thus, one single cell hard error in a codeword will put the memory in a state where the next error of any type in that codeword will cause memory failure. Likewise, one column failure in the chip will also cause the memory to fail with the next error of any type in any codeword in the same block since this error causes a single error in every codeword in the memory block. Other types of hard errors will overwhelm the error correcting code and cause memory failure with their first occurrence. Note also that placing more than one codeword for each row creates the possibility that multiple single cell or column hard errors may not cause failure; these modes are complex and

will not be covered here. (These effects will be treated by considering interleaved codeword groups as independent blocks of cells.)

Several memory chips which employ on-chip error correcting codes have been fabricated. A production chip fabricated by Micron Technologies employs a (12,8) Hamming code, interleave depth 4, on a 256Kbit × 1 RAM [10]. Asakura, *et al.*, have employed a (40,32) SEC-DED modified Hamming code on a 1Mbit cache DRAM [11]. Horiguchi, *et al.*, have also constructed a RAM with coding, although this is a multi-level, 4 bits/cell 4Mbit DRAM [5]. The code is a (131,128) cyclic SEC 16-ary code with interleave depth 8 to correct single cell errors, the equivalent to four bit errors. Chiueh, Goodman, and Sayano [12] have constructed a 2K × 1 static RAM chip with a double error correcting linear sum code proposed by Fuja, Heegard, and Goodman [13]. Most codes used in practice have been simple binary Hamming type codes, in order to reduce complexity of the on-chip decoder.

Most previous MTTF calculations were based entirely or in part on the binomial distribution [1,14-17]. The results tended to be complex, and some were simplified by taking the Poisson approximation to the binomial distribution at one point or another. Using the Poisson distribution from the start, first done in [18], provides a far less complex yet accurate result. The Poisson distribution will be used in this analysis. Furthermore, row-column type hard errors were first accurately modeled in [19]; previous papers did not address this failure mode. Now the effects of soft error scrubbing in semiconductor random access memories coded with on-chip single error correcting codes and subject to both hard and soft errors will be examined. Such a model can be used by system designers to assess the mean time to failure improvement bought by using coding.

Subsequent sections will develop this model. Section 2.2 will deal with the Poisson failure model which will be used in subsequent sections as the mathematical representation of the chip. Section 2.3 then develops the model for both hard and soft single cell failures with soft error scrubbing. Characteristics of the model will be explored for various limits of parameters. Section 2.4 then expands the model to include multiple-cell hard failures (such as row, column, row-column, and entire chip failures). In addition, the case where chips are composed of multiple blocks of memory cell arrays will be addressed here. Section 2.5 presents experimental results from computer simulations, which help confirm the accuracy of the model for varying parameter values and for values typical of actual devices.

2.2. Poisson Failure Model

An accurate model of the mean time to failure of a semiconductor memory with a single error correcting code placed along each row is necessary for this analysis. However, the model must also be computationally simple enough to ease its determination. The mathematical model used here will be the Poisson distribution for determination of probability of error in a given codeword. The accuracy of this model has been justified in previous work [18-21]. A short review of the model is provided below.

The Poisson distribution gives the probability that there are no events (failures) in time t as

$$P_0(t) = e^{-\lambda t}$$

where λ is the mean event arrival rate, and the probability that there is exactly one event (failure) in time t as

$$P_1(t) = \lambda t e^{-\lambda t}$$

The row reliability function is the probability that the row has not failed in time t; for a codeword protected by a single error correcting code and error arrival rate per row λ, this becomes [21]

$$\begin{aligned} R(t) &= P(0 \text{ or } 1 \text{ error}) \\ &= (1 + \lambda t)e^{-\lambda t} \end{aligned}$$

By assuming independence among the rows, the chip reliability function, *i.e.* the probability that the entire chip has not yet failed by time t, is given by

$$R_{chip}(t) = R^M(t)$$

where there are M codewords in a chip.

The mean time to failure is

$$MTTF = \int_0^\infty R^M(t)dt$$

as shown in [21]. Modeling such a function may be time-intensive, so for simulation purposes the mean events to failure (METF) is used. This is the average number of events which must occur before memory failure occurs; if this number is large, then by Wald's identity (and Little's Law, which

states that, for Poisson processes, the mean waiting time—in this case, for failure—is the product of the mean number in the system and the inverse of the mean arrival rate) the approximation

$$MTTF = \frac{1}{\lambda} METF \tag{1}$$

can be used, because $1/\lambda$ represents the mean arrival rate of events (mean rate of errors) in the Poisson distribution [19].

2.3. Analysis of Single Cell Failures

2.3.1. The MTTF Calculation

Initially the case of only single cell hard and soft errors attacking the memory will be discussed. Furthermore, soft errors are confined to affecting only one cell. The symbols used here are shown below.

λ_h = single cell hard error arrival rate per cell
λ_s = single cell soft error arrival rate per cell
λ_{sc} = $\lambda_h + \lambda_s$ = total single cell error arrival rate per cell
t_s = soft error scrubbing period
n = number of bits in a codeword
M = number of codewords on a chip

Soft error scrubbing will be conducted. This means that at periodic intervals t_s, the entire chip will be subject to internal error correction where all codewords are read, corrected, and rewritten. This cannot remove hard errors, but it can remove soft errors, so long as no more than one error occurs in a single codeword. Note that no restriction is placed here on the number of codewords in each word line (in each row). Therefore, the word line may contain more than one codeword, such as when spatial interleaving is done. Thus, the number of codewords does not necessarily equal the number of rows on the chip. Also, there may be multiple blocks on the chip in a manner similar to that shown in Figure 5.

The following two assumptions will be made.

Assumption I: The scrub cycle must be small compared to the mean time to failure: $t_s \ll MTTF$

Since the scrub cycle (which, in dynamic RAMs, occurs in conjunction with the refresh cycle) is rarely longer than 100 seconds, a system with a mean time to failure which is not far greater than this is not reliable enough for use.

Assumption II: The hard error rate must be small compared to the soft error rate: $\lambda_h \ll \lambda_s$

Hard errors cannot be removed by scrubbing and therefore accumulate; soft errors, which

occur more frequently [6], can be removed. If hard errors occur with greater frequency than soft errors, then soft error scrubbing is useless ($\lambda_h \gg \lambda_s$ and λ_s is small enough so that scrubbing need not be done) or the chip is unreliable for use ($\lambda_h \gg \lambda_s$ and λ_h is too large for the chip to be used reliably). These assumptions are true for all practical memory chips; a rigorous argument justifying these assumptions in the model, along with a more accurate but complex analysis, is presented in the appendix to this chapter. No mention is made of the reading and writing mechanism here because if scrubbing is done, each word is accessed within the period of one scrub cycle. If failure is declared only after a word is accessed and found to be in error, since Assumption I holds true, the increased time before failure is declared is negligible compared to the case where failure is declared immediately following two failures in any codeword.

The situation is therefore this: While there are no hard errors in a codeword, the memory does not fail so long as no more than one soft error occurs in each time interval t_s in each codeword. If a hard error does occur, then no further errors can be tolerated in that same codeword. The two cases, when a hard error has not occurred and exactly one hard error has occurred by time t, will be treated separately below.

The probability of codeword success at time t with no hard errors occurring is the probability that no hard errors have occurred and that only zero or one soft error has occurred in any time segment t_s from time 0 to t. Thus,

$$\begin{aligned} R_0(t) &= P(\text{no hard errors})P(0 \text{ or } 1 \text{ soft error}) \\ &= [e^{-\lambda_h nt}][e^{-\lambda_s nt_s} + \lambda_s nt_s e^{-\lambda_s nt_s}]^{t/t_s} \\ &= e^{-\lambda_{sc} nt}(1 + \lambda_s nt_s)^{t/t_s} \end{aligned} \tag{2}$$

The probability of codeword success at time t with the occurrence of exactly one hard error before time t is more complex. First, assume that the hard error had occurred at time τ. Therefore, the probability of success is the joint probability that there were no codeword failures given no hard failures have occurred up to time τ, that exactly one hard error occurred between time τ and $\tau + d\tau$, and that neither soft nor hard errors occurred from time τ to t. This is then integrated over τ from 0 to t to consider all possible times that the hard error could have occurred.

$$\begin{aligned} R_1(t) &= \int_0^t R_0(\tau)P(1 \text{ hard error in } d\tau)P(\text{no errors in } \tau \text{ to } t) \\ &= \int_0^t R_0(\tau)[\lambda_h n d\tau][e^{-\lambda_{sc} n(t-\tau)}] \\ &= e^{-\lambda_{sc} nt}\lambda_h n \int_0^t (1 + \lambda_s nt_s)^{\tau/t_s} d\tau \end{aligned}$$

$$= \frac{e^{-\lambda_{sc} n t} \lambda_h n t_s}{\ln(1+\lambda_s n t_s)} [(1+\lambda_s n t_s)^{t/t_s} - 1] \tag{3}$$

Therefore, the reliability function for each codeword is now given by combining equations (2) and (3).

$$\begin{aligned} R(t) &= R_0(t) + R_1(t) \\ &= e^{-\lambda_{sc} n t}(1+\lambda_s n t_s)^{t/t_s} + \frac{e^{-\lambda_{sc} n t} \lambda_h n t_s}{\ln(1+\lambda_s n t_s)} [(1+\lambda_s n t_s)^{t/t_s} - 1] \end{aligned} \tag{4}$$

The mean time to failure (MTTF) for a chip with M codewords is then

$$\begin{aligned} MTTF &= \int_0^\infty R^M(t) dt \\ &= \int_0^\infty e^{-\lambda_{sc} n M t} \left[\exp[\frac{t}{t_s} \ln(1+\lambda_s n t_s)] \left(1 + \frac{t_s \lambda_h n}{\ln(1+\lambda_s n t_s)}\right) - \frac{t_s \lambda_h n}{\ln(1+\lambda_s n t_s)} \right]^M dt \end{aligned} \tag{5}$$

Equation (5) is of the form

$$MTTF = \int_0^\infty e^{c_0 t} \left[e^{c_1 t} c_2 + c_3 \right]^M dt \tag{6}$$

where

$$\begin{aligned} c_0 &= -\lambda_{sc} n M \\ c_1 &= \frac{1}{t_s} \ln(1+\lambda_s n t_s) \\ c_2 &= 1 + \frac{t_s \lambda_h n}{\ln(1+\lambda_s n t_s)} \\ c_3 &= -\frac{t_s \lambda_h n}{\ln(1+\lambda_s n t_s)} \end{aligned}$$

Equation (6) can be evaluated by making the substitution $y = e^{c_1 t}$.

$$\begin{aligned} MTTF &= \frac{1}{c_1} \int_1^\infty y^{(c_0/c_1)-1} [y c_2 + c_3]^M dy \\ &= \sum_{i=0}^{M} \binom{M}{i} \frac{c_2^i c_3^{M-i} y^{(c_0/c_1)+i}}{c_0 + i c_1} \Bigg|_1^\infty \end{aligned}$$

which does not converge unless

$$y^{(c_0/c_1)+i} \to 0 \; as \; y \to \infty$$

which means that $c_0 + c_1 i < 0$. Since $c_0 < 0$ and $c_1 > 0$, the condition may be violated when i is as large as possible, *i.e.* when $i = M$. But $\ln(1+x) < x$, so

$$\begin{aligned} c_0 + c_1 M &= -\lambda_{sc} n M + \frac{M}{t_s} \ln(1+\lambda_s n t_s) \\ &< -\lambda_{sc} n M + \lambda_s n M \\ &= -\lambda_h n M \\ &< 0 \end{aligned}$$

Also, the denominator cannot be zero for any i, $0 \leq i \leq M$:

$$\lambda_{sc} nM \neq \frac{i}{t_s}(1 + t_s n \lambda_s) \quad \text{for} \quad 0 \leq i \leq M \tag{7}$$

Using $\ln(1+x) < x$ again,

$$\lambda_s n i > i \frac{\ln(1 + \lambda_s n t_s)}{t_s}$$

Since $\lambda_{sc} > \lambda_s$, and since $i \leq M$, the condition set forth in equation (7) is satisfied.

The equation for mean time to failure becomes

$$\begin{aligned} MTTF &= -\sum_{i=0}^{M} \binom{M}{i} \frac{c_2^i c_3^{M-i}}{c_0 + c_1 i} \\ &= \sum_{i=0}^{M} \binom{M}{i} \frac{\left(1 + \frac{t_s \lambda_h n}{\ln(1+\lambda_s n t_s)}\right)^i \left(-\frac{t_s \lambda_h n}{\ln(1+\lambda_s n t_s)}\right)^{M-i}}{\lambda_{sc} nM - \frac{i}{t_s} \ln(1 + \lambda_s n t_s)} \end{aligned} \tag{8}$$

Alternatively, by making the substitutions

$$\begin{aligned} y &= \lambda_s n t_s \\ z &= \frac{\lambda_h}{\lambda_s} \end{aligned}$$

the mean time to failure can be represented in a clearer form:

$$MTTF = \frac{1}{\lambda_{sc} nM} \sum_{i=0}^{M} \binom{M}{i} \frac{\left(1 + \frac{yz}{\ln(1+y)}\right)^i \left(-\frac{yz}{\ln(1+y)}\right)^{M-i}}{1 - \frac{i}{M} \frac{\ln(1+y)}{y(z+1)}} \tag{9}$$

The coding gain, which is the advantage of the mean time to failure over the uncoded case, is therefore

$$\begin{aligned} CG &= \frac{k}{n} \frac{MTTF_{coded}}{MTTF_{uncoded}} \\ &= \frac{k}{n} \sum_{i=0}^{M} \binom{M}{i} \frac{\left(1 + \frac{yz}{\ln(1+y)}\right)^i \left(-\frac{yz}{\ln(1+y)}\right)^{M-i}}{1 - \frac{i}{M} \frac{\ln(1+y)}{y(z+1)}} \end{aligned} \tag{10}$$

where (n, k) codes are used.

The mean time to failure, therefore, is effectively a function of three parameters, y, z, and M. The parameter y represents the relation between the scrub interval and the soft error rate; it is the mean number of soft errors which strike each codeword in a scrub period. The parameter z is the ratio of soft to hard error rates. Note that if λ_s, λ_h, and t_s are adjusted such that y and z are kept constant, then the coding gain remains unchanged: The parameter y can be kept constant by maintaining a constant $\lambda_s t_s$; the parameter z can be kept constant by altering λ_s and λ_h proportionally. Such a dependence implies that time is being scaled.

Note that equation (9) appears similar to a binomial expansion of order M, a consequence of expanding and integrating $R^M(t)$. Also, if y and z are small, then the denominator term will be small, and therefore each term of the sum will be large. This means that if the hard error rate is small compared to the soft error rate, and if soft errors are scrubbed out before they are allowed to accumulate, MTTF will be large, a conclusion which confirms intuition.

Subsequent subsections will explore various special cases of equation (9). Specifically, of interest are the limit of very fast scrubbing ($t_s \to 0$) and the limit of no hard errors ($\lambda_h \to 0$). In addition, an analysis will be made for when Assumptions I and II are violated. These cases will be examined for three reasons. First, these cases, as will be shown, will provide a connection between the results of previous work (most notably, [19-21]). Second, when the assumptions are violated, the model should break down predictably, based on intuition, simulation, and previous work. Third, the special cases are examined for completeness.

2.3.2. MTTF With Fast Scrubbing

If the interval between each soft error scrub is allowed to decrease to zero, that is, scrubbing is done continuously, there is a maximum mean time to failure which cannot be exceeded. In the limit $t_s \to 0$, or equivalently $y \to 0$, equation (9) becomes

$$\begin{aligned} MTTF &= \frac{1}{\lambda_{sc} nM} \sum_{i=0}^{M} \binom{M}{i} \frac{(1+z)^i (-z)^{M-i}}{1 - \frac{i}{M}\frac{1}{z+1}} \\ &= \frac{1}{\lambda_{sc} nM} \sum_{i=0}^{M} \binom{M}{i} \frac{\left(1+\frac{\lambda_h}{\lambda_s}\right)^i \left(-\frac{\lambda_h}{\lambda_s}\right)^{M-i}}{1 - \frac{i}{M}\frac{1}{\frac{\lambda_h}{\lambda_s}+1}} \end{aligned} \tag{11}$$

The coding gain is therefore

$$CG = \frac{k}{n} \sum_{i=0}^{M} \binom{M}{i} \frac{\left(1+\frac{\lambda_h}{\lambda_s}\right)^i \left(-\frac{\lambda_h}{\lambda_s}\right)^{M-i}}{1 - \frac{i}{M}\frac{1}{\frac{\lambda_h}{\lambda_s}+1}} \tag{12}$$

Equations (11) and (12) are simpler expressions to evaluate than equations (9) and (10), and they show the dependence of MTTF on λ_h/λ_s. Thus, if $t_s \to 0$, the ratio between the hard and soft error rates becomes important.

Note that if λ_h/λ_s is extremely small, the last term of the summation in equation (11) is the dominant term:

$$MTTF \approx \frac{1}{\lambda_{sc} nM} \frac{\left(1+\frac{\lambda_h}{\lambda_s}\right)^M}{\frac{\lambda_h}{\lambda_h+\lambda_s}}$$

$$\approx \frac{1}{\lambda_h nM} \tag{13}$$

$$CG \approx \frac{k}{n}\left(1+\frac{\lambda_s}{\lambda_h}\right) \tag{14}$$

This means that if soft errors occur much more often compared to hard errors but slow enough to be scrubbed out before more than one can accumulate in a single codeword, then the MTTF approaches that of the unprotected chip with only hard errors. This is reasonable, since when a hard error occurs, a soft error has a high probability of occurring relatively immediately after it in the same codeword, thus causing a failure. However, since the scrub interval is short enough, soft errors without a hard error in the same codeword never accumulate fast enough to cause failure. Thus, the model behaves as expected for $t_s \to 0$. Note that this provides an upper bound on performance: Given an error arrival rate of λ_s and λ_h, the MTTF cannot exceed that given in equation (11) even with extremely fast soft error scrubbing.

2.3.3. Soft Errors as the Dominant Error Type

For the case of only soft errors occurring, *i.e.* $\lambda_h \to 0$ ($z \to 0$) the chip can fail only if two or more soft errors occur in the same codeword and in the same scrub interval. Therefore, the mean time to failure will be

$$\begin{aligned} MTTF &= \int_0^\infty R_0^M(t)dt \\ &= \int_0^\infty e^{-\lambda_s nMt}(1+\lambda_s nt_s)^{Mt/t_s}dt \\ &= \int_0^\infty \exp\left[-\lambda_s nMt + \frac{Mt}{t_s}\ln(1+\lambda_s nt_s)\right]dt \\ &= \left[\lambda_s nM - \frac{M}{t_s}\ln(1+\lambda_s nt_s)\right]^{-1} \end{aligned} \tag{15}$$

If the model is accurate, then the same result should be achieved when equation (9) is used. By taking $z = 0$ in equation (9),

$$\begin{aligned} MTTF &= \frac{1}{\lambda_{sc} nM}\sum_{i=0}^{M}\binom{M}{i}\frac{1^i 0^{M-i}}{1-\frac{i}{M}\frac{\ln(1+y)}{y}} \\ &= \frac{1}{\lambda_s nM}\left[1-\frac{\ln(1+\lambda_s nt_s)}{\lambda_s nt_s}\right]^{-1} \end{aligned} \tag{16}$$

The coding gain is

$$CG = \frac{k}{n}\left[1-\frac{\ln(1+\lambda_s nt_s)}{\lambda_s nt_s}\right]^{-1} \tag{17}$$

Equation (16) is the same as equation (15). Note that if $\lambda_s nt_s$ is held constant, then the coding gain remains constant, as expected: Time scaling should not affect the coding gain. Also note that

forcing $t_s \rightarrow 0$ now yields $MTTF \rightarrow \infty$, as there are no hard errors to build up on the chip to cause two or more errors in a codeword during any single scrub cycle. The asymptotic behavior of this case can be determined by taking the Taylor series of $\ln(1+y)$:

$$\begin{aligned} MTTF &= \frac{1}{\lambda_s nM}\left[1 - \frac{\ln(1+y)}{y}\right]^{-1} \\ &= \frac{1}{\lambda_s nM}\left[1 - \frac{y - \frac{1}{2}y^2 + \frac{1}{3}y^3 \cdots}{y}\right]^{-1} \\ &\approx \frac{1}{\lambda_s nM}\left[\frac{2}{y}\right] \qquad (18) \\ CG &\approx \frac{k}{n}\left[\frac{2}{y}\right] \qquad (19) \end{aligned}$$

for small y. Thus, the MTTF and the scrub interval are inversely related.

This analysis has given the maximum possible MTTF given t_s and λ_s for a highly reliable chip; the upper bound on performance for a given t_s and λ_s is therefore easily found using equation (15). Furthermore, this special case result is consistent with the results in [22], showing that this model is a more comprehensive representation of the memory chip.

2.3.4. Very Slow Soft Scrubbing

For the case of very slow soft error scrubbing, Assumption I is violated, and equation (9) does not apply. The analysis must be reconducted to develop a new representation. Violation of Assumption I without violation of Assumption II, for typical values of the other parameters, corresponds to having a very long scrub period. This case is not realistic: Having a long scrub cycle is useless, since if many errors are allowed to occur in a single codeword before soft error scrubbing is conducted the error correcting power of the code is more likely to be overwhelmed. Though this case is not expected to occur in reality, it will be studied to determine how chip performance can be expected to degrade. In this case, instead of the continuous scrub period analysis conducted thus far, a discrete scrub period analysis must be made. In previous sections, since Assumption I assured the passing of many scrub periods before a chip failure was expected to occur, a continuous scrub period analysis, as done in equation (2), was possible. The equivalent analysis using discrete scrub period intervals is much more complex. (This analysis is covered briefly in the appendix, where its complexity becomes evident.)

Instead, an analysis of the special case $t_s \rightarrow \infty$ will be conducted. This will provide a lower bound and an intuitive feel for the behavior of the model. The model must be reconstructed from the reliability function.

$$R_0(t) = e^{-\lambda_{sc} nt}[1 + \lambda_s nt] \qquad (20)$$

and

$$R_1(t) = \lambda_h n t e^{-\lambda_{sc} n t} \tag{21}$$

are now used to determine the reliability function instead of equations (2) and (3). $R_0(t)$ represents the probability that the codeword has yet to fail given that no hard errors have occurred. In this case, $R_0(t)$ is the probability that zero or one soft error and no hard errors have occurred. $R_1(t)$ is the probability that the codeword has yet to fail given that exactly one hard error has occurred. Thus, the new reliability function is the sum of equations (20) and (21)

$$R(t) = e^{-\lambda_{sc} n t}[1 + \lambda_{sc} n t] \tag{22}$$

Therefore, the mean time to failure becomes

$$\begin{aligned} MTTF &= \int_0^\infty e^{-\lambda_{sc} n M t}[1 + \lambda_{sc} n t]^M dt \\ &= \int_0^\infty e^{-\lambda_{sc} n M t} \sum_{i=0}^{M} \binom{M}{i} (\lambda_{sc} n t)^i dt \\ &= \frac{1}{\lambda_{sc} n M} \sum_{i=0}^{M} \binom{M}{i} \frac{i!}{M^i} \end{aligned} \tag{23}$$

There is a factor of

$$B(M) = \sum_{i=0}^{M} \binom{M}{i} \frac{i!}{M^i} \tag{24}$$

in the equation which is only a function of the number of codewords in the chip. Note that the MTTF is greater than the uncoded mean time to failure by the factor $B(M)$. This factor $B(M)$ was analyzed as an extension of the Birthday Surprise problem in [19,20] and has been shown to be, for large M,

$$B(M) = \sqrt{\frac{\pi M}{2}} + \frac{2}{3} + O(M^{-1/2}) \tag{25}$$

Note that equation (25) is equivalent to the solution found via a different analysis in [23].

The coding gain, therefore, is

$$CG = \frac{k}{n} B(M) \tag{26}$$

and is no longer a function of the error arrival rates. This is reasonable, as the lack of soft error scrubbing removes the time dependence. Only the error correcting code's correctional power is important, as reflected by $B(M)$. Some values of $B(M)$ are listed in Table 1. Note that for typical memory chips of the order of 1Mbit and larger the error between the actual value of $B(M)$ and the value found by equation (25) is much less than 1%.

M	B(M)	% error	M	B(M)	% error
1	2.00	-4.00	2^{13}	114.1	-1.01×10^{-3}
2	2.50	-2.44	2^{14}	161.1	-5.05×10^{-4}
2^2	3.22	-1.41	2^{15}	227.5	-2.53×10^{-4}
2^3	4.25	-7.87×10^{-1}	2^{16}	321.5	-1.27×10^{-4}
2^4	5.70	-4.27×10^{-1}	2^{17}	454.4	-6.34×10^{-5}
2^5	7.77	-2.26×10^{-1}	2^{18}	642.4	-3.17×10^{-5}
2^6	10.71	-1.18×10^{-1}	2^{19}	908.2	-1.59×10^{-5}
2^7	14.86	-6.06×10^{-2}	2^{20}	1284.1	-7.93×10^{-6}
2^8	20.73	-3.09×10^{-2}	2^{21}	1815.7	-3.96×10^{-6}
2^9	29.03	-1.57×10^{-2}	2^{22}	2567.5	-1.98×10^{-6}
2^{10}	40.78	-7.93×10^{-3}	2^{23}	3630.7	-9.87×10^{-7}
2^{11}	57.39	-4.00×10^{-3}	2^{24}	5134.2	-4.90×10^{-7}
2^{12}	80.88	-2.01×10^{-3}			

Table 1: Some values of factor $B(M)$. Also tabulated are errors between approximation (25) and the exact values.

2.3.5. Hard Error Dominance

For the case where hard errors dominate, Assumption II does not hold. A new model will be constructed in this section for this case. Violation of Assumption II corresponds to having hard errors as the dominant error type. This is not a realistic situation: Hard errors are much less common than soft errors [6], and if they were not, either the chip itself is very unreliable and therefore not suitable for use (λ_h is too large) or soft error scrubbing is not needed (λ_s is very small). In both cases, soft error scrubbing is useless because in the former, most of the errors cannot be removed by scrubbing, and in the latter, there are no soft errors to be scrubbed. However, for completeness this case is considered.

For simplicity and for an intuitive feel of the behavior of the chip for very large λ_h/λ_s, the limiting case $\lambda_h/\lambda_s \to \infty$ will be examined. The reliability function can be reconstructed from the basic Poisson model, or it can be adapted from equation (4) as shown below; both give equivalent models. This occurs because if the equations given in the appendix, the correct complete analysis, were carried out, the range of t where $R(t)$ is significant will be such that $\lfloor t/t_s \rfloor < 1$. This is true because if hard errors are more common than soft errors, then they are likely to cause failure by themselves far more quickly than soft errors. Since soft errors are assumed to occur at a rate slow enough to be scrubbed out, any arrival of errors much faster than this implies Assumption II, that the scrub interval is much less than the MTTF, is violated. Completing this analysis yields equation (4) as the reliability function.

Using the substitution $z = \lambda_h/\lambda_s$, the reliability function given by equation (4) becomes

$$R(t) = e^{-\lambda_s(1+z)nt}(1+\frac{\lambda_h nt_s}{z})^{t/t_s} + \frac{e^{-\lambda_s(1+z)nt}\lambda_h nt_s}{\ln(1+\frac{\lambda_h nt_s}{z})}[(1+\frac{\lambda_h nt_s}{z})^{t/t_s} - 1] \tag{27}$$

By letting $z \rightarrow \infty$ and using

$$\ln(1+x) \approx x \; as \; x \rightarrow 0$$

and

$$(1+x)^k \approx 1 + kx \; as \; x \rightarrow 0$$

equation (27) becomes

$$\begin{aligned} R(t) &= e^{-\lambda_s znt} + e^{-\lambda_s znt}\frac{\lambda_h nt_s}{z} + e^{-\lambda_s znt}\lambda_h nt_s \frac{z}{\lambda_h nt_s}\frac{\lambda_h nt}{z} \\ &= e^{-\lambda_h nt} + e^{-\lambda_h nt}\lambda_h nt \\ &= e^{-\lambda_h nt}[1+\lambda_h nt] \end{aligned} \tag{28}$$

Note that equation (28) is the same form as equation (22); therefore

$$MTTF = \frac{1}{\lambda_h nM}\sum_{i=0}^{M}\binom{M}{i}\frac{i!}{M^i} \tag{29}$$

Again, the factor given in equation (24)

$$B(M) = \sum_{i=0}^{M}\binom{M}{i}\frac{i!}{M^i}$$

appears in the equation. The coding gain is again given by equation (26)

$$CG = \frac{k}{n}B(M)$$

Equations (23) and (29) are identical except for the value of λ they employ; equation (23) uses λ_{sc} while equation (29) uses λ_h. The coding gain, however, is identical whenever either Assumption I or II is violated.

2.4. The General Case: Multiple Types of Hard Errors

2.4.1. Multiple Hard Error Types with One Block per Chip

The preceding section dealt with single cell hard and soft errors in an M-codeword memory encoded along the rows with a single error correcting code. Here the situation will be the same,

except that other types of hard errors, namely row failures, column failures, row-column failures, and entire chip failures will also be considered. Analysis will again be done via Poisson distribution. In this analysis, coding will be restricted to one codeword per row for the sake of simplicity, and the chip is assumed to be composed of one block.

Since the memory is encoded with codewords along rows, the entire chip can survive exactly one column failure. However, the subsequent appearance of a single error of any type anywhere in the chip will then cause a failure. Any other type of multiple-cell hard error (row, row-column, and entire chip failures) at any time will cause the chip to fail immediately. These latter types of errors will be designated catastrophic failures.

The symbols used here are shown below.

λ_h = single cell hard error arrival rate per cell
λ_s = single cell soft error arrival rate per cell
$\lambda_{sc} = \lambda_h + \lambda_s$ = total single cell error arrival rate per cell
λ_c = column failure rate per block
λ_r = row failure rate per block
λ_{rc} = row-column failure rate per block
λ_{wc} = whole block failure rate per block
$\lambda_f = \lambda_r + \lambda_{rc} + \lambda_{wc}$ = catastrophic failure rate per block
$\lambda_{mc} = \lambda_r + \lambda_{rc} + \lambda_{wc} + \lambda_c$ = muliple cell error arrival rate per block
$\lambda = \lambda_{mc} + Mn\lambda_{sc}$ = total error arrival rate per block
t_s = soft error scrubbing period
n = number of bits in a codeword
M = number of codewords on a block

The analysis here is for when Assumptions I and II hold. Therefore, the codeword level reliability function remains unchanged from equation (4) in the range where the two assumptions are valid; however, the chip level reliability function is no longer the row reliability function raised to the power of M. The probability of chip success at time t is controlled by the following: (1) There must be no chip level failures (multi-cell hard errors) or codeword level failures which cause the memory to fail; (2) if there is a column failure (a chip level failure) at time τ, there must be no further errors of any type from time τ to t.

$$
\begin{aligned}
R_c(t) &= P(\text{no multicell failures})P(\text{no chip failures from single cell errors}) \\
&\quad + \int_0^t P(\text{no hard errors in 0 to } \tau)P(1 \text{ column failure in } d\tau)P(\text{no errors in } \tau \text{ to } t) \\
&= \left[e^{-\lambda_{mc}t}\right]\left[[R_0(t) + R_1(t)]^M\right] + \int_0^t \left[e^{-\lambda_{mc}\tau} {R_0}^M(\tau)\right]\left[\lambda_c d\tau\right]\left[e^{-\lambda(t-\tau)}\right] \\
&= e^{-\lambda t}\left[e^{c_1 t}c_2 + c_3\right]^M + \lambda_c e^{-\lambda t}\int_0^t e^{c_1 M\tau} d\tau
\end{aligned} \tag{30}
$$

where, as shown previously,

$$c_1 = \frac{1}{t_s}\ln(1 + \lambda_s n t_s)$$

$$c_2 = 1 + \frac{t_s \lambda_h n}{\ln(1 + \lambda_s n t_s)}$$

$$c_3 = -\frac{t_s \lambda_h n}{\ln(1 + \lambda_s n t_s)}$$

Integration of equation (30) yields a chip reliability function of

$$R_c(t) = e^{-\lambda t}\Big[e^{c_1 t} c_2 + c_3\Big]^M + \frac{\lambda_c e^{-\lambda t}}{M c_1}\Big[e^{c_1 M t} - 1\Big] \tag{31}$$

The mean time to failure (MTTF) is now

$$\begin{aligned}
MTTF &= \int_0^\infty R_c(t)dt \\
&= \left[\int_0^\infty e^{-\lambda t}\Big[e^{c_1 t} c_2 + c_3\Big]^M dt\right] + \left[\frac{\lambda_c}{M c_1}\int_0^\infty \Big(\exp[-\lambda t + c_1 M t] - \exp[-\lambda t]\Big)dt\right] \\
&= \left[\sum_{i=0}^{M}\binom{M}{i}\frac{c_2^i c_3^{M-i}}{\lambda - c_1 i}\right] + \left[\frac{\lambda_c}{M c_1}\Big(\frac{1}{\lambda - M c_1} - \frac{1}{\lambda}\Big)\right] \\
&= \left[\sum_{i=0}^{M}\binom{M}{i}\frac{c_2^i c_3^{M-i}}{\lambda - c_1 i}\right] + \frac{\lambda_c}{\lambda}[\lambda - M c_1]^{-1} \\
&= \sum_{i=0}^{M}\binom{M}{i}\frac{\Big(1 + \frac{t_s\lambda_h n}{\ln(1+\lambda_s n t_s)}\Big)^i\Big(-\frac{t_s\lambda_h n}{\ln(1+\lambda_s n t_s)}\Big)^{M-i}}{\lambda - \frac{i}{t_s}\ln(1+\lambda_s n t_s)} + \frac{\lambda_c}{\lambda}\Big[\lambda - \frac{M}{t_s}\ln(1+\lambda_s n t_s)\Big]^{-1}
\end{aligned} \tag{32}$$

The coding gain is

$$\begin{aligned}
CG = \frac{k}{n\lambda}\sum_{i=0}^{M}\binom{M}{i}&\frac{\Big(1 + \frac{t_s\lambda_h n}{\ln(1+\lambda_s n t_s)}\Big)^i\Big(-\frac{t_s\lambda_h n}{\ln(1+\lambda_s n t_s)}\Big)^{M-i}}{\lambda - \frac{i}{t_s}\ln(1+\lambda_s n t_s)} \\
&+ \frac{k\lambda_c}{n\lambda^2}\Big[\lambda - \frac{M}{t_s}\ln(1+\lambda_s n t_s)\Big]^{-1}
\end{aligned} \tag{33}$$

where (n, k) codes are used.

Thus, the mean time to failure in equation (32) is modified from equation (9) by two factors: First, instead of the total single cell error rate $Mn\lambda_{sc}$ in the denominator of the summation, the total error rate λ is used. Since λ is slightly larger than $Mn\lambda_{sc}$, the denominator has increased slightly; thus, each term of the summation in equation (32) is slightly smaller than in equation (9), leading to a smaller overall sum in equation (32) than in equation (9). Second, an additive term which increases the MTTF is present. This second term in equation (32) can be approximated as follows:

$$\begin{aligned}
\frac{\lambda_c}{\lambda}\Big[\lambda - \frac{M}{t_s}\ln(1+\lambda_s n t_s)\Big]^{-1} &\le \frac{\lambda_c}{\lambda}\Big[\lambda - \frac{M}{t_s}\lambda_s n t_s\Big]^{-1} \\
&= \frac{\lambda_c}{\lambda}[\lambda_{mc} + Mn\lambda_h]^{-1} \\
&= \frac{1}{\lambda}\Big[\frac{\lambda_c}{\lambda_{mc} + Mn\lambda_h}\Big]
\end{aligned} \tag{34}$$

The additive term is shown by equation (34) as being small compared to the summation term in equation (33): In typical memory chips, λ_c, the rate of column failure, is less than 10% of the total hardware failure rate [21]. Therefore, the additive term in equation (33) is typically less than one-tenth the mean time to failure for an unprotected chip. A protected chip can be expected to have a much longer MTTF than an unprotected one (*i.e.* $CG \gg 1$), so the additive term is, in most cases, insignificant in comparison with total MTTF.

2.4.2. Multiple Hard Error Types with Multiple Blocks per Chip

If the chip is subdivided into several identical blocks as shown in Figure 5, some assumptions need to be made regarding the structure of the chip and its failure modes. The simplest case to analyze (and the design with the simplest and most dense construction) is when, as shown in [6], row, row-column, and column errors occur across the entire chip, so that the only additional hard error type from the five discussed in the previous section is block failures. (Chips of this type are those with only one set of selection circuitry instead of separate selection circuits for each block.) There can be multiple-block failures, but since any type of block failure, as well as row and row-column failures, are catastrophic, the error arrival rates of all of these errors can be lumped together:

$$\lambda_f = \lambda_r + \lambda_{rc} + \lambda_{wc} + \lambda_{block}$$

$$\lambda_{block} = \sum \text{arrival rates of all types of block failures for each block}$$

Only the error arrival rate λ_{sc} is altered, and the equation for mean time to failure remains unchanged from equation (32).

For other cases, such as completely independent selection and read-write circuitry for each block, the analysis becomes more complex. Here, the assumption is made that each block is totally independent in that each block has its own selection circuitry. Such a chip can be easily analyzed from the results of the previous subsection if column errors are assumed to be limited to single blocks. No similar restriction is placed on row, row-column, or entire block failures.

From the previous subsection, the reliability of a chip subject to various hard error types was found to be

$$R_c(t) = e^{-\lambda t}\left[e^{c_1 t}c_2 + c_3\right]^M + \frac{\lambda_c e^{-\lambda t}}{M c_1}\left[e^{c_1 M t} - 1\right] \tag{31}$$

where

$$c_1 = \frac{1}{t_s}\ln(1 + \lambda_s n t_s)$$

$$c_2 = 1 + \frac{t_s \lambda_h n}{\ln(1 + \lambda_s n t_s)}$$
$$c_3 = -\frac{t_s \lambda_h n}{\ln(1 + \lambda_s n t_s)}$$

This is the reliability of a chip assuming that the entire chip is effectively one block; therefore, in the independent block case studied here, this is the block reliability. The chip reliability, with N_b blocks per chip, is now

$$\begin{aligned} R_{chip}(t) &= R_c^{N_b}(t) \\ &= e^{-\lambda N_b t}\left[\left[e^{c_1 t}c_2 + c_3\right]^M + \frac{\lambda_c}{M c_1}\left[e^{c_1 M t} - 1\right]\right]^{N_b} \end{aligned} \tag{35}$$

Expanding the binomial powers, equation (35) becomes

$$\begin{aligned} R_{chip}(t) &= e^{-\lambda N_b t}\sum_{i=0}^{N_b}\binom{N_b}{i}\left[e^{c_1 t}c_2 + c_3\right]^{Mi}\left[\frac{\lambda_c}{M c_1}\right]^{N_b - i}\left[e^{c_1 M t} - 1\right]^{N_b - i} \\ &= e^{-\lambda N_b t}\sum_{i=0}^{N_b}\binom{N_b}{i}\left[e^{c_1 t}c_2 + c_3\right]^{Mi}\left[\frac{\lambda_c}{M c_1}\right]^{N_b - i}\sum_{l=0}^{N_b - i}\binom{N_b - i}{l}(-1)^{N_b - i - l}e^{c_1 M t l} \end{aligned} \tag{36}$$

Substituting $c_4 = -\lambda N_b + c_1 M l$ into equation (36),

$$R_{chip}(t) = \sum_{i=0}^{N_b}\sum_{l=0}^{N_b - i}\binom{N_b}{i}\binom{N_b - i}{l}\left[\frac{\lambda_c}{M c_1}\right]^{N_b - i}(-1)^{N_b - i - l}e^{c_4 t}\left[e^{c_1 t}c_2 + c_3\right]^{Mi}$$

which, when integrated over all positive t, yields the mean time to failure as

$$MTTF = \sum_{i=0}^{N_b}\sum_{l=0}^{N_b - i}\sum_{j=0}^{Mi}\binom{N_b}{i}\binom{N_b - i}{l}\binom{Mi}{j}\left[\frac{\lambda_c}{M c_1}\right]^{N_b - i}(-1)^{N_b - i - l + 1}\frac{c_2^j c_3^{Mi - j}}{c_4 + c_1 j} \tag{37}$$

or, equivalently,

$$\begin{aligned} MTTF = \frac{1}{\lambda N_b}&\sum_{i=0}^{N_b}\sum_{l=0}^{N_b - i}\sum_{j=0}^{Mi}\binom{N_b}{i}\binom{N_b - i}{l}\binom{Mi}{j} \\ &\times\left[\frac{\lambda_c}{M c_1}\right]^{N_b - i}(-1)^{N_b - i - l}\frac{\left(1 + \frac{t_s \lambda_h n}{\ln(1 + \lambda_s n t_s)}\right)^j\left(-\frac{t_s \lambda_h n}{\ln(1 + \lambda_s n t_s)}\right)^{Mi - j}}{1 - \frac{j + Ml}{\lambda N_b t_s}\ln(1 + \lambda_s n t_s)} \end{aligned} \tag{38}$$

The coding gain is

$$\begin{aligned} CG = \frac{k}{n}&\sum_{i=0}^{N_b}\sum_{l=0}^{N_b - i}\sum_{j=0}^{Mi}\binom{N_b}{i}\binom{N_b - i}{l}\binom{Mi}{j} \\ &\times\left[\frac{\lambda_c}{M c_1}\right]^{N_b - i}(-1)^{N_b - i - l}\frac{\left(1 + \frac{t_s \lambda_h n}{\ln(1 + \lambda_s n t_s)}\right)^j\left(-\frac{t_s \lambda_h n}{\ln(1 + \lambda_s n t_s)}\right)^{Mi - j}}{1 - \frac{j + Ml}{\lambda N_b t_s}\ln(1 + \lambda_s n t_s)} \end{aligned} \tag{39}$$

Equations (38) and (39) are much more complex than equations (32) and (33), the case of multiple cell errors with one block on a chip. The form of each, however, is similar.

Subsequent subsections will explore various special cases of equation (38) in a manner similar to the single cell error case examined in the previous section. Again, the limit of very fast scrubbing and soft error dominance will be explored, as well as the very slow scrubbing and hard error dominance cases.

2.4.3. MTTF With Fast Scrubbing

For very fast scrubbing, the interval between each soft error scrub, t_s, is made very small. Therefore, c_0 and c_4 remain unchanged, while

$$\begin{aligned} c_1 &= \frac{1}{t_s}\ln(1+\lambda_s n t_s) \quad \rightarrow \quad \lambda_s n \\ c_2 &= 1+\frac{t_s\lambda_h n}{\ln(1+\lambda_s n t_s)} \quad \rightarrow \quad 1+\frac{\lambda_h}{\lambda_s} \qquad \text{as } t_s \rightarrow 0 \\ c_3 &= -\frac{t_s\lambda_h n}{\ln(1+\lambda_s n t_s)} \quad \rightarrow \quad -\frac{\lambda_h}{\lambda_s} \end{aligned}$$

Substituting into equation (37), the mean time to failure becomes

$$\begin{aligned} MTTF = \frac{1}{\lambda N_b}\sum_{i=0}^{N_b}\sum_{l=0}^{N_b-i}\sum_{j=0}^{Mi}\binom{N_b}{i}\binom{N_b-i}{l}\binom{Mi}{j} \\ \times\left[\frac{\lambda_c}{\lambda_s nM}\right]^{N_b-i}(-1)^{N_b-i-l}\frac{\left[1+\frac{\lambda_h}{ls}\right]^j\left[-\frac{\lambda_h}{ls}\right]^{Mi-j}}{1-\frac{\lambda_s n(Ml+j)}{\lambda N_b}} \end{aligned} \tag{40}$$

The coding gain is

$$\begin{aligned} CG = \frac{k}{n}\sum_{i=0}^{N_b}\sum_{l=0}^{N_b-i}\sum_{j=0}^{Mi}\binom{N_b}{i}\binom{N_b-i}{l}\binom{Mi}{j} \\ \times\left[\frac{\lambda_c}{\lambda_s nM}\right]^{N_b-i}(-1)^{N_b-i-l}\frac{\left[1+\frac{\lambda_h}{ls}\right]^j\left[-\frac{\lambda_h}{ls}\right]^{Mi-j}}{1-\frac{\lambda_s n(Ml+j)}{\lambda N_b}} \end{aligned} \tag{41}$$

Intuitively, if scrubbing is done continuously and if soft errors are much more likely than hard errors of any type, *i.e.* $\lambda_s nM \approx \lambda$, then this chip should have an MTTF which is equivalent to the same chip not subject to soft errors and without error correction coding. If $\lambda_s nM \approx \lambda$, then the term $j = Mi$ dominates the third summation; therefore, equation (40) becomes

$$MTTF = \frac{1}{\lambda N_b}\sum_{i=0}^{N_b}\sum_{l=0}^{N_b-i}\binom{N_b}{i}\binom{N_b-i}{l}\left[\frac{\lambda_c}{\lambda_s nM}\right]^{N_b-i}(-1)^{N_b-i-l}\frac{\left[1+\frac{\lambda_h}{ls}\right]^{Mi}}{1-\frac{\lambda_s nM(l+i)}{\lambda N_b}}$$

Furthermore, since $\lambda_c \ll \lambda_h nM \ll \lambda_s nM$, the term

$$\left[\frac{\lambda_c}{\lambda_s nM}\right]^{N_b-i}$$

dominates the summation when $i = N_b$. This forces $l = 0$; therefore,

$$\begin{aligned} MTTF &= \frac{1}{\lambda N_b}\frac{\left[1+\frac{\lambda_h}{ls}\right]^{MN_b}}{1-\frac{\lambda_s nM}{\lambda}} \\ &= [\lambda N_b - \lambda_s nMN_b]^{-1} \\ &= [N_b(\lambda_h nM + \lambda_c + \lambda_f)]^{-1} \end{aligned} \tag{42}$$

Thus, equation (41) confirms the above intuition and is consistent with the results of the single cell fast scrubbing case.

2.4.4. Soft Error Dominance

The effectiveness of soft error scrubbing can be easily seen for the case where soft errors are the dominant error type. In this case, $\lambda_s nM \approx \lambda$; therefore, c_1 remains unchanged,

$$\begin{aligned} 1 + \frac{t_s \lambda_h n}{\ln(1+\lambda_s n t_s)} &\rightarrow 1 \\ -\frac{t_s \lambda_h n}{\ln(1+\lambda_s n t_s)} &\rightarrow 0 \end{aligned} \qquad \text{as } \lambda_h \rightarrow 0$$

Substituting into equation (38), the mean time to failure becomes

$$MTTF = \frac{1}{\lambda N_b} \sum_{i=0}^{N_b} \sum_{l=0}^{N_b - i} \binom{N_b}{i} \binom{N_b - i}{l} \left[\frac{\lambda_c}{M c_1}\right]^{N_b - i} (-1)^{N_b - i - l} \frac{1^{Mi} 0^0}{1 - \frac{(i+l)M}{\lambda N_b t_s} \ln(1 + \lambda_s n t_s)}$$

Since λ_c is small compared to c_1 (which is $\ln(1+\lambda_s n t_s)/t_s$), $i = N_b$ is the only nonzero term, and this forces $l = 0$. Also, $\lambda_s nM \approx \lambda$; the mean time to failure is now:

$$\begin{aligned} MTTF &= \frac{1}{\lambda N_b} \frac{1}{1 - \frac{M}{\lambda N_b t_s} \ln(1 + \lambda_s n t_s)} \\ &= \frac{1}{\lambda N_b} \frac{1}{1 - \frac{\ln(1+\lambda_s n t_s)}{\lambda_s n t_s}} \end{aligned} \tag{43}$$

The coding gain is

$$CG = \frac{k}{n} \frac{1}{1 - \frac{\ln(1+\lambda_s n t_s)}{\lambda_s n t_s}} \tag{44}$$

Using the Taylor series approximation for $\ln(1+y)$ gives the asymptotic behavior of equation (43):

$$\begin{aligned} MTTF &= \frac{1}{\lambda N_b} \left[1 - \frac{\ln(1 + \lambda_s n t_s)}{\lambda_s n t_s}\right]^{-1} \\ &= \frac{1}{\lambda N_b} \left[1 - \frac{\ln(1+y)}{y}\right]^{-1} \\ &= \frac{1}{\lambda N_b} \left[1 - \frac{y - \frac{1}{2}y^2 + \frac{1}{3}y^3 \ldots}{y}\right]^{-1} \\ &\approx \frac{1}{\lambda N_b} \left[\frac{2}{y}\right] \end{aligned} \tag{45}$$

$$CG \approx \frac{k}{n} \left[\frac{2}{y}\right] \tag{46}$$

for small y. These are of identical form to equations (18) and (19), the soft error dominant case analyzed for only single cell failures. The same conclusions drawn in that analysis also apply here.

2.4.5. Very Slow Soft Scrubbing

Analysis of the very slow soft scrubbing ($t_s \to \infty$) case involves reconstruction of the reliability function from first principles; therefore, equations (20) and (21) are used instead of equations (2) and (3):

$$R_0(t) = e^{-\lambda_{sc} nt}[1 + \lambda_s nt] \tag{20}$$

and

$$R_1(t) = \lambda_h nte^{-\lambda_{sc} nt} \tag{21}$$

Using these equations in equation (30), the block reliability function becomes

$$\begin{aligned} R_c(t) &= e^{-\lambda t}(1 + \lambda_{sc} nt)^M + \int_0^t (1 + \lambda_{sc} n\tau)^M \lambda_c d\tau e^{-\lambda t} \\ &= e^{-\lambda t}(1 + \lambda_{sc} nt)^M + e^{-\lambda t} \frac{\lambda_c}{\lambda_{sc} n(M+1)} \left[(1 + \lambda_{sc} nt)^{M+1} - 1\right] \end{aligned}$$

The chip reliability function is now

$$\begin{aligned} R_{chip}(t) &= R_c^{N_b}(t) \\ &= e^{-\lambda N_b t} \sum_{i=0}^{N_b} \binom{N_b}{i} \left[\frac{\lambda_c}{\lambda_{sc} n(M+1)}\right]^i \left[(1 + \lambda_{sc} nt)^{M+1} - 1\right]^i \left[1 + \lambda_{sc} nt\right]^{M(N_b - i)} \\ &= e^{-\lambda N_b t} \sum_{i=0}^{N_b} \sum_{l=0}^{i} \binom{N_b}{i} \binom{i}{l} \left[\frac{\lambda_c}{\lambda_{sc} n(M+1)}\right]^i (-1)^{i-l} [1 + \lambda_{sc} nt]^{(M+1)l} [1 + \lambda_{sc} nt]^{M(N_b - i)} \\ &= e^{-\lambda N_b t} \sum_{i=0}^{N_b} \sum_{l=0}^{i} \sum_{j=0}^{M'} \binom{N_b}{i} \binom{i}{l} \binom{M'}{j} \left[\frac{\lambda_c}{\lambda_{sc} n(M+1)}\right]^i (-1)^{i-l} (\lambda_{sc} nt)^j \end{aligned} \tag{47}$$

where $M' = (M+1)l + M(N_b - i)$. Integrating equation (47) over all positive t,

$$MTTF = \frac{1}{\lambda N_b} \sum_{i=0}^{N_b} \sum_{l=0}^{i} \binom{N_b}{i} \binom{i}{l} \left[\frac{\lambda_c}{\lambda_{sc} n(M+1)}\right]^i \sum_{j=0}^{M'} \binom{M'}{j} \frac{j!}{\left[\frac{\lambda N_b}{\lambda_{sc} n}\right]^j} \tag{48}$$

$$CG = \frac{k}{n} \sum_{i=0}^{N_b} \sum_{l=0}^{i} \binom{N_b}{i} \binom{i}{l} \left[\frac{\lambda_c}{\lambda_{sc} n(M+1)}\right]^i \sum_{j=0}^{M'} \binom{M'}{j} \frac{j!}{\left[\frac{\lambda N_b}{\lambda_{sc} n}\right]^j} \tag{49}$$

The last summation term in equations (48) and (49) is very similar to the $B(M)$ term occurring in [19,20]; since the multiple-cell error rate λ_{mc} is usually much less than the single cell failure rate $\lambda_{sc} nM$, the approximation that

$$\sum_{j=0}^{M'} \binom{M'}{j} \frac{j!}{\left[\frac{\lambda N_b}{\lambda_{sc} n}\right]^j} \approx B(MN_b)$$

can be used. Indeed, if the multiple-cell failure rate approaches zero, equation (48) becomes

$$MTTF = \frac{1}{\lambda N_b} \sum_{j=0}^{MN_b} \binom{MN_b}{j} \frac{j!}{(MN_b)^j} \tag{50}$$

which is consistent with the results obtained in the single cell case examined in the previous section.

2.4.6. Hard Error Dominance

For completeness, the case of hard errors being far more common than soft errors will be analyzed. In this case, the soft error rate is assumed to be very small ($\lambda_s nM \ll \lambda$); therefore, the row reliability function becomes

$$R(t) = e^{-\lambda_h nt}[1 + \lambda_h nt]$$

and the chip reliability function is

$$\begin{aligned} R_{chip}(t) &= \left[e^{-\lambda_{mc}t}e^{-\lambda_h nMt}(1+\lambda_h nt)^M + e^{-\lambda_h nMt}\lambda_c te^{-\lambda_c t}e^{-\lambda_f t}\right]^{N_b} \\ &= e^{-(\lambda_h nM+\lambda_{mc})N_b t}\left[\lambda_c t + (1+\lambda_h nt)^M\right]^{N_b} \\ &= e^{-(\lambda_h nM+\lambda_{mc})N_b t}\sum_{i=0}^{N_b}\binom{N_b}{i}(\lambda_c t)^{N_b-i}(1+\lambda_h nt)^{Mi} \\ &= e^{-(\lambda_h nM+\lambda_{mc})N_b t}\sum_{i=0}^{N_b}\sum_{l=0}^{Mi}\binom{N_b}{i}\binom{Mi}{l}(\lambda_c t)^{N_b-i}(\lambda_h nt)^l \end{aligned}$$

Integrating over all positive t,

$$MTTF = \frac{1}{(\lambda_h nM+\lambda_{mc})N_b}\sum_{i=0}^{N_b}\sum_{l=0}^{Mi}\binom{N_b}{i}\binom{Mi}{l}(\lambda_c)^{N_b-i}(\lambda_h n)^l\frac{(N_b-i+l)!}{[N_b(\lambda_h nM+\lambda_{mc})]^{N_b-i+l}} \quad (51)$$

$$CG = \frac{k}{n}\sum_{i=0}^{N_b}\sum_{l=0}^{Mi}\binom{N_b}{i}\binom{Mi}{l}(\lambda_c)^{N_b-i}(\lambda_h n)^l\frac{(N_b-i+l)!}{[N_b(\lambda_h nM+\lambda_{mc})]^{N_b-i+l}} \quad (52)$$

If multiple-cell errors do not occur ($\lambda_{mc} \to 0$), then equation (51) becomes

$$\begin{aligned} MTTF &= \frac{1}{\lambda_h nMN_b}\sum_{l=0}^{MN_b}\binom{MN_b}{l}\frac{l!}{(MN_b)^l} \\ MTTF &= \frac{1}{\lambda_h nMN_b}B(MN_b) \end{aligned} \quad (53)$$

which is identical to the single cell error results given in the previous section for hard error dominance with single cell failures only.

2.4.7. Multiple-Cell Hard Error Dominance

Consider the case where the multiple-cell hard errors are dominant. In this case, the single cell soft and hard error rates are assumed to be very small compared to the multiple-cell rates ($\lambda_{sc} nM \ll \lambda_{mc}$). First, consider the case where the catastrophic error rate λ_f is dominant. The block reliability function, given by equation (30), is dominated by the term $e^{-\lambda_{mc}t}$: At any given time t,

$$e^{-\lambda_{mc}t} \ll R_0^M(t) \qquad \text{and} \qquad e^{-\lambda_{mc}t} \ll R_1^M(t)$$

Using the above and $\lambda_f \approx \lambda$, equation (30) becomes

$$\begin{aligned} R_c(t) &= e^{-\lambda_f t} + \int_0^t e^{-\lambda_f \tau} \lambda_c d\tau e^{-\lambda_f (t-\tau)} \\ &= e^{-\lambda_f t} \end{aligned}$$

The chip reliability function is now

$$R_{chip}(t) = e^{-\lambda_f N_b t}$$

and the mean time to failure and coding gain are

$$MTTF = \frac{1}{\lambda_f N_b} \tag{54}$$

$$CG = \frac{k\lambda_f}{n\lambda} \tag{55}$$

which are similar to the results expected from an unprotected chip with only catastrophic errors.

The case of column error dominance is slightly more complex. Returning to equation (30), the approximation $\lambda_c \approx \lambda$, and

$$e^{-\lambda_c t} \ll R_0^M(t) \qquad \text{and} \qquad e^{-\lambda_c t} \ll R_1^M(t)$$

the block reliability function becomes

$$\begin{aligned} R_c(t) &= e^{-\lambda_c t} + \int_0^t e^{-\lambda_c \tau} \lambda_c d\tau e^{-\lambda_c (t-\tau)} \\ &= e^{-\lambda_c t}[1 + \lambda_c t] \end{aligned}$$

The chip reliability function is

$$R_{chip}(t) = e^{-\lambda_c N_b t}[1 + \lambda_c t]^{N_b}$$

This is of the same form as equation (23), and the mean time to failure and coding gain are of similar form to equations (23) and (26):

$$MTTF = \frac{1}{\lambda_c N_b} B(N_b) \tag{56}$$

$$CG = \frac{k}{n} B(N_b) \tag{57}$$

where

$$B(N_b) = \sum_{i=0}^{N_b} \binom{N_b}{i} \frac{i!}{{N_b}^i} \tag{24}$$

which is the Birthday Surprise factor [19,20]. This is intuitively obvious: If column errors dominate, and if each block can tolerate a single column error, then the chip reliability is of similar form to the case of a chip composed of a single block subject to only single cell failures without soft error scrubbing.

2.4.8. Single Cell Error Dominance

Consider the case where the single cell hard and soft errors are dominant. In this case, the column and catastrophic error rates are assumed to be very small compared to the single cell rate ($\lambda_f \ll \lambda_{sc} nM$ and $\lambda_c \ll \lambda_{sc} nM$). First, consider the case where the catastrophic error rate λ_f is negligible without making any assumptions on the value of λ_c. Then, by simply substituting $\lambda_c + \lambda_{sc} nM$ for λ in equations (38) and (39),

$$MTTF = \frac{1}{(\lambda_c + \lambda_{sc} nM) N_b} \sum_{i=0}^{N_b} \sum_{l=0}^{N_b - i} \sum_{j=0}^{Mi} \binom{N_b}{i} \binom{N_b - i}{l} \binom{Mi}{j} \times \left[\frac{\lambda_c}{Mc_1}\right]^{N_b - i} (-1)^{N_b - i - l} \frac{\left(1 + \frac{t_s \lambda_h n}{\ln(1+\lambda_s n t_s)}\right)^j \left(-\frac{t_s \lambda_h n}{\ln(1+\lambda_s n t_s)}\right)^{Mi-j}}{1 - \frac{j+Ml}{(\lambda_c + \lambda_{sc} nM) N_b t_s} \ln(1 + \lambda_s n t_s)} \tag{58}$$

and

$$CG = \frac{k}{n} \sum_{i=0}^{N_b} \sum_{l=0}^{N_b - i} \sum_{j=0}^{Mi} \binom{N_b}{i} \binom{N_b - i}{l} \binom{Mi}{j} \times \left[\frac{\lambda_c}{Mc_1}\right]^{N_b - i} (-1)^{N_b - i - l} \frac{\left(1 + \frac{t_s \lambda_h n}{\ln(1+\lambda_s n t_s)}\right)^j \left(-\frac{t_s \lambda_h n}{\ln(1+\lambda_s n t_s)}\right)^{Mi-j}}{1 - \frac{j+Ml}{(\lambda_c + \lambda_{sc} nM) N_b t_s} \ln(1 + \lambda_s n t_s)} \tag{59}$$

Consider now the case of very few column errors ($\lambda_c \to 0$). Therefore, in equation (38), the term $[\lambda_c / Mc_1]^{N_b - i}$ is significant only when $N_b = i$, or when the term is equal to unity. This selection of i forces l to be 0, and therefore, equation (38) becomes

$$MTTF = \frac{1}{\lambda N_b} \sum_{j=0}^{MN_b} \binom{MN_b}{j} \frac{\left(1 + \frac{t_s \lambda_h n}{\ln(1+\lambda_s n t_s)}\right)^j \left(-\frac{t_s \lambda_h n}{\ln(1+\lambda_s n t_s)}\right)^{MN_b - j}}{1 - \frac{j}{\lambda N_b t_s} \ln(1 + \lambda_s n t_s)} \tag{60}$$

and the coding gain is

$$CG = \frac{k}{n} \sum_{j=0}^{MN_b} \binom{MN_b}{j} \frac{\left(1 + \frac{t_s \lambda_h n}{\ln(1+\lambda_s n t_s)}\right)^j \left(-\frac{t_s \lambda_h n}{\ln(1+\lambda_s n t_s)}\right)^{MN_b - j}}{1 - \frac{j}{\lambda N_b t_s} \ln(1 + \lambda_s n t_s)} \tag{61}$$

Equations (60) and (61) are similar to the case of only single cell failures given in equations (9) and (10). Intuitively, if the column errors are no longer significant, the only failure modes are two single cell failures in any given row or a catastrophic error; the model confirms this analysis.

2.5. Simulation Results

Theoretical expectations were calculated and compared with simulation results. The complex model represented by equation (38) was used for completeness. Since mean time to failure

simulations are very time-consuming, the mean events to failure (METF) were computed in the simulations; the MTTF can be determined from the METF and the mean error arrival rate λ via equation (1):

$$MTTF = \frac{1}{\lambda} METF$$

The METF was obtained by the following means: In most realistic cases, Assumptions I and II hold, so the hard error arrival rate is much lower than the soft error arrival rate, and the refresh cycle length is shorter than the mean time between soft error arrivals. An error of some type (soft, single cell hard, column, or catastrophic error) was assumed to occur, with any number of additional errors determined randomly with a Poisson distribution, in a single scrub cycle time period. The type of error was determined randomly so that the average number of each type of error is proportional to its error arrival rate. These errors were allowed to occur in any one of N_b blocks containing M codewords. If more than one error occurred in any codeword, a failure was declared. Scrubbing was conducted after these error insertions; all soft errors were cleared while hard errors remained.

The number of errors required to result in a chip failure were recorded and averaged; this provided the mean events to failure. This procedure is justified by the following. Since errors occur only occasionally in each scrub cycle, those scrub cycles where no error occurs can be ignored; this means that all scrub cycles where at least one error of any type occurs are examined but all other scrub cycles are ignored. Since the mean events to failure is typically large—the effect of soft errors is diminished by scrubbing and hard errors are rare—the mean time to failure can be approximated by the product of the mean error inter-arrival time $1/\lambda$ and the mean events to failure. Results were obtained by averaging over 1000 tries; these are plotted below.

2.5.1. Varying the Scrub Interval

Figure 6 shows the effect of keeping the error rates constant while the scrub cycle time was varied. This log plot shows that there is a threshold characteristic for soft error scrubbing; if the chip is scrubbed faster than some rate, there is little increase in MTTF. Also, if the chip is scrubbed much slower, the MTTF does not decrease much beyond some other level. This transition is directly linked to the failure mode of the chip: When the scrub cycle time is slow, then most of the chip failures are of the soft-soft type, where two soft errors cause chip failure, because $\lambda_s nM \gg (\lambda_h nM + \lambda_c + \lambda_f)$. These failure modes were confirmed through simulations. When the chip is scrubbed at a faster rate, the failures are of the soft-hard type, where one soft and one hard error in a codeword cause

chip failure.

Note that the model given by equation (38) matches the simulation results closely except for large values of scrub cycle time, where the model described by equation (50) holds. The data smoothly switch from one model to the other. For small t_s, the MTTF should be as given in equation (42). This is indeed the case, as $\lambda_h nMN_b = 10^{-6} sec^{-1}$, and the MTTF levels off just above $10^6 sec$. For large t_s, the mean time to failure should be as given in equation (50):

$$MTTF = \frac{1}{\lambda N_b} \sum_{j=0}^{MN_b} \binom{MN_b}{j} \frac{j!}{(MN_b)^j}$$

For $MN_b = 1024$, $B(1024) = 40.78$; again the model matches simulation, as $\lambda N_b \approx 10^{-3} sec^{-1}$ and the MTTF levels off at about $4 \times 10^5 sec$.

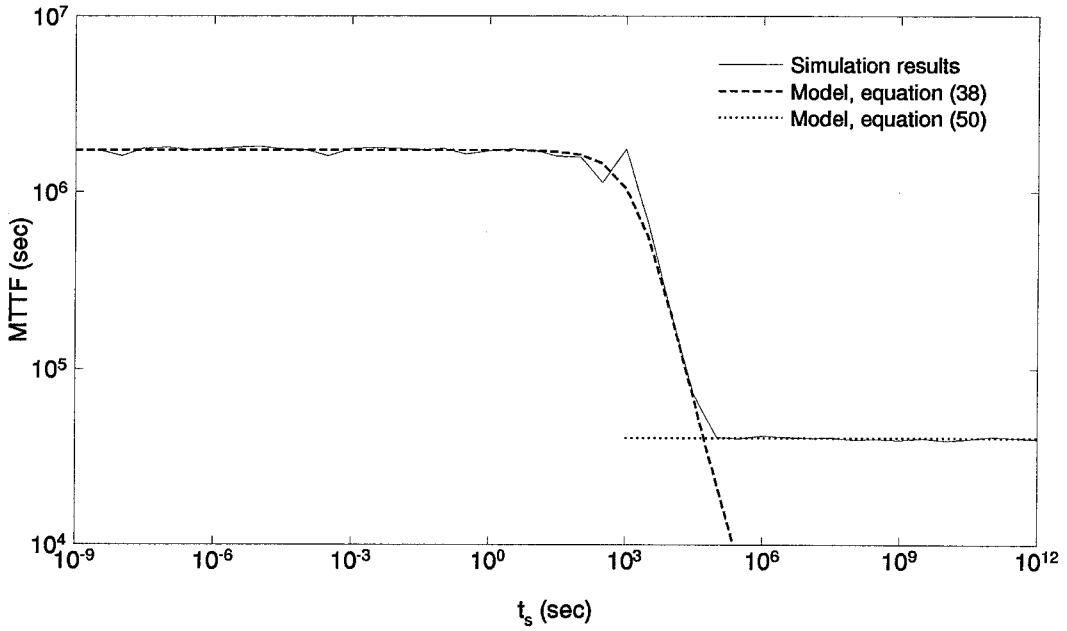

Figure 6: MTTF vs. t_s. MTTF with $\lambda_h nMN_b = 10^{-6} sec^{-1}$, $\lambda_s nMN_b = 10^{-3} sec^{-1}$, $\lambda_c N_b = 10^{-9} sec^{-1}$, $\lambda_f N_b = 10^{-12} sec^{-1}$, $M = 128$, and $N_b = 8$.

The MTTF is expected to be near the hard error rate if scrubbing is fast, because soft errors do not have time to build up and hard errors remain. Failure primarily occurs from having a soft error occurring in the same codeword as a hard error (which is not removed by scrubbing). Since the hard error rate is much lower than the soft error rate, the relative time period between a hard error strike and a subsequent soft error strike in the same codeword is small compared to the time period from start to first hard error strike. Therefore, the MTTF approaches the MTTF with no

protection and only hard errors, as expected by equation (42):

$$MTTF = \frac{1}{N_b(\lambda_h nM + \lambda_c + \lambda_f)} \quad \text{as } t_s \to 0,\ \lambda_s \approx \lambda$$

When scrubbing is slow, the model described by equation (50) applies, and this predicts a plateau at

$$MTTF = \frac{1}{\lambda N_b} B(MN_b)$$

when $t_s \to \infty$. Note that in this case, soft error scrubbing is not effective; therefore, the MTTF is determined mainly by the number of codewords and not the scrub period.

A point of interest to system designers would be the knee of the curve, where the MTTF drops from its higher plateau. In Figure 6 this occurs at $t_s \approx 10^3$ seconds. This is the maximum scrub interval permissible before system performance degrades significantly.

2.5.2. Varying the Soft Error Rate

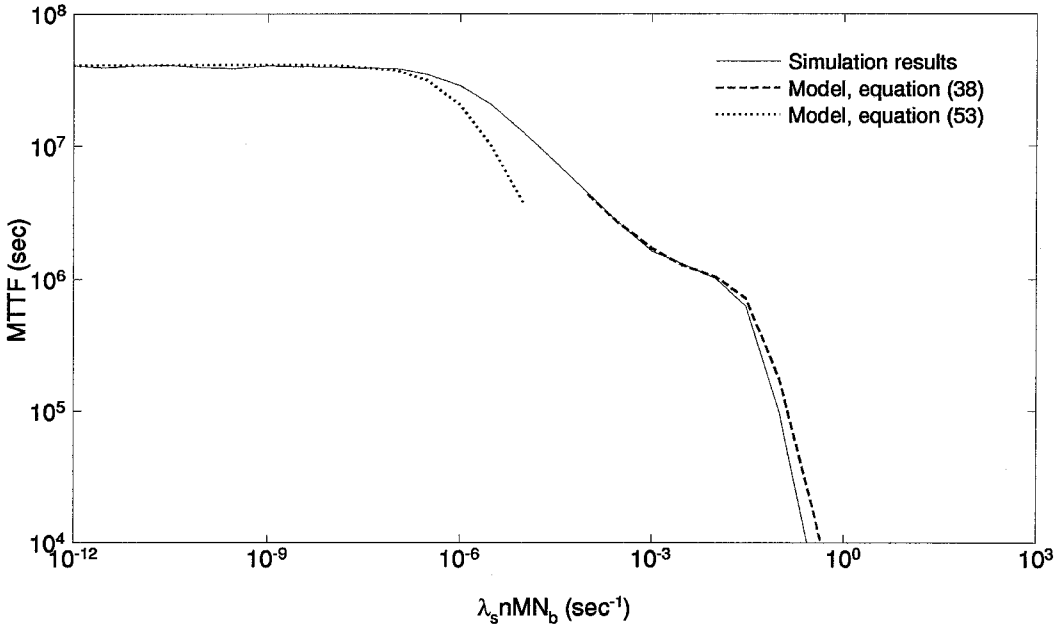

Figure 7: MTTF vs. $\lambda_s nMN_b$. MTTF with $\lambda_h nMN_b = 10^{-6} sec^{-1}$, $\lambda_c N_b = 10^{-9} sec^{-1}$, $\lambda_f N_b = 10^{-12} sec^{-1}$, $t_s = 1.0 sec$, $M = 128$, and $N_b = 8$.

If the soft error rate is altered, then, as shown in Figure 7, there is one flat region in the MTTF plot. Again, model and simulation are close. If λ_s is very large, then most of the failures are of the soft-soft type, where two soft errors strike the same codeword within the same scrub period, and soft error scrubbing is not removing soft errors fast enough. Thus, as the soft error arrival rate

increases, MTTF decreases. The decrease is linear on the log-log plot, and this confirms equation (45), the case where soft errors dominate.

When λ_s is very small, the hard error arrival rates determine the value of the MTTF. Most failures come from two hard failures occurring in the same codeword; these are kept constant here, so the plot is flat. This is the case where soft errors are practically nonexistent and therefore hard errors are dominant: Assumption II is not valid, and the model described by equation (53) must be used. In this region, the simulation results match the results from equation (53) well.

In between these two regions, there is a region with not as great a fall-off as when λ_s is large. This is where failures of the soft-hard type occur, where one hard error and one soft error occur in the same codeword. Since the MTTF here is dependent on the hard error rate, which remains constant for this plot, and the soft error rate, the decrease in MTTF is not as drastic as it is for larger λ_s, where the soft errors dominate the MTTF.

2.5.3. Varying the Hard Error Rate

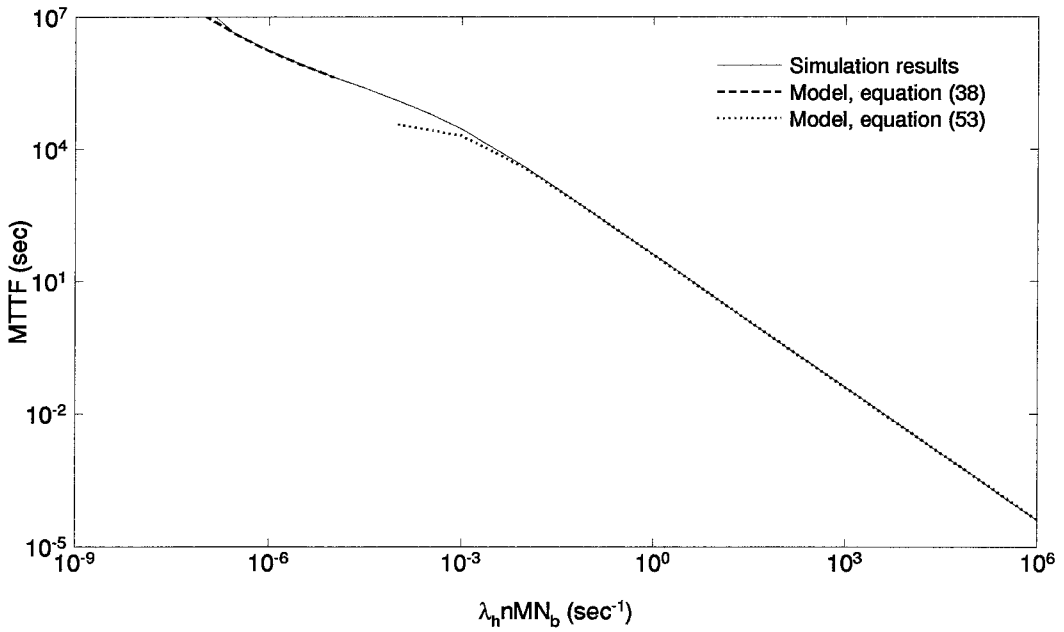

Figure 8: MTTF vs. $\lambda_h nMN_b$. MTTF with $\lambda_s nMN_b = 10^{-3} sec^{-1}$, $\lambda_c N_b = 10^{-9} sec^{-1}$, $\lambda_f N_b = 10^{-12} sec^{-1}$, $t_s = 1.0 sec$, $M = 128$, and $N_b = 8$. Model given by (38) in dashed curve; model (53) in dotted curve; simulation results in solid curve.

The effect of altering the hard error rate λ_h is shown in Figure 8. For the left side of the curve, the failure modes are primarily soft-hard: Soft errors are scrubbed out fast enough to prevent accumulation in a codeword during a single scrub cycle, and hard errors are much less frequent

than soft errors, so the codeword fails primarily when a soft error strikes a codeword whose error correcting power has been negated previously by a hard error. This is the case described by equation (42); therefore, the hard error arrival rate dictates the MTTF, and the response should be linear with a negative slope for a log-log plot. The model tracks the simulation closely.

The right side of the plot, however, is the region where hard errors dominate; here, failures occur most commonly from two hard errors occurring in the same codeword. Here, Assumption II is not valid, so the model developed in equation (53) must be used. This is also plotted on Figure 8, and is shown to track the experimental data well.

2.5.4. Varying the Multiple-Cell Error Rate

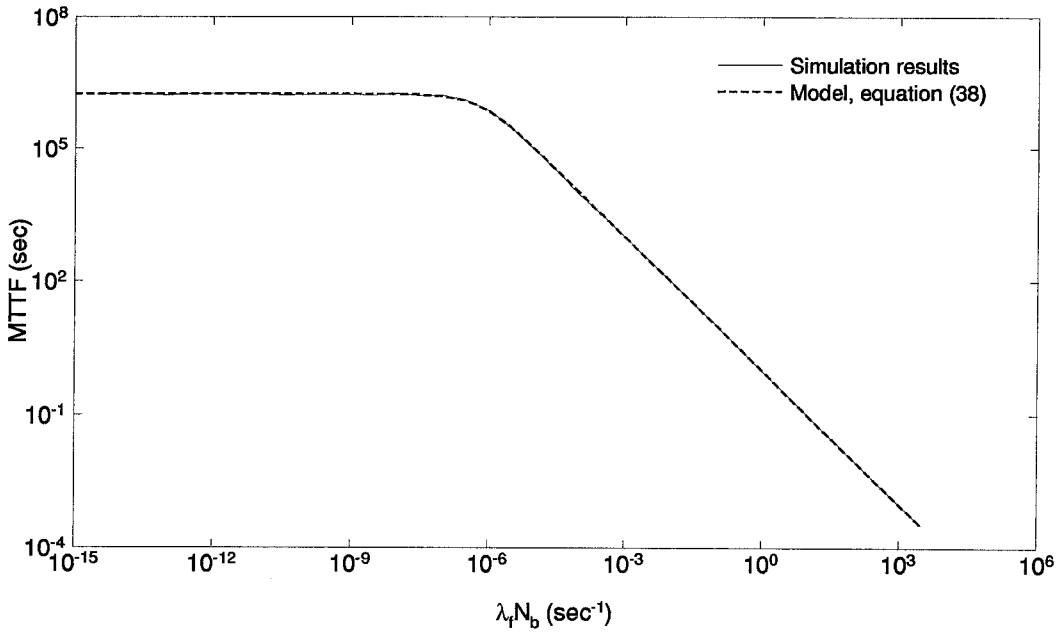

Figure 9: MTTF vs. $\lambda_f N_b$. MTTF with $\lambda_h nMN_b = 10^{-6}sec^{-1}$, $\lambda_s nMN_b = 10^{-3}sec^{-1}$, $\lambda_c N_b = 10^{-9}sec^{-1}$, $t_s = 1.0sec$, $M = 128$, and $N_b = 8$.

The effect of altering the multiple-cell error rates λ_c and λ_f yields results which are simple and predictable. First, consider the case when the column failure rate λ_c is altered. When the column failure rate is very small, the MTTF is predictably flat, since the failure modes are predominantly single cell failures. However, as λ_c increases, the main failure mode becomes the column-soft type. (In this case the soft errors are much more common than single-cell hard errors.) The MTTF, therefore, degrades linearly with respect to λ_c on a log-log plot.

However, as λ_c increases further, the common failure mode becomes two column errors

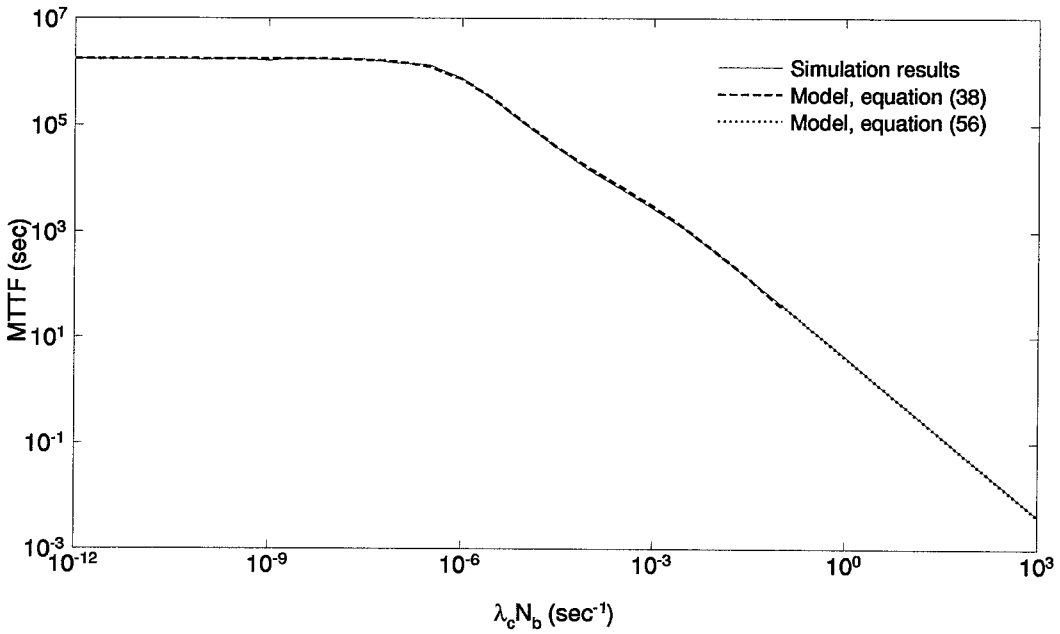

Figure 10: MTTF vs. $\lambda_c N_b$. MTTF with $\lambda_h nMN_b = 10^{-6}sec^{-1}$, $\lambda_s nMN_b = 10^{-3}sec^{-1}$, $\lambda_f N_b = 10^{-12}sec^{-1}$, $t_s = 1.0sec$, $M = 128$, and $N_b = 8$.

occurring in the same block; therefore, the MTTF behaves as predicted in equation (56). Both models track the MTTF simulation very closely.

Now consider the case when the catastrophic failure rate λ_c is altered. When the catastrophic failure rate is very small, the MTTF is predictably flat, since the failure modes are predominantly single cell failures. However, since the chip fails at the first occurrence of a catastrophic error, the MTTF drops linearly with respect to λ_f on a log-log plot as the catastrophic failure rate is increased.

2.5.5. Varying the Number of Codewords

Figure 11 shows the effect on the MTTF of altering the number of codewords M while the per-chip error arrival rates remain constant. This is the case where the chip is coded with varying size codewords while the number of cells remains constant. As shown in Figure 11, the increase in MTTF from increasing the number of codewords is small; in previous plots, the increases have been over several orders of magnitude. Here, the range is linear, and the MTTF is barely doubled by squaring the number of codewords in the chip. This confirms the relation found in [21]:

$$METF \sim \sqrt{M} \tag{62}$$

Again, the model is close to the simulation results.

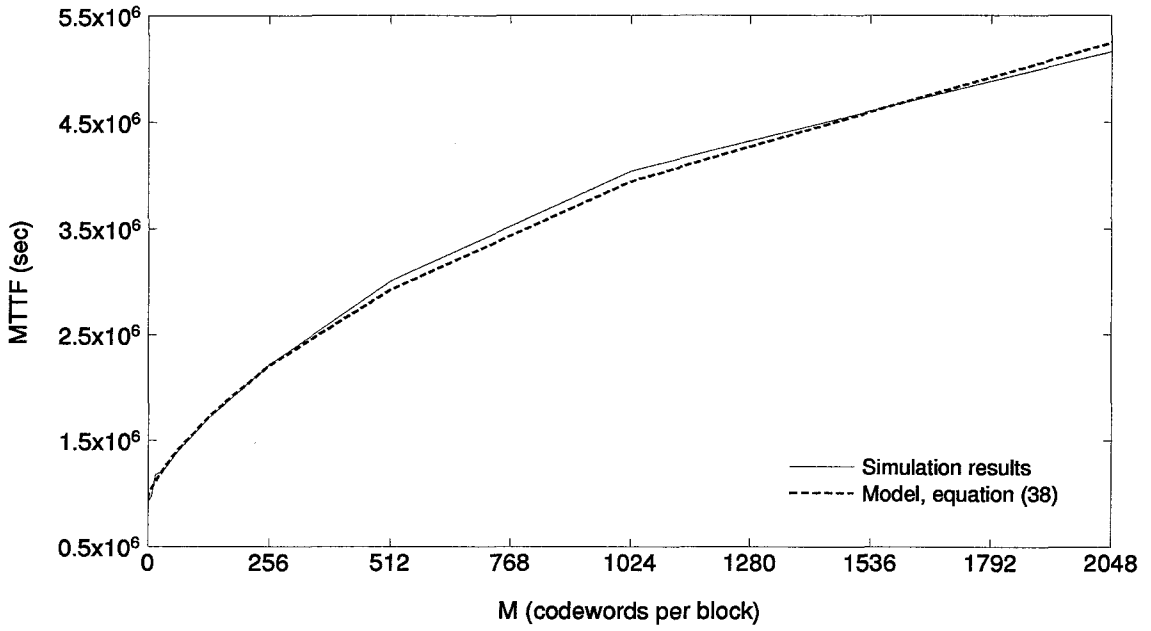

Figure 11: MTTF vs. M. MTTF with $\lambda_h nMN_b = 10^{-6}sec^{-1}$, $\lambda_s nMN_b = 10^{-3}sec^{-1}$, $\lambda_c N_b = 10^{-9}sec^{-1}$, $\lambda_f N_b = 10^{-12}sec^{-1}$, $t_s = 1.0sec$, and $N_b = 8$.

2.6. Conclusion

This chapter develops an accurate model for the mean time to failure of a semiconductor RAM protected with a single error correcting code along its word lines. The model takes into account multiple blocks per chip, several types of hard errors, single cell soft errors, and soft error scrubbing to derive the mean time to failure, thus providing a complete picture of the gain obtained by coding a memory chip with a single error correcting code along the rows. The model was tested with computer simulations and was found to be an accurate representation of the memory. The equation for MTTF presented can be directly used by the system designer wishing to assess the benefits of coding for memory protection.

2.7. Appendix

In this chapter, two assumptions were presented and justified as reasonable characteristics of real chips. First, the mean time to failure must be much greater than the scrub interval; second, the soft error rate must be much greater than the hard error rate:

Assumption I: The scrub cycle must be small compared to the mean time to failure: $t_s \ll MTTF$

Assumption II: The hard error rate must be small compared to the soft error rate: $\lambda_h \ll \lambda_s$

These assumptions must hold in the model described by equation (8) as well for the following reasons. In equation (2) the probability of correct operation given that no hard errors occurred was presented as

$$R_0(t) = e^{-\lambda_{sc} nt}(1 + \lambda_s n t_s)^{t/t_s}$$

This was derived by using the approximation that a smooth, continuous function which equals another piecewise continuous function at regular intervals can be used to approximate that piecewise function. That piecewise function arises from the discrete time analysis of the soft error scrubbing effect, and it is, for the case of having no hard errors in the codeword,

$$\begin{aligned} R_0'(t) &= \left[e^{-\lambda_h t}\right] P(\text{0 or 1 soft error in time 0 to } \lfloor t/t_s \rfloor) P(\text{0 or 1 soft error in time } \lfloor t/t_s \rfloor \text{ to } t) \\ &= \left[e^{-\lambda_h nt}\right]\left[(e^{-\lambda_s nt_s} + \lambda_s n t_s e^{-\lambda_s nt_s})^{\lfloor t/t_s \rfloor}\right]\left[e^{-\lambda_s n(t - \lfloor t/t_s \rfloor t_s)}[1 + \lambda_s n(t - \lfloor t/t_s \rfloor t_s)]\right] \\ &= e^{-\lambda_{sc} nt}(1 + \lambda_s n t_s)^{\lfloor t/t_s \rfloor}[1 + \lambda_s n t_s(t/t_s - \lfloor t/t_s \rfloor)] \end{aligned} \tag{63}$$

where $\lfloor x \rfloor$ is the largest integer less than x.

R_0' is equal in value to R_0 when t/t_s is an integer; that is, $t/t_s = \lfloor t/t_s \rfloor$. In between these points, when t/t_s is not an integer, $\lfloor t/t_s \rfloor$ is constant at the last integer value. R_0 and R_0' are of the form

$$\begin{aligned} R_0(t) &= e^{-At}(1 + \lambda_s n t_s)^{t/t_s} \\ &= e^{-At} Q(t) \\ R_0'(t) &= e^{-At}(1 + \lambda_s n t_s)^{\lfloor t/t_s \rfloor}[1 + \lambda_s n t_s(t/t_s - \lfloor t/t_s \rfloor)] \\ &= e^{-At} Q'(t) \end{aligned}$$

As can be seen, $Q'(t)$ is a piecewise linear approximation of $Q(t)$, a monotonically increasing function. Therefore, $Q'(t) > Q(t)$, and thus, $R_0' > R_0$. Thus, equation (2) is a lower bound of the exact solution given by equation (63) using the Poisson approximation. However, $R_0 = R_0'$ at values of t determined by t_s, and R_0 does not vary far from R_0' when t_s is sufficiently small. t_s is sufficiently small when $t_s \ll MTTF$. This confirms the need for Assumption I, as otherwise $R_0(t)$ will not be a good approximation for the more accurate $R_0'(t)$. (See Figure 12.)

Note that the discrete scrub period analysis is much more complex than the continuous scrub period case, as is evident by the complexity of the reliability function. Carrying through this analysis will yield a mean time to failure model which is more complex to derive and interpret than the models presented in this chapter, and is therefore unsuitable for use.

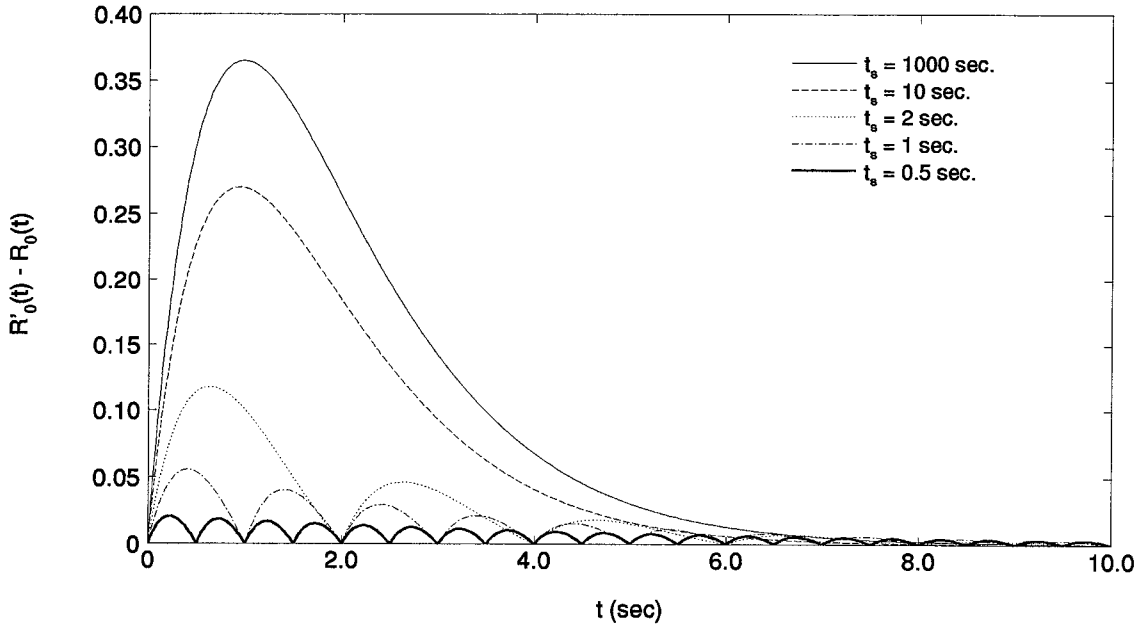

Figure 12: Difference between $R_0'(t)$ and $R_0(t)$. Parameters were $\lambda_h = 10^{-7} sec^{-1}$, $\lambda_s = 10^{-4} sec^{-1}$, and $n = 1024$.

In equation (3), another assumption was made, that Assumption II must hold as well as Assumption I. This assumption assures that $R_1(t)$ is a good approximation of $R_1'(t)$ in the following manner. $R_1(t)$ is the integral over τ of the product of three terms: First the probability that no failure had occurred given no hard error had occurred from 0 to τ; second, the probability that a hard error occurred in time $d\tau$; third, the probability that no error, hard or soft, occurred between τ and t. Thus, the first term is $R_0(\tau)$. This by itself justifies the use of Assumption I here.

In addition, the following analysis is needed. Without soft errors, the peak of $R_1(t)$ occurs at $1/\lambda_h$. This peak will be shifted toward zero with the occurrence of soft errors. Since $R_0(t)$ is not accurate for large t_s—and equivalently small λ_s, due to the time scaling property of the model as described in equation (9)—there must be sufficiently many scrub cycles before $t = 1/\lambda_h$. This means that $1/\lambda_h \gg t_s$. Now $\lambda_s n t_s \sim 1$ so that soft errors do not pile up in any one codeword during one scrub cycle. Also, if $\lambda_s n t_s \gg 1$, there will be a high probability that two or more errors will occur in a single codeword within a scrub cycle, so in this case fewer errors (and therefore fewer scrub cycles) are needed to cause a failure. Thus, $\lambda_s n t_s \sim 1$ must be maintained to avoid violation of Assumption I. This leads to

$$\frac{1}{\lambda_h} \gg t_s \sim \frac{1}{\lambda_s}$$

or

$$\frac{\lambda_h}{\lambda_s} \ll 1$$

Thus, Assumption II must hold.

CHAPTER 3
PHASED BURST
ERROR CORRECTING ARRAY CODES

3.1. Introduction

In computer memory and communications applications information can be corrupted by bursts of noise which occur within one of many predetermined sectors or time intervals. These noise patterns will be called phased burst errors [24] because although the noise pattern may be random at each burst, its duration and starting points are restricted to certain intervals. Noise sources which can generate these errors include line noise, synchronization errors in demodulation, timing errors in multi-valued memories, and backscatter radar signals. These errors are often periodic in time (or, in the case of memories, in position) and can be long in duration. (See Figure 13.)

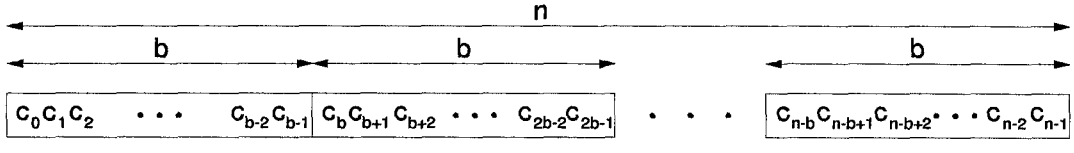

Figure 13: Phased codeword. Errors can occur only within each b-symbol section.

A motivation for studying this problem is the encoding of multi-level random access memories, where each cell contains more than one bit of data. These memories use dynamic RAM cells to store one of several discrete voltages. An experimental 4 Mbit chip with 16 possible voltage levels (4 bits worth of data) stored in each cell was reported in [5]. Voltage levels in each cell are stored and sensed by ramping the voltages on pertinent row and column select lines. These memories, unlike CCD type memories, are not serial, but have random access times which are much longer ($\sim 200\mu$sec) than binary RAMs and are therefore not optimal for working memory. However, since an entire row can be read at each access, blocks of data are available quickly; this, coupled with their high density, makes them ideal for mass storage devices.

The advantages of multi-level memories are evident when they are used as mass storage

devices. Compared to magnetic hard disks, multi-level RAMs have faster block access times ($\sim 200\mu$s vs. $\sim$20ms), comparable throughput (20Mb/s), and low power requirements (standby power of $\sim 100\mu$W and active power of $\sim$10mW vs. standby and active power of $\sim$1W). Magnetic disks require mechanical drives which cannot be scaled downward with disk capacity; that is, for smaller disks, there is a significant "volume overhead" which decreases the storage density with respect to the entire disk package. Multi-level RAMs have no such penalty. Therefore, these memories are useful for replacing medium to small capacity magnetic hard disks where power is restricted, as in laptop computers.

The main drawback of multi-level RAMs, aside from the need for constant power to maintain memory contents, is the following. Since the cells contain charges separated by small discrete steps and since access timing accuracy is important, timing errors can cause errors in entire rows of memory. Error correcting codes used at the chip level typically encode a single row as a codeword; these types of codes are thus useless in this case, where the entire row can be erroneous. In practice, large random access memory chips are broken up into a number of blocks. A code can be placed across these block lines to break up the long burst at the cost of increased access circuitry and encoder/decoders. The resulting codewords experience long phased bursts of errors in relatively short codewords. (See Figure 14.) A code which can correct long phased bursts with high rate and short codeword length is desirable for this application.

Most burst error correcting codes which can correct long bursts have extremely long codewords which must be shortened—and therefore result in a lower rate code—when used in applications with small block sizes. Fire codes, for example, have high rate at the expense of large codewords, and they do not take advantage of the phased nature of the error. Thus, standard burst error correcting codes such as the Fire code may not be best for correcting phased bursts in practice, especially where a high-rate, high-speed, short codeword length code is needed. However, some codes, such as Reed-Solomon codes over serially arranged bit blocks, have high rate, can correct phased bursts, and are optimal. Here, array codes are presented as an alternative to the Reed-Solomon codes to correct single phased burst errors.

Array codes offer the advantages of block structure and easy encoding and decoding. The concept was first introduced by Elias [25]; the first array codes were Gilbert codes, developed in 1960 [26]. Gilbert codes are constructed from a two-dimensional array of cells with row and column parities, and their size can be varied greatly. Given a diagonal (helical) readout order on a rectangular array of size n_1 by n_2, with s being the diagonal skipping value, burst error correction is possible.

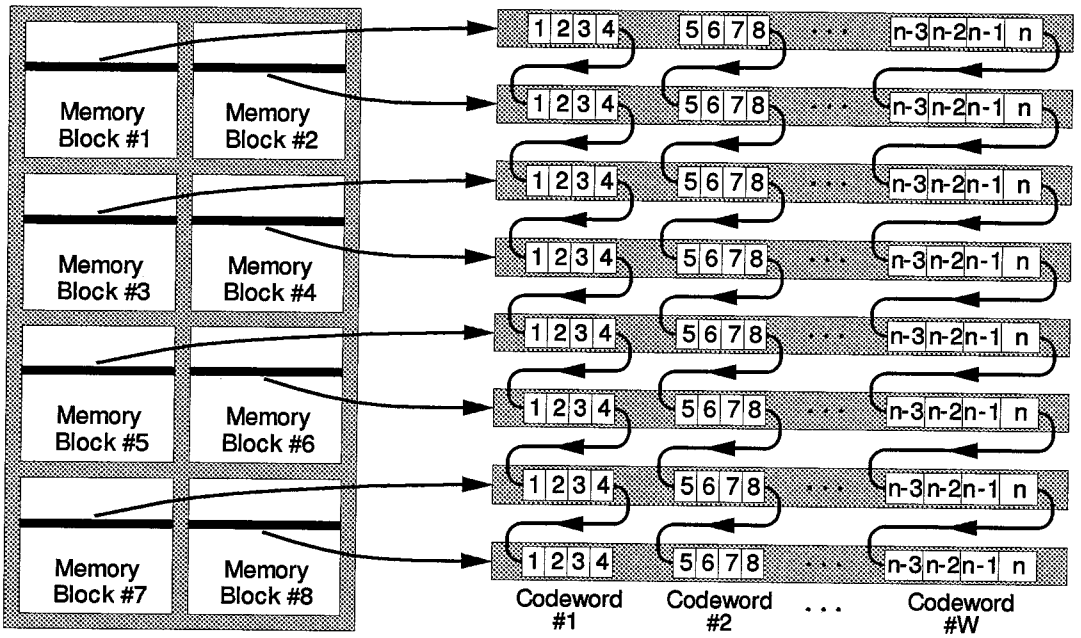

Figure 14: Coding across multiple rows. This figure shows an example with eight blocks in a chip, N cells per row, $W = N/4$ codewords of size 8×4 with possible phased bursts of $b = 4$. Any one row in a block can be in error, and the code can correct this error.

An example is shown in Figure 15.

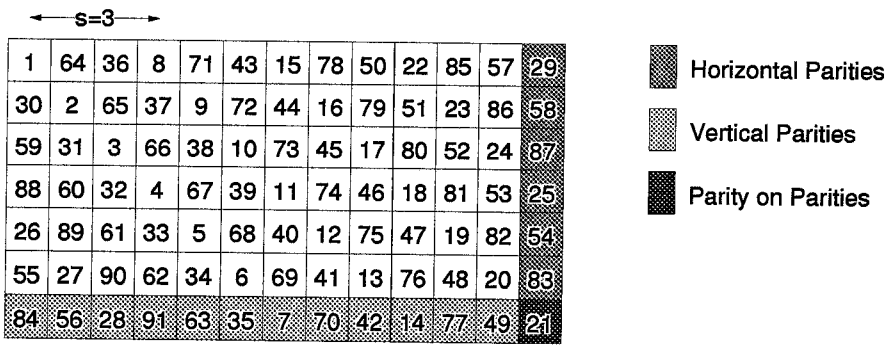

Figure 15: Array codeword. Example is shown for $n_1 = 7$, $n_2 = 13$, and $s = 3$. Readout order as numbered.

Bounds on correctable burst lengths of Gilbert codes were studied by Neumann in 1965 [27]. Bahl and Chien in 1969 [28] made corrections to Neumann's work, and generalized the result for higher dimensional Gilbert codes in 1971 [29]. Burton and Weldon also studied bounds on burst length [30]. Generalization of the Gilbert code to form burst error correcting array codes, along with the study of the special case for $s = 1$, was done by Farrell and Hopkins in 1982 [31]. In 1986, Blaum, *et al.*, showed that for $b = n_1 - 1$, $n_2 \geq 2n_1 - 3$ is required [32]. Zhang and Wolf generalized

the bound to find b for any n_1, n_2, and s in 1988 [33]. Further work by Blaum has shown that $b = n_1$ is attainable for $s = -1$ and $s = -2$ [34,35]; these results were generalized and codes with efficiencies approaching unity were found by Zhang [36] and Sivarajan, McEliece, and van Tilborg [37].

Previous work concentrated mainly on general bursts, where burst starting location is not restricted. Under the limitation that all bursts occur only within one of many preselected blocks, array codes can be made extremely powerful and efficient without much difficulty, resulting in short codewords with long burst correcting powers. Phased burst error correcting array codes are constructed, as are general burst error correcting array codes, from two-dimensional arrays of cells with row and column simple parity checks and diagonally cyclic readout order. Note that the parity on parities (also called the check on checks, in the rightmost bottom corner) can be considered to be both a vertical and a horizontal parity check. Errors are now restricted to occur only in a single diagonal; therefore, the readout parameter s, which denotes how many columns to skip before reading the next diagonal, is irrelevant here. Previous work on general burst error correcting array codes made mention of s: The length of a correctable burst depends on s [33,34]. For phased bursts, s can be any value so long as s and n_2 are mutually prime (which ensures that the entire array is filled).

Various properties of single phased burst error correcting array codes will be explored in this chapter. Section 3.2 contains theorems on the allowed codeword sizes for phased burst correction. In Section 3.3 optimal codeword sizes for this code are discussed. Section 3.4 contains two possible decoding algorithms. Section 3.5 will cover array codes for the correction of approximate errors, those errors in q-ary codes in which the erroneous value is restricted to be no more than Δ away from the true value.

3.2. Allowed Codeword Sizes

As mentioned in the previous section, the connection between array sizes and general burst error length for array codes has been found. For phased burst error correction, not much work has been accomplished. Here, this problem of finding the allowed array sizes for array codes capable of correcting a burst confined to a single diagonal in the readout order will be solved.

Array codes have the highest rate when they are square; that is, when $n_2 = n_1$. Consider array codes capable of correcting phased bursts of length n_1. These codes cannot have $n_2 \leq n_1$.

Lemma 1: An array code with diagonal cyclic readout order can never correct a single phased burst along a diagonal if $n_1 \geq n_2$.

Proof: Consider two cases, $n_1 = n_2$ and $n_1 > n_2$. For $n_1 = n_2$, a burst pattern of

$$E = [\underbrace{1 \quad 1 \quad 1 \quad \cdots \quad 1}_{n_1 \text{ terms}}]$$

will, regardless of location, create the same values for the vertical and horizontal parity checks: Each vertical and horizontal parity will be nonzero. Therefore, regardless of the size, if $n_1 = n_2$, then the array code cannot correct phased bursts along one diagonal.

For $n_1 > n_2$, consider a burst of the form

$$E = [1 \underbrace{0 \quad 0 \quad \cdots \quad 0}_{s\text{-}1 \text{ terms}} 1 \underbrace{0 \quad 0 \quad \cdots \quad 0}_{n_2\text{-}s\text{-}1 \text{ terms}} 1 \underbrace{0 \quad 0 \quad \cdots \quad 0}_{n_1\text{-}n_2\text{-}1 \text{ terms}}] \tag{64}$$

Note that $0 < s < n_1$. This error pattern will result in all zeroes in both the horizontal and vertical parity checks; therefore, if $n_1 > n_2$, then the array code cannot correct phased bursts along one diagonal. (See Figure 16.) ∎

1	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 16: Example of decoder error with $n_1 > n_2$. The parity checks are contained in the rightmost column and bottommost row. Error pattern of the form given by equation (64) is shown for $n_1 = 11$, $n_2 = 8$, and $s = 3$. Note that parity check values are all zero, thus making this error not correctable.

These array codes can always correct a single phased error when $n_2 \geq 2n_1$. This is obvious: The vertical parities will have a string of zeroes which is $n_2 - n_1$ long; this is where the error cannot exist. This region is unambiguous because the number of leading or trailing zeroes to the phased burst is known from the horizontal parities, which reflects the error pattern.

Lemma 2: An array code with diagonal cyclic readout order can always correct a single phased burst along a diagonal if $n_2 \geq 2n_1$.

Proof: The horizontal parities provide the burst pattern. Thus, the number of trailing and leading zeroes to the error pattern are known. On the vertical parities, there will be a string of zero elements at least $n_2 - n_1$ long, because the burst can affect at most n_1 vertical parity elements.

The "starting" position of the burst projected onto the vertical parities can be found by finding the "ending" position of the string of cyclically contiguous zeroes that is at least $n_2 - n_1$ long and then offsetting this position by the number of leading zeroes of the burst. Since $n_2 \geq 2n_1$, the string of cyclically contiguous zeroes is at least of length n_1. All bursts are confined to being no longer than n_1 in length; therefore, the positions which correspond to this string are unambiguous, and the codeword can be decoded regardless of the error pattern or location. ■

In the region $n_1 < n_2 < 2n_1$, the code is not guaranteed to successfully correct a single phased burst, but it is also not guaranteed to fail, either. In this region the following theorem governs the existence or nonexistence of a single phased burst error correcting code.

Theorem 1: An array code with diagonal cyclic readout order can correct a single phased burst along a diagonal for $n_1 < n_2 \leq 2n_1$ if and only if

$$n_2 \neq \frac{\alpha+1}{\alpha}(n_1 - \beta) \tag{65}$$

where α and β are positive nonzero integers [38].

Proof: For equation (65) to be true, the complement of equation (65) must be a necessary and sufficient condition for erroneous decoding. That is, the array is not capable of correcting a single phased burst if and only if

$$n_2 = \frac{\alpha+1}{\alpha}(n_1 - \beta) \tag{66}$$

where α and β are positive nonzero integers.

For single phased error correction, the horizontal parities of the codeword uniquely provide the phased burst error pattern regardless of the vertical parities, but give no information regarding the burst location. The vertical parities must therefore provide this positional information. Decoder error or failure can occur if and only if there exists an error pattern which gives the same vertical parity pattern for two different positions.

This statement is equivalent to saying decoder error or failure occurs if and only if the vertical parity pattern is periodic; *i.e.* the vertical parities, arranged in a circular list, remain unchanged after a cyclical shift by e. A symbolic representation of this vertical parity pattern is given in Figure 17. The vertical parity pattern remains unchanged when the parities are cyclically

shifted by e positions. The parity values are represented by sections of length x which are regions not all equal to zero and sections of length b, a, and c which are all zeroes.

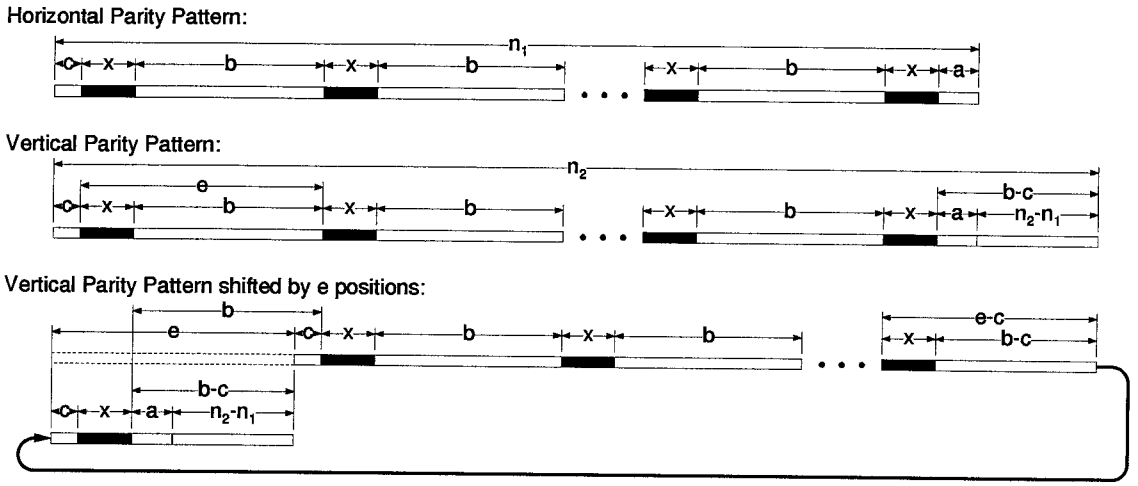

Figure 17: Symbolic representation of the horizontal and vertical parity patterns. Regions with not all zero values of length x are interleaved with regions of all zero values of length b; leading and trailing edge values of length c and a, respectively, are also all zero. The vertical parity pattern which is cyclically shifted by e is indistinguishable from the original pattern.

Note that the case where the shift is $2e$ or some larger multiple of e is also possible. However, such a shift is only indistinguishable if the original shift of e is indistinguishable; therefore, this case need not be addressed separately.

From Figure 17,

$$n_2 = (\alpha + 1)(x + b) \tag{67}$$

where α is a nonnegative integer, and

$$b = e - x = c + a + n_2 - n_1 \tag{68}$$

Combining equations (67) and (68) and solving for n_2 yields

$$n_2 = \frac{\alpha + 1}{\alpha}[n_1 - (x + c + a)]$$

Because $x \geq 1$, $c \geq 0$, and $a \geq 0$, the factor $x + c + a$ can be compressed into one term β, $\beta \geq 1$, so that

$$n_2 = \frac{\alpha + 1}{\alpha}(n_1 - \beta) \tag{66}$$

These are the values of n_2 which are periodic.

Equation (66) therefore gives the necessary and sufficient conditions for decoder error or failure; therefore, the array code can correct a single phased burst if and only if

$$n_2 \neq \frac{\alpha+1}{\alpha}(n_1 - \beta) \tag{65}$$

where α and β are positive nonzero integers. ■

n_1	Allowed n_2, $n_2 < 2n_1$	n_1	Allowed n_2, $n_2 < 2n_1$
2	**3**	**22**	**23**,29,31,33,35,37,39,41,43
3	5	23	29,31,35,37,39,41,43,45
4	**5**,7	24	29,31,35,37,39,41,43,45,47
5	7,9	25	29,31,35,37,39,41,43,45,47,49
6	**7**,9,11	26	29,31,35,37,39,41,43,45,47,49,51
7	11,13	27	29,31,35,37,41,43,45,47,49,51,53
8	11,13,15	**28**	**29**,31,35,37,41,43,45,47,49,51,53,55
9	11,13,15,17	29	31,37,41,43,45,47,49,51,53,55,57
10	**11**,13,15,17,19	**30**	**31**,37,41,43,45,47,49,51,53,55,57,59
11	13,17,19,21	31	37,41,43,47,49,51,53,55,57,59,61
12	**13**,17,19,21,23	32	37,41,43,47,49,51,53,55,57,59,61,63
13	17,19,21,23,25	33	37,41,43,47,49,51,53,55,57,59,61,63,65
14	17,19,21,23,25,27	34	37,41,43,47,49,51,53,55,57,59,61,63,65,67
15	17,19,23,25,27,29	35	37,41,43,47,49,53,55,57,59,61,63,65,67,69
16	**17**,19,23,25,27,29,31	**36**	**37**,41,43,47,49,53,55,57,59,61,63,65,67,69,71
17	19,23,25,27,29,31,33	37	41,43,47,49,53,55,57,59,61,63,65,67,69,71,73
18	**19**,23,25,27,29,31,33,35	38	41,43,47,49,53,55,57,59,61,63,65,67,69,71,73,75
19	23,25,29,31,33,35,37	39	41,43,47,49,53,55,59,61,63,65,67,69,71,73,75,77
20	23,25,29,31,33,35,37,39	**40**	**41**,43,47,49,53,55,59,61,63,65,67,69,71,73,75,77,79
21	23,29,31,33,35,37,39,41	41	43,47,49,53,55,59,61,63,65,67,69,71,73,75,77,79,81

Table 2: Codeword sizes. Allowed codeword sizes for $n_1 < n_2 < 2n_2$. Optimal codes are in bold type, for which n_2 is prime.

Equation (65) serves to sieve out those values of n_2 which do not form array codes capable of correcting single phased errors. Some of these allowed sizes are listed in Table 2. However, finding the allowed codeword sizes for a given n_1 requires applying equation (65) for all α which are the prime factors of $n_1 - \beta$ and all β from 1 to $(n_2 - n_1 + 1)$. This is not efficient. A faster means can be found by simply solving equation (66) for n_1 and taking the complement:

$$\begin{aligned} n_2 &= \frac{\alpha+1}{\alpha}(n_1 - \beta) \\ \frac{\alpha}{\alpha+1}n_2 &= n_1 - \beta \\ n_1 &= n_2\frac{\alpha}{\alpha+1} + \beta \end{aligned} \tag{66}$$

where α and β are integers such that $\alpha \geq 1$ and $\beta \geq 1$. Therefore, the equivalent statement to equation (65) is

$$n_1 \neq n_2\frac{\alpha}{\alpha+1} + \beta \qquad \text{where } \alpha \geq 1 \text{ and } \beta \geq 1 \tag{69}$$

Since $\beta \geq 1$, the right side of equation (69) can increase without limit; therefore, equation (69) is equivalent to

$$n_1 \leq n_2 \frac{\alpha}{\alpha+1} \qquad \text{where } \alpha \geq 1 \tag{70}$$

Since the right side of equation (70) must be an integer, $\alpha+1$ must be a factor of n_2. The right side is small when $\alpha+1$ is small; therefore, the right side is smallest when $\alpha+1$ is the smallest factor of n_2, which is equivalent to the smallest prime factor of n_2; call this factor κ. Substitution of κ for $\alpha+1$ in equation (70) results in

$$n_1 \leq n_2\left(1-\frac{1}{\kappa}\right) \tag{71}$$

where κ is the smallest prime factor of n_2. The allowed values of n_2 can be found more easily using equation (71) than by using equation (65).

Neumann presented a result very similar to equation (71) in [27] as the maximum general burst length correctable by an array. Specifically, the claim was made that given an array of size $n_1 \times n_2$, the maximum burst length correctable in a Gilbert code is

$$b = \min\left[n_1\left(1-\frac{1}{\kappa_1}\right), \quad n_2\left(1-\frac{1}{\kappa_2}\right)\right] \tag{72}$$

where κ_1 is the smallest prime divisor of n_1 and κ_2 is the smallest prime divisor of n_2. This was shown to be incorrect by Bahl and Chien [28]. For phased bursts, Neumann's results are not applicable; therefore, this is a new result.

3.3. Optimal Codeword Sizes

The results of the previous section, namely, Theorem 1 and equation (71), can be used to find the characteristics of optimal single phased error correcting array codes.

Corollary 1.1: For any n_1, the set of allowed values of n_2 for the single phased burst error correcting code includes all prime numbers greater than n_1.

Proof: From equation (71),

$$n_1 \leq n_2\left(1-\frac{1}{\kappa}\right)$$

where κ is the smallest prime factor of n_2. If n_2 is prime, then because $\kappa = n_2$, equation (71) becomes $n_1 \leq n_2 - 1$. Thus, all prime numbers greater than n_1 are acceptable values of n_2. ∎

3.3.1. Meeting the Singleton Bound

Burst error correcting codes are measured in optimality by efficiency rather than by rate. However, single phased burst error correcting codes can be considered not as being burst error correcting, since these phased bursts are effectively symbols taken in a serial bit stream, but as being single error correcting codes. Therefore, the rate is the measure of optimality.

Linear error correcting codes can reach but not exceed the Singleton Bound, which states that to correct t errors, a code must have at least $2t$ parity symbols:

$$2t \leq n - k$$

The code is capable of correcting a single phased burst, so $t = 1$. When n_2 is prime, using Corollary 1.1,

$$n_2 \geq n_1 + 1$$

Take $n_2 = n_1 + 1$, so that the array is as square as possible. This should provide a codeword with as high a rate as possible [39]. Then, taking each symbol to be a group of bits n_1 long,

$$\begin{aligned} n - k &= \left\lceil \frac{n_1 n_2}{n_1} \right\rceil - \left\lceil \frac{(n_1 - 1)(n_2 - 1)}{n_1} \right\rceil \\ &= n_2 - \frac{n_1 n_2 - n_1 - n_2 + 1}{n_1} \\ &= n_2 - n_2 + \frac{n_1 + (n_1 + 1) - 1}{n_1} \\ &= 2 \end{aligned}$$

Therefore, single phased burst array codes can be optimal in the sense that they reach the Singleton Bound; this occurs when n_2 is prime and $n_1 = n_2 - 1$. This was known previously [40] and is confirmed as a special case of Theorem 1.

Table 2 lists some allowed codeword sizes for single phased burst error correcting array codes; optimal codeword sizes are in boldface type. Additionally, note that the smallest n_2 allowed for a given n_1 is the smallest prime number larger than n_1. This claim will be shown to be true for all $n_2 < 10^7$ in the next subsection; to prove the claim true for all n_2 is an unsolved number theory problem.

3.3.2. Minimum Value of n_2

Table 2 hints at the claim made here, that for any n_1, the smallest allowed n_2 is the smallest prime number which is still greater than n_1. Though this claim cannot be proven outright, it can be given strong supporting evidence. In this section, a theorem which, coupled with examination of all

prime numbers less than 10^7, proves the claim for all $n_2 < 10^7$, will be presented. Since practically block lengths are much smaller than 10^7, the claim is proven for all practical values of n_2 and n_1.

The claim that the smallest n_2 for any given n_1 is the smallest prime number greater than n_1 will be supported in two steps. First, a theorem will be proven showing that if the difference between the two dimensions $n_2 - n_1$ is less than the square root of the larger number, the original claim is true. In the worst case, this difference between the two dimensions will be the difference between two consecutive prime numbers. Second, this theorem will be applied to all prime numbers below 10^7; the claim will be shown to hold true for these numbers. These two will help show that in the finite range $0 < n_2 < 10^7$ the claim is true, and that the claim in general relies on an unsolved problem in number theory.

Theorem 2: The smallest allowed n_2 for a given value of n_1 is the smallest prime number larger than n_1 if

$$n_2 - n_1 < \sqrt{n_2} \tag{73}$$

Proof: Lemma 1 shows that $n_2 > n_1$ is required for the code to be capable of correcting a single phased burst. Lemma 2 shows that if $n_2 \geq 2n_1$ the code is guaranteed to correct a single phased burst. Therefore, the region of interest, where the code is capable but not assured of correcting a single phased burst, is $n_1 < n_2 < 2n_1$. Corollary 1.1 has shown that all prime numbers in this range are valid values for n_2 for the code to work.

Choose n_{2p} to be the smallest prime number greater than n_1, and n_{2c} to be one of the composite (nonprime) numbers greater than n_1 and less than n_{2p}. If equation (71) is to be satisfied,

$$n_1 \leq n_{2c}\left(1 - \frac{1}{\kappa_c}\right) \tag{74}$$

where κ_c is the smallest prime factor of n_{2c}. The smallest prime factor of any composite number is less than the square root of that number; therefore, for an acceptable n_{2c}—where equation (74) holds—to exist,

$$\begin{aligned} n_1 &\leq n_{2c}\left(1 - \frac{1}{\sqrt{n_{2c}}}\right) \\ &\leq n_{2c} - \sqrt{n_{2c}} \end{aligned} \tag{75}$$

For any n_{2c} such that $n_1 < n_{2c} < n_{2p}$, if equation (75) is true, then n_{2c} is a valid codeword dimension. Since the opposite is desired, that is, that there are no valid composite (nonprime) n_{2c} in the range $n_1 < n_{2c} < n_{2p}$, and since equation (74) is a direct consequence of Theorem 1, which

gives a necessary and sufficient condition for acceptable values of n_1 and n_2,

$$n_1 > n_{2c} - \sqrt{n_{2c}}$$

which can be rearranged to obtain

$$n_{2c} - n_1 < \sqrt{n_{2c}} \tag{76}$$

as the condition required for no n_2 less than the smallest prime number larger than n_1 to be an acceptable codeword size. Since the largest that n_{2c} can be is $n_{2p} - 1$, equation (76) becomes

$$(n_{2p} - 1) - n_1 < \sqrt{n_{2p} - 1}$$

which implies

$$n_{2p} - n_1 < \sqrt{n_{2p}}$$

■

Since the worst case (largest) difference between n_1 and n_2 occurs when both n_1 and n_2 are primes, all prime numbers below 10^7 were checked; this revealed that equation (73) holds for all pairs of consecutive primes except the pair 113 and 127. Here, $n_2 - n_1 < 14$ is possible while $\sqrt{n_2} = \sqrt{127} = 11.27$. Because Theorem 2 covers only cases where $n_2 - n_1 < \sqrt{n_2}$, n_2 is covered only in the range $113 < n_2 \leq 124$. Thus, the cases $n_2 = 125$ and $n_2 = 126$ were checked manually using equation (71),

$$n_1 \leq n_2\left(1 - \frac{1}{\kappa}\right)$$

When $n_2 = 125$, $n_1 \leq 100$ is required, and when $n_2 = 126$, $n_1 \leq 63$ is required for the code to be phased burst correcting. Since $n_1 = 113$, these two are not valid codewords. Therefore, even in the special case where equation (73) fails, the claim holds true for all prime numbers less than 10^7. Since most codewords are smaller than 10^7, the analysis thus far is usually sufficient for practical codeword sizes.

Theorem 2 suggests that if all consecutive primes p_n and p_{n+1} are separated by a distance $d_n = p_{n+1} - p_n$ which is less than $\sqrt{p_{n+1}}$, then the claim regarding phased burst array code sizes, namely, that the smallest n_2 allowed is the smallest prime number greater than n_1, is true. This is an unsolved problem in number theory. The best results obtained so far are close but insufficient [41],

$$d_n = O(p_n^\theta) \qquad \text{where } \theta = \frac{11}{20} - \frac{1}{384} \approx 0.5474$$

However, since the theorem has been proven for all n_2 less than 10^7, this claim can be assumed true for all practical purposes: Almost all applications have codeword sizes of less than 10^7.

3.4. Decoding Strategies

Encoding information bits entails taking the parities of the rows and the columns as needed. For q-ary codes, these parities are taken modulo q, and it is therefore an easy and fast operation. This can be implemented quickly and in parallel using XOR gates for $q = 2$ or their nonbinary equivalent, addition modulo q, for $q > 2$.

For decoding, the parities are first recomputed from the received codeword; from these the positions with parity violations (and in the case of nonbinary fields, the amount of the violation) are known. Consider the parity on parities to be both a horizontal as well as a vertical parity value. Because bursts are restricted to one diagonal, the horizontal parities give the burst pattern. The vertical parities, given the burst pattern, provide information on the burst position. The horizontal parities are "stuffed" with $n_2 - n_1$ zeroes so that both the horizontal and vertical parities are of length n_2. The vertical and horizontal parities can be represented as

$$V = [v_0 \quad v_1 \quad v_2 \quad \cdots \quad v_{n_2-1}]$$

$$H = [h_0 \quad h_1 \quad h_2 \quad \cdots \quad h_{n_2-1}]$$

respectively. The problem now is finding the error position; that is, finding a value of e such that $0 \leq e < n_2$ and

$$[v_{e \bmod n_2} \quad v_{(e+1) \bmod n_2} \quad v_{(e+2) \bmod n_2} \quad \cdots \quad v_{(e+n_2-1) \bmod n_2}] = [h_0 \quad h_1 \quad h_2 \quad \cdots \quad h_{n_2-1}]$$

Once the error location is known, error correction is accomplished by XORing this erroneous row with the horizontal parities. (For q-ary applications, the horizontal parities are subtracted modulo n_2 from the erroneous row.)

3.4.1. Cyclic Convolution

One method for finding the error position e is cyclic convolution. This algorithm can be parallelized for extremely fast implementation. By cyclically convolving the two parity sets V and H, the error position can be found: The error location e occurs where the convolution yields a maximum. In both the binary and nonbinary cases, this amounts to finding the number of cyclic shifts of the horizontal parities needed to match the pattern given by the vertical parities. The decoding algorithm can be implemented in parallel with n_2 convolution circuits, thus quickly yielding the position of the burst in two steps: First, calculate the cyclic convolutions in parallel; second, find the maximum value. This, however, requires a large amount of circuitry or computing nodes ($n_2{}^2$

computations). Serially implemented, this algorithm requires ${n_2}^2$ multiplications, ${n_2}^2 - n_2$ additions, and n_2 comparisons. In parallel, using ${n_2}^2$ computing nodes, the algorithm requires the time for one multiplication, $\log_2 n_2$ additions, and $\log_2 n_2$ comparisons in sequence.

3.4.2. Shiloach's Algorithm

One other method for finding e is using Shiloach's Algorithm [42,43]. This is a serial implementation which requires, at most, $3(n_2 - 1)$ comparisons. Shiloach's Algorithm is much faster and requires less circuitry than a serially implemented cyclic convolution algorithm, but it is still slower than a parallel implementation of the cyclic convolution algorithm and cannot be effectively parallelized. An outline of Shiloach's algorithm follows. For two vectors A and B of length n defined as

$$A = [a_0 \quad a_1 \quad a_2 \quad \cdots \quad a_{n-1}]$$

$$B = [b_0 \quad b_1 \quad b_2 \quad \cdots \quad b_{n-1}]$$

two integers $i, j < n$ can be found such that

$$[a_{i \bmod n} \quad a_{(i+1) \bmod n} \quad a_{(i+2) \bmod n} \quad \cdots \quad a_{(i+n-1) \bmod n}]$$

$$= [b_{j \bmod n} \quad b_{(j+1) \bmod n} \quad b_{(j+2) \bmod n} \quad \cdots \quad b_{(j+n-1) \bmod n}]$$

The algorithm returns $k = 0$ if no such match can exist and $k = n$ if there is a valid match.

$i = 0;$
$j = 0;$
$k = 0;$
WHILE $(i < n$ AND $j < n$ AND $k < n)$
{ IF $(a_{(i+k) \bmod n} = b_{(j+k) \bmod n})$ $k = k + 1;$
ELSE IF $(a_{(i+k) \bmod n} < b_{(j+k) \bmod n})$ { $i = i + k + 1;$ $k = 0;$ }
ELSE { $j = j + k + 1;$ $k = 0;$ }
}

Implemented serially, this algorithm requires at most $3(n_2 - 1)$ comparisons and $3(n_2 - 1)$ additions.

3.5. Correcting Approximate Errors

In some analog memory and communications applications the codeword contains q-ary symbols, that is, more than one bit of information is stored in a single memory cell (16-level RAMs [5], for example) or more than one bit is sent per use of the channel through the use of many discrete analog values to represent these q-ary symbols (amplitude and phase modulation in high speed modems, for example). If these codewords are subject to errors which only change the received/retrieved symbol

slightly from the transmitted/stored symbol, the information which remains in these cells can be used to help correct the errors. For example, in multi-level memories, as mentioned in Section 2.1, if there are timing errors in reading or writing the row of cells, the contents of that entire row will possibly be close to but in error from the true value. This information can be used to help correct the codeword.

Another example is in PSK (phase shift keying) modulation. Here, the symbols are a ring of size q, and the most common errors will be those closest to the correct value. Drifting from the correct sync value will cause errors which are close to the correct value. This condition can be corrected the next time the signal is synchronized, and synchronization is typically done either continuously or at regular intervals. The second case, used more often for high-speed applications, can result in time intervals with incorrect sync which are corrected by the next interval, resulting in bursts which are phased. FSK (frequency shift keying) has a similar problem in that the correct frequency may be mistaken for one close to that frequency; therefore, a code which is tailored to correct these most common errors with as high a rate as possible will be useful.

In this section an approach using the remanent information in corrupted symbols to correct these errors—and therefore use less redundant symbols in the encoding—will be explored. This type of error will be referred to as approximate errors, or those with erroneous values which are always approximately equal to the actual values. (This situation can be considered to be one with a skewed error distribution, since the errors are restricted to a certain class. Bitwise errors of this type and codes to handle them were studied in [3]; here a different approach is used.) If the actual value stored in a given symbol were x and if the symbol were in error, the resulting received/retrieved symbol will be between $x - \Delta$ and $x + \Delta$, where $\Delta < q/2$ and q is the number of values each symbol can take. The symbols can be considered to be a ring of size q or can be terminated at the minimum and maximum.

3.5.1. Parity Values for Approximate Errors

The errors addressed in this section have magnitudes within Δ of the correct values. Since the errors are of limited severity, the parities need not be taken modulo q, as they were before. To correct errors of magnitude Δ, it is necessary to take parities modulo $2\Delta+1$ to be able to distinguish between the $2\Delta + 1$ values which the error can take. (This includes one value for zero, the no error case.) However, it is not necessary to take two orthogonal sets of parities modulo $2\Delta + 1$. The two sets of parities in the phased burst error correcting array code contribute in a distinct manner,

with the horizontal set of parities determining error pattern and vertical parities determining error position with the help of the horizontal parities. The horizontal parities, therefore, must be taken modulo $2\Delta+1$ to uniquely represent the error pattern. Determining error position, however, can be done by knowing only the existence or nonexistence of an error at each horizontal and vertical parity position. This can be accomplished by taking the parity modulo $\Delta+1$, to distinguish between the Δ levels (including the no error case) of error possible.

Therefore, it is sufficient to record only the parities taken modulo $\Delta+1$ for the vertical parities to determine error location and modulo $2\Delta+1$ for the horizontal parities to determine error value. By taking the more numerous vertical parities modulo $\Delta+1$ and the less numerous horizontal parities modulo $2\Delta+1$ fewer bits are needed to store this information. The parity on parities is taken modulo $2\Delta+1$ so that it may be used as a horizontal parity value.

Initially only the approximate error correcting case will be explored; later, parity cell compression will also be included in the analysis. The encoding format is the same as before, but with the parities taken with differing modulo sums. Decoding is also as before, with the following exceptions: First, the parities which were taken modulo $2\Delta+1$ need to be recomputed for modulo $\Delta+1$. These parities are then used to locate the error burst in the manner described above. Second, the burst is then corrected using the modulo $2\Delta+1$ parities.

Error correction is accomplished as follows. The magnitude of the error is limited, $|e|<\Delta$. Define $\delta=2\Delta+1$. The horizontal parities were originally computed so that for each row i,

$$\left[\sum_{j=0}^{n_2-1} c_{ij}+h_i\right] \bmod \delta = 0$$

where c_{ij} represents the contents of the jth cell in row i, and h_i is the horizontal parity of row i. Suppose an error of e occurs in cell c_{ik}. If r_{ij} is the received (and possibly erroneous) version of c_{ij}, then

$$\begin{aligned}\left[\sum_{j=0}^{n_2-1} r_{ij}+h_i\right] \bmod \delta &= e \bmod \delta \\ &= z\end{aligned}$$

Since values of $e<0$ are mapped onto the region $\Delta+1$ to 2Δ, the following correction rules apply:

$$c_{ik}=\begin{cases} r_{ik}-z & \text{if } \ 0<z\le\Delta \\ r_{ik}-z+\delta & \text{if } \ \Delta<z\le 2\Delta \end{cases}$$

Thus, correction of approximate errors is possible with vertical parities taken modulo $\Delta+1$ and horizontal parities taken modulo $2\Delta+1$.

3.5.2. Parity Cell Compression

Parity cell compression for phased burst error correcting array codes is achieved as follows. The vertical parities are taken modulo $\Delta + 1$, and the horizontal parities, modulo δ. Since only a few bits of information are needed to record both sets of parities, there is some unused capacity for storing information in these parity symbols. These can be taken advantage of by storing more than one parity value in each symbol. There are two ways this can be accomplished: The parities can be compressed and represented as a column of length $n_1 - 1$, in which case this acts like another column added to the array of information symbols (without any parities), or they can be represented as a row of length $n_2 - 1$, in which case this acts like another row added to the array of information symbols. (See Figure 18.)

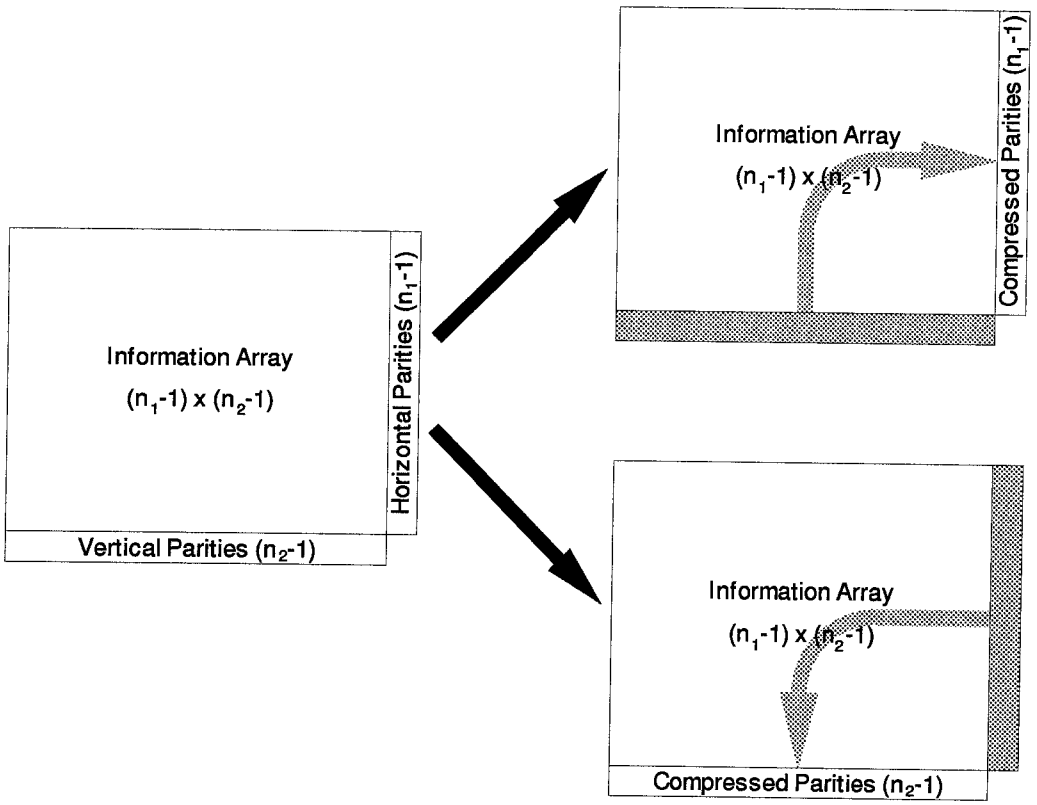

Figure 18: Parity cell compression methods. Parities can be packed into $n_1 - 1$ cells (top) or $n_2 - 1$ cells (bottom). Code becomes effectively a $(n_1 - 1) \times n_2$ code capable of correcting a phased burst of $n_1 - 1$, or a $n_1 \times (n_2 - 1)$ code capable of correcting a phased burst of length n_1. Parity on parities is not used and assumed to be errorless.

In both cases, the parity on parities value is unused and is therefore not coded. Effectively, the code becomes a linear sum code [13]. This code can still correct a single phased burst using the information that the parity on parities is immune to error, a condition which occurs because the

parity on parities is always assumed to be zero.

Note that this parity compression, therefore, is equivalent to removing a row or a column of parity values from the code and placing them in other cells. The error correction capacity of the code should remain unchanged; therefore, the code must be capable of correcting errors along any single diagonal, though the error magnitude must be within Δ for all values. This implies that the remaining column or row of parities can contain an error as well. However, because there is more than one parity value recorded into each symbol, the values must be coded in such a way that the additional parity values stored there are not affected, since these additional values record parity from rows or columns which differ from the location at which they are stored. Therefore, the requirements for parity compression are: Vertical parities are taken modulo $\Delta + 1$; horizontal parities are taken modulo $2\Delta + 1$; parities must be stored into $n_1 - 1$ or $n_2 - 1$ symbols; parity on parities is unneeded; and those additional parity values stored in a symbol must remain unaffected by errors within magnitude Δ.

The original parity value can be stored without coding since if it is erroneous, it can be corrected. This error can alter the value of the entire symbol by up to $\pm\Delta$. However, if a simple means of storing two parity values in the same symbol is used, a carry or borrow error can occur. That is, by storing the two parity values as

$$P = p_{\text{original}} + (\Delta + 1)p_{\text{additional}}$$

the two values can be found by

$$p_{\text{additional}} = \left\lfloor \frac{P}{\Delta + 1} \right\rfloor$$

$$p_{\text{original}} = P \bmod (\Delta + 1)$$

However, if an error occurs in P, a carry or a borrow may occur and alter the values stored, causing an error to propagate into the value for $p_{\text{additional}}$:

$$\left\lfloor \frac{P}{\Delta + 1} \right\rfloor - 1 \leq \left\lfloor \frac{P + e}{\Delta + 1} \right\rfloor \leq \left\lfloor \frac{P}{\Delta + 1} \right\rfloor + 1$$

The effect of a carry, borrow, or neither alters the resulting calculated $p_{\text{additional}}$ by $+1$, -1, or 0, respectively.

Thus, a buffer is needed between p_{original} and $p_{\text{additional}}$ that is large enough to distinguish between the carry, borrow, or neither carry nor borrow cases. The two cases, either having horizontal parities included into existing vertical parities or having vertical parities included into existing horizontal parities, will be addressed separately. The strategy for both is similar.

For horizontal parities included into existing vertical parities, consider the following. A horizontal parity value ($p_2 = h_i$) is included into an existing vertical parity value ($p_1 = v_j$) by

$$P = p_1 + \gamma p_2$$

where γ is some constant. Let $Q = \gamma p_2$. Assume p_2 is known.

$$Q \leq P \leq Q + \Delta$$

It is obvious that $\gamma > \Delta$ is necessary. Now let an error of up to Δ in magnitude occur in P.

$$Q + e \leq P + e \leq Q + e + \Delta$$

which, in the worst case (maximum range the values can span), is

$$Q - \Delta \leq P + e \leq Q + \Delta + \Delta \tag{77}$$

To be uniquely able to distinguish between the case where a carry, a borrow, or neither occurs, there must be an unambiguous mapping given that the worst case occurs. From equation (77), the maximum range which $P + e$ can span given p_2 is

$$\gamma - 1 = (Q + 2\Delta) - (Q - \Delta)$$

$$\gamma = 3\Delta + 1 \tag{78}$$

Therefore, to include a horizontal parity into an existing vertical parity,

$$P = v_j + (3\Delta + 1)h_i \tag{79}$$

Figure 19 outlines this analysis.

Decomposing these two parities into their original forms in the presence of errors is straightforward. The vertical parity in this case should reflect the error imposed upon the column.

$$p_1 = [P \bmod (3\Delta + 1)] \bmod (\Delta + 1) \tag{80}$$

The horizontal parity should remain unaffected by errors. Using

$$p_2 = P \operatorname{div}(3\Delta + 1)$$

where $(a \operatorname{div} b)$ is the closest integer to a/b, can still be in error due to a possible carry: In general, $P + e \leq Q + 2\Delta$. If $P + e$ is larger than $Q + \gamma/2$, a carry error can occur: Since

$$Q + 2\Delta > Q + \frac{3\Delta + 1}{2} = Q + \frac{\gamma}{2}$$

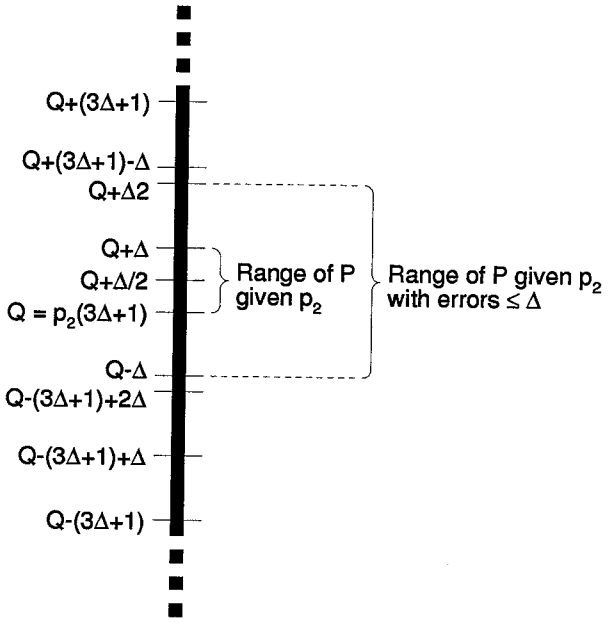

Figure 19: Including horizontal parities into existing vertical parities. Range of possible values for P given p_2 is shown, along with range possible with errors of magnitude $\leq \Delta$. To avoid overlap, $\gamma = 3\Delta + 1$.

an error can result.

Therefore, an offset μ must be used:

$$\mu + Q - \Delta > Q - \frac{3\Delta + 1}{2} \quad \text{— and —} \quad \mu + Q + 2\Delta < Q + \frac{3\Delta + 1}{2}$$

which becomes

$$\mu > -\frac{\Delta + 1}{2} \quad \text{— and —} \quad \mu < -\frac{\Delta - 1}{2}$$

Therefore, if μ is even, then

$$p_2 = \left[P - \frac{\Delta}{2}\right] \operatorname{div}(3\Delta + 1) \tag{81}$$

If μ is odd, then

$$p_2 = \left[P - \frac{\Delta + 1}{2}\right] \operatorname{div}(3\Delta + 1) \tag{82}$$

Equation (82) assures that, using the usual technique for rounding off fractional values (*i.e.* $1.4 \rightarrow 1$ and $1.5 \rightarrow 2$), p_2 is found without error:

$$\left[Q - \Delta - \frac{\Delta + 1}{2}\right] = Q - \frac{3\Delta + 1}{2}$$

and

$$\left[Q + 2\Delta - \frac{\Delta + 1}{2}\right] = Q + \frac{3\Delta + 1}{2} - 1$$

For vertical parities included into existing horizontal parities, the strategy is very similar. A vertical parity value ($p_2 = v_j$) is included into an existing horizontal parity value ($p_1 = h_i$) by

$$P = p_1 + \gamma p_2$$

where γ is some constant. Let $Q = \gamma p_2$. Assume p_2 is known.

$$Q \leq P < Q + \delta$$

or, equivalently,

$$Q \leq P \leq Q + 2\Delta$$

Now let an error of up to Δ in magnitude occur in P.

$$Q + e \leq P + e \leq Q + e + 2\Delta$$

which, in the worst case (maximum range the values can span), is

$$Q - \Delta \leq P + e \leq Q + \Delta + 2\Delta \tag{83}$$

Equation (83) is similar to equation (77); the former differs from the latter only by a constant Δ on the right far side. To be uniquely able to distinguish between the case where a carry, a borrow, or neither occurs, there must be an unambiguous mapping given that the worst case occurs. From equation (83), the maximum range which $P + e$ can span given p_2 is

$$Q - \Delta + \gamma - 1 = Q + 3\Delta$$

$$\gamma = 4\Delta + 1 \tag{84}$$

which is similar in form to equation (78). Therefore, to include a horizontal parity into an existing vertical parity,

$$P = h_i + (4\Delta + 1)v_j \tag{85}$$

Figure 20 outlines this analysis.

Decomposing these two parities into their original forms in the presence of errors is done as before. The horizontal parity in this case should reflect the error imposed upon the row.

$$p_1 = [P \bmod (4\Delta + 1)] \bmod (2\Delta + 1) \tag{86}$$

The vertical parity should remain unaffected by errors; again, an offset μ must be used:

$$\mu + Q - \Delta > Q - \frac{4\Delta + 1}{2} \qquad \text{— and —} \qquad \mu + Q + 3\Delta < Q + \frac{4\Delta + 1}{2}$$

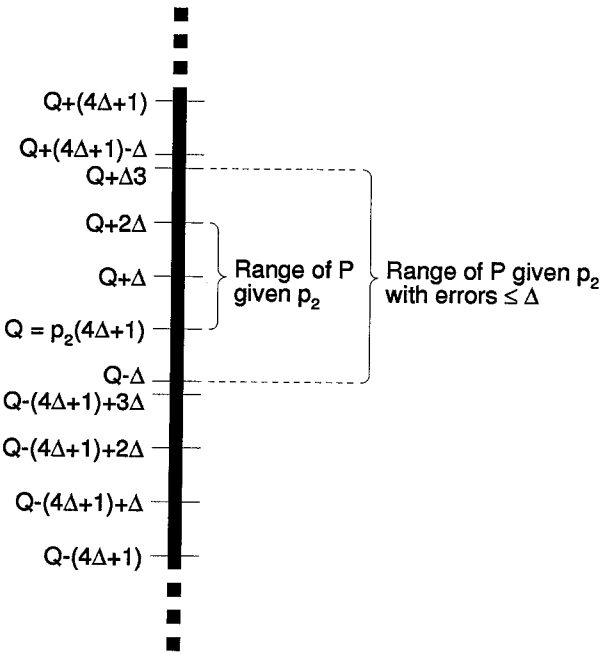

Figure 20: Including vertical parities into existing horizontal parities. Range of possible values for P given p_2 is shown, along with range possible with errors of magnitude $\leq \Delta$. To avoid overlap, $\gamma = 4\Delta + 1$.

which becomes

$$\mu > -\Delta + \frac{1}{2} \qquad \text{— and —} \qquad \mu < -\Delta - \frac{1}{2}$$

Therefore, $\mu = \Delta$, and

$$p_2 = [P - \Delta] \operatorname{div} (4\Delta + 1) \tag{87}$$

3.5.3. Actual Compression Technique

When this codeword is actually implemented, it may not be efficient to attempt integer division and remainder (modulo) calculations. High speed applications require the code to have as few operations as possible for encoding and decoding, and division operations are time intensive. Therefore, placing parity values into bit fields within the parity symbol may be advantageous: In this format, examining the bit fields is sufficient to obtain the necessary parity values.

The best choice for Δ is $2^m - 1$; the vertical parities are taken modulo 2^m and the horizontal parities are taken modulo 2^{m+1}. The range of the horizontal parities is larger than necessary, but 2^{m+1} is used since it allows easy derivation of the modulo $\Delta + 1$ parity value for determining error location: The m least significant bits of the horizontal parity value provide this parity information.

To store more than one parity value into each symbol of redundancy, the strategy used in

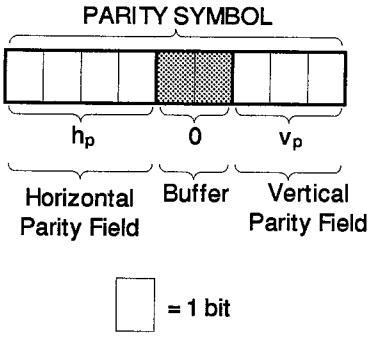

Figure 21: Horizontal parity cell compression. Compression into bit fields, with horizontal parity included into vertical parity symbol. One parity symbol is shown, with parity values encoded into bits. Blank space of two bits is safety buffer to avoid carry-over errors into horizontal parity field.

the previous subsection is applied. If horizontal parities are being stored in vertical parity symbols, then the minimum γ is $3\Delta + 1$; this translates into at least $m + 2$ bits, or a buffer of two blank unused bits between the vertical and horizontal parity value bit fields. (See Figure 21.) These bit fields serve as flags as to when a carry or a borrow had occurred. Correction for carries and borrows is as follows:

buffer bitfield = 00: No adjustments needed
buffer bitfield = 01: No adjustments needed
buffer bitfield = 10: This case will never occur
buffer bitfield = 11: Add 1 to horizontal parity field

If vertical parities are being stored in horizontal parity symbols, then the minimum γ is $4\Delta + 1$; this translates into at least $m + 2$ bits, or a buffer of one blank unused bit between the horizontal and vertical parity value bit fields. (See Figure 22.) This one extra bit with additional information obtained from the value in the horizontal parity bit field provides carry or borrow information. Because the horizontal parity is between 0 and $2^{m+1} - 1$, inclusive, an error can at most only increase this value to $2^{m+1} + 2^m - 2$, which is represented by a carry of 1 and a residue of $2^m - 2$. Likewise, an error can only decrease this value to $-\Delta$, which is represented by a borrow of 1 and a residue of $2^m + 1$. Therefore, if a carry occurs and the residue is greater than or equal to $2^m + 1$, or $\Delta + 2$, then a borrow has occurred; if a carry occurs and the residue is less than or equal to $2^m - 2$, or $\Delta - 1$, then a carry has occurred. Therefore, when the buffer is nonzero, if the most significant bit of the horizontal bit field is 1, then a borrow has occurred; otherwise, a carry has occurred.

buffer bitfield = 0: No adjustments needed
buffer bitfield = 1 and MSB of horizontal parity field = 0: No adjustments needed
buffer bitfield = 1 and MSB of horizontal parity field = 1: Add 1 to vertical parity field

Note that for parity values which are inserted into existing parity symbols, there is no need

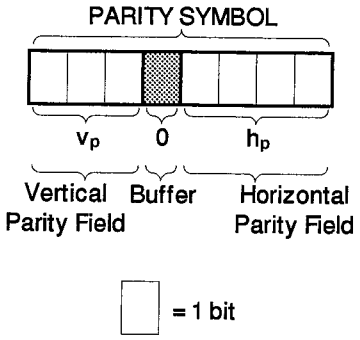

Figure 22: Vertical parity cell compression. Compression into bit fields, with vertical parity included into horizontal parity symbol. One parity symbol is shown, with parity values encoded into bits. Blank space of one bit is safety buffer to avoid carry-over errors into vertical parity field.

for contiguous bit fields to be used. Therefore, these inserted parity values may be placed in any order to fit them in the available space. However, even if noncontiguous bit field packing is used, the magnitude of approximate errors correctable using this implementation is less than that possible by using the algorithm outlined in the previous subsection. The tradeoff made here is ease in encoding and decoding in favor of optimality of parity symbol usage. Thus, the horizontal and vertical parities are recorded in fewer symbols but in a manner which allows easy determination of parity values without integer division or remainder (modulo) calculations, resulting in an easily encodable and decodable code which has very high rate and can correct a phased burst of approximate errors.

3.5.4. Example

Consider two codewords, both capable of correcting a single phased burst of length 9 and approximate errors of size $\Delta = 7$ from symbols of size $q = 512$. The first is the standard phased burst array codeword of dimensions $n_1 = 9$, $n_2 = 11$; the second, the codeword with parity compression derived from the standard codeword of dimensions $n_1 = 10$, $n_2 = 11$. The latter is effectively a 9×11 codeword with 90 information symbols capable of correcting a burst of length 9 and of size $\Delta = 7$. The first has a rate of 0.81, with 80 information symbols, and is capable of correcting a single phased burst of length 9 and of any error magnitude. The second has a rate of 0.91 with 90 information symbols and is capable of correcting a single phased burst of length 9 so long as the error magnitude of each symbol does not exceed 7. (See Figure 23.)

Vertical parity values are placed in each horizontal parity symbol using every possible space; therefore, the four least significant bits are used to store the horizontal parities, while the most significant four bits are used to store the vertical parity values. The fifth most significant bit, in

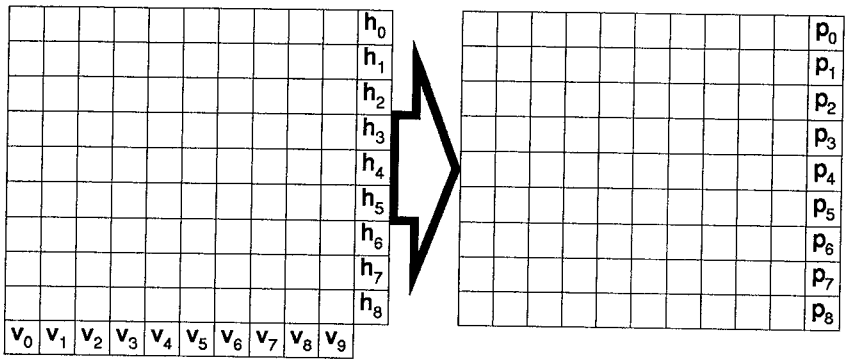

Figure 23: Actual parity cell compression. Codeword on left is the standard 10×11 phased burst correcting array code; codeword on right is the 9×11 phased burst correcting array code correcting approximate errors of $\Delta \leq 7$ with parity cell compression. Parity values are compressed into fewer symbols.

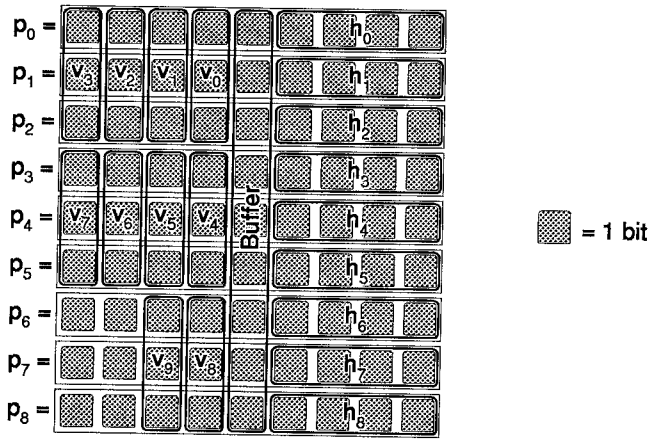

Figure 24: Parity symbols shown bitwise. Each parity symbol is composed of 9 bits (for a total of 512 possible values per symbol). Bit mapping for each parity value is shown.

between the two bit fields, is used as a buffer. The vertical parity values are stored across the parity symbols such that four bits each from nine symbols are used to store 10 three bit numbers. One possible arrangement is shown in Figure 24; a total of six bits remain unused. (Note that a more optimal utilization of available parity symbol space would have resulted in $\Delta = 8$ but at the cost of increased computational complexity.)

3.6. Conclusions

Various aspects of the single phased burst error correcting array code have been explored. The general solution for valid codeword sizes to correct phased bursts of length n_1 was presented.

The code can have dimensions such that the Singleton bound is met with equality and is therefore optimal. The minimum size for n_2, given n_1, was shown to be the next highest prime larger than n_1 for $n_2 < 10^7$; it is an unsolved number theory problem to prove this true for all n_1 and n_2. Two decoding algorithms were covered. Also, encoding and decoding strategies for approximate errors were presented, with means of compressing parities to further increase the rate of these codes.

CHAPTER 4
IMPROVEMENTS TO A LOCALLY ADAPTIVE VECTOR QUANTIZATION ALGORITHM FOR IMAGE COMPRESSION

4.1. Introduction

Storage of information in image form provides the advantages of fast storage and retrieval from distant locations, application of analysis tools, and dense storage capacity. Furthermore, since more and more images are recorded digitally, storage in digital media is easily accomplished. However, this storage and retrieval of a large number of images can become cumbersome and inefficient. Image quality must be high to maintain accuracy of records, therefore necessitating many bytes for each image; however, storage space is limited and transmitting large images over communications links which are already at or near maximum capacity is time-consuming. Therefore, some means of compressing the images is required.

Image compression relies on removing the redundancies within an image to reduce the data necessary to represent a given image. These redundancies may be positional or statistical. The representation can be lossless (the original can be reconstructed from the compressed version) or lossy (an approximation to the original can be constructed from the compressed version). Furthermore, for lossy compression, the compressed file size can be fixed (and therefore allowing the distortion to vary from image to image) or a maximum distortion threshold can be set (and therefore allowing the compressed file size to vary from image to image). Note that the latter of the two, fixing distortion and allowing file size to vary, is preferable in most situations: A high quality image is usually desired, and packet-based communication systems can accommodate varying image sizes.

Vector quantization algorithms (VQ) compress images by reducing the statistical redundancy of an image: The image is blocked into vectors with components derived from the pixels and is then substituted by an index which represents a vector which is similar to the original. VQ has the advantage of higher compression rate for a given distortion than scalar quantization by this grouping

of pixel data. One method for applying VQ to images is presented in [44]; this algorithm is known as the Linde-Buzo-Gray (LBG) algorithm. It uses a generalized Lloyd's algorithm to generate the code book.

There are three issues regarding VQ coding for images. First, the method of code book design must be addressed. Large code books can code images with less distortion at the cost of higher rates, both to send the indices and to send the code book. In some applications the code book is sent once for a group of images or not at all (a standardized code book is assumed); in this work the worst case is assumed in that the code book is sent explicitly for each image. Small code books offer higher rate, but since there are fewer codewords to represent the vectors in the image, the distortion is much higher. In addition, smaller code books must be generated carefully to best represent the vector space spanned by the image blocks. This code book generation process is computationally intensive and is not fit for high speed applications.

Second, once the code book is generated, each vector in the image must be mapped to a given codeword. This is usually done by finding the codeword which gives the least mean squared difference between the vector and codeword, also a computationally intensive task. Various algorithms have been developed to reduce the time taken to search for the appropriate codeword.

Third, because the codeword which best approximates the image vector is used, often the block boundaries are visible. This effect (blockiness) is significant when distortion is large and rate is low (thus when the image is compressed to a small size), causing a staircase or sawtooth effect. It also occurs around edges and regions of high detail, where these edges are not represented accurately and where small details are washed out.

Many adaptations and improvements have been made to the basic concept of vector quantization to remove these three difficulties while improving or at least not degrading significantly its rate-distortion performance. There are a number of review articles which summarize these developments [45-47]. Some developments are covered below.

Standard VQ algorithms are memoryless; that is, they do not take into account positional information from vector to vector. Higher rates are possible if such information is used. The technique applied to VQ algorithms is analogous to differentially pulse coded modulation (DPCM) and trellis coding, where the state of the coder, dictated by previously coded blocks, affects the codewords or determines which of several code books are used in coding. Thus, effectively smaller code books are used, resulting in improved rate-distortion and speed performance. This is called Finite State VQ (FSVQ) [48-54]. The shortfall of this technique lies in the design of the finite state

machine: If the image has features which the finite state machine is not equipped to handle, the code book can be affected so as to increase distortion.

Another way to improve code book design is grouping the blocks into different classes. By assigning special classes to blank regions, high detail regions, edges, and other features, the code book can be designed to accommodate these regions and represent them more accurately. The code book is split into several smaller code books to represent each type of feature. This is called classified VQ (CVQ) [55-58]. VQ algorithms of this type can record edges and details better than standard VQ algorithms; they trade this capability off for increased complexity or degradation in rate-distortion performance.

Various means for generating a code book with lower complexity than the generalized Lloyd's algorithm described by [44] have been investigated. These include codeword merging [59] and codeword splitting [60] algorithms. These algorithms trade off lower rate-distortion performance for higher speed.

Algorithms which shorten codeword search times can also be used to speed up code book generation. These algorithms share the same concept: The distances between the vector being coded and the codewords are computed partially at first (typically by finding the mean value); these partial distances are used as a sieve to remove codewords which are too far away. A more thorough distance calculation is done on the remaining more likely codeword candidates [61-68].

Generating tree structures for code books is another way in which fast codeword searching can be accomplished; these algorithms are known as tree search VQ (TSVQ). Searching a tree, which can be done very quickly, accomplishes the coding. These algorithms, however, suffer from slight degradation of rate-distortion performance since the resulting code books are often suboptimal. Furthermore, the trees are not simple to construct; they must often be grown and then pruned to be usable. This process can be computationally intensive [69-71].

Along with classified VQ, there are several other ways to remove blockiness and edge effects from a coded image. One way is to adapt the block size and shape. By allocating smaller blocks to regions with more detail, a more accurate representation of the image can result. Furthermore, low detail regions are coded with larger blocks to improve rate-distortion performance. At issue is how to segment the image into as many large blocks as possible [71-77]. The obvious tradeoff, therefore, is increased complexity (in determining the block map) for improved subjective performance.

Another way to remove blocking effects is to filter the image. Both pre-emphasis and post-deemphasis can be used. Filters can be used to attenuate the spatial block frequencies which cause

the blockiness. To avoid removing actual image components, these frequencies are amplified prior to encoding with the VQ algorithm; after decoding, the block spatial frequencies are attenuated [46]. The problem with this method, of course, is that attenuation strong enough to filter the blockiness effectively results in an excessively blurred image. This can be avoided by a preprocessing algorithm which takes into account the limitations and characteristics of the human eye; the inverse map is imposed at the decoder [78]. This second approach is similar to one proposed in [79].

An approach used to improve performance for low-detail images is block truncation coding (BTC). Here, blocks are assumed to take on only two values determined by a mean value and a step size. All pixels within the block are assumed to take values one step above or below the block; the mean and step size are determined separately for each block. BTC is not very effective for high detail images [80-83].

Coding for image sequences comes naturally as an extension to image compression. With the exception of those algorithms specifically designed for coding video images, these algorithms rely on shared code books which are either designed for the entire sequence [84] or are updated for each image [71,85-87]. These latter algorithms are adaptive across frames in that the code book is altered to accommodate the source statistics.

However, they are not directly suitable for coding single frames adaptively. Other methods for adaptive image compression exist; these include an algorithm based on DPCM [88] and on artificial partitioning of the image into subimages to apply frame adaptive methods [89]. Other methods include adaptive code book reorganization to split classes with the greatest dispersion [90] and separating each vector into low, medium, or high activity regions to be coded separately, in a manner similar to classified VQ [58]. In addition, there is the algorithm under study here, the locally adaptive vector quantization (LAVQ) algorithm [91].

In this chapter various improvements to the locally adaptive vector quantization algorithm will be explored. Section 4.2 covers the basic algorithm, where its advantages and limitations will be discussed. Subsequent sections deal with removing the algorithm's shortfalls without compromising its advantages. In Section 4.3 the fast searching algorithms are incorporated. In Section 4.4 methods to decrease the number of bits used to represent the codeword values are considered; Section 4.5 does the same for the code book indices. Section 4.6 covers a difference coding method. In Section 4.7 scanning methods are explored. Sections 4.8 and 4.9 cover postprocessing techniques to improve image quality. In Section 4.10, all of these improvements are considered to obtain the best case performance for LAVQ.

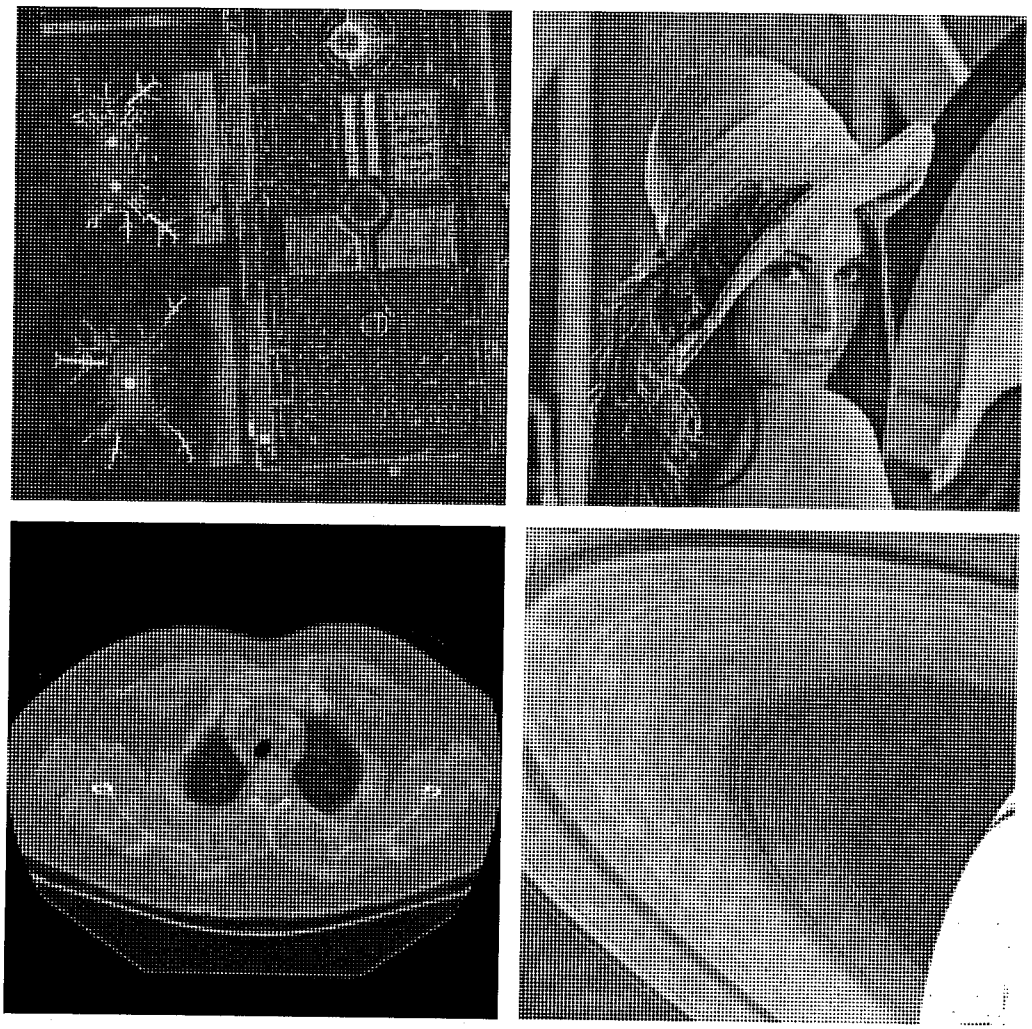

Figure 25: Images used in this work. Clockwise from upper left, *lax*, *lenna*, *saturn1*, and *med01*.

Four test images will be used in the analysis. These four were selected to give a varied selection of possible image types and characteristics; they are all 512×512 pixel, 8 bit/pixel monochrome images. They are an overhead view of part of Los Angeles International Airport (*lax*), a portrait (*lenna*, also known as *womanhat*), a medical image (*med01*), and a planetary image (*saturn1*). (See Figure 25.) For brevity, only the results of processing the *lenna* image will be presented for each individual improvement. However, all four images were processed for each improvement for confirmation, and the best case results will be presented for these four images.

4.2. Basic LAVQ Algorithm

The basic locally adaptive vector quantization (LAVQ) algorithm provides a simple yet effective one-pass image compression strategy. The encoder has a code book containing codewords (vectors) where the index of the codeword corresponds to its position in the code book. A block is taken from the image and compared to the stored codewords; if there exists a codeword sufficiently close to the image block (within the error allowance) the index itself is sent, and that codeword is moved to the top of the code book. (Refer to Figure 26.) If no such codeword exists, a special index is sent; this index is followed by the block itself. This block becomes a new codeword and is placed at the top of the code book. All other codewords are pushed down, and if the number of codewords exceeds the maximum allowed, the last codeword is lost. (Refer to Figure 27.) Typically, the code book is initially empty or full from the previous image encoded. For all results discussed in this work, the code book will be initially empty.

On the decoder side, the decoder expects an index. If this index is the special one denoting that a new block was sent, the receiver expects a block to be received immediately following; this block is placed at the top of the code book and all other codewords are pushed down. If the code book were already full, the last codeword is discarded. This new block is also placed into the image being built by the decoder. (See Figure 27.) If the index is not one designating a new block, then the codeword corresponding to the index is put into the image being built, and that codeword is moved to the top of the code book. (See Figure 26.) Thus, if the encoder and decoder start with the same code book, they will have the same code book at each step, and the image will be successfully sent [91-95].

The LAVQ strategy maintains the most recently used vectors in the code book in order of last usage; this allows the algorithm to quickly and efficiently code any image without code book training: The algorithm needs only one raster-scanned pass of the image to code it entirely. In serial implementation, LAVQ has time complexity $O(nm)$ and space complexity $O(m)$, where n is the number of pixels in the image and m is the number of codewords in the code book. Furthermore, LAVQ is capable of adapting to the local image statistics to preserve details: New codewords are generated more often in regions containing edges and fine features while blank regions are coded with fewer new codewords. This feature-preservation property, along with its one pass, high speed code book generation and encoding property are the two main advantages of LAVQ.

However, the basic LAVQ algorithm has shortcomings which arise from precisely these two

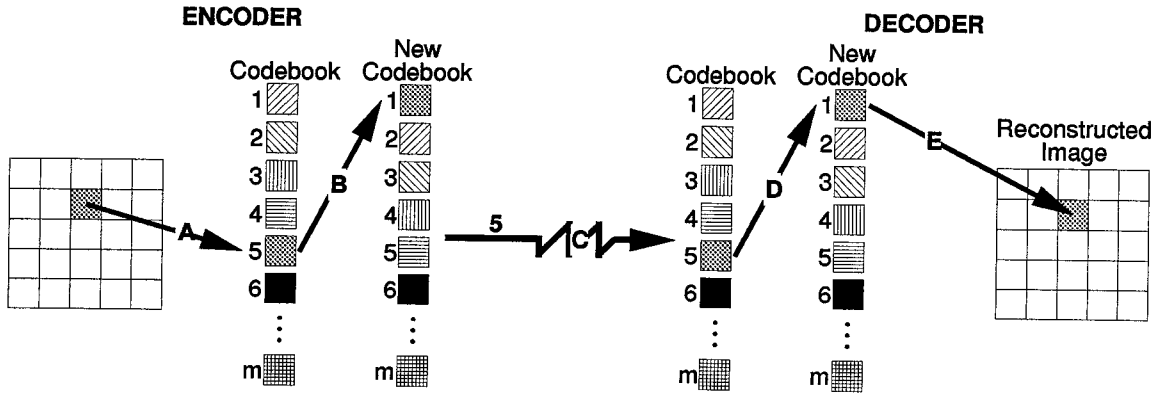

Figure 26: LAVQ with existing codeword. Encoder and decoder operation in the case where a codeword which approximates the input exists in the code book. An image block is compared to the entries in the code book and a match is determined (A). That codeword is moved to the top of the code book (B) and the index is transmitted (C). On the receiver side, the index is received and the corresponding codeword (D) is moved to the top of the code book and the block information is inserted into the reconstructed image (E).

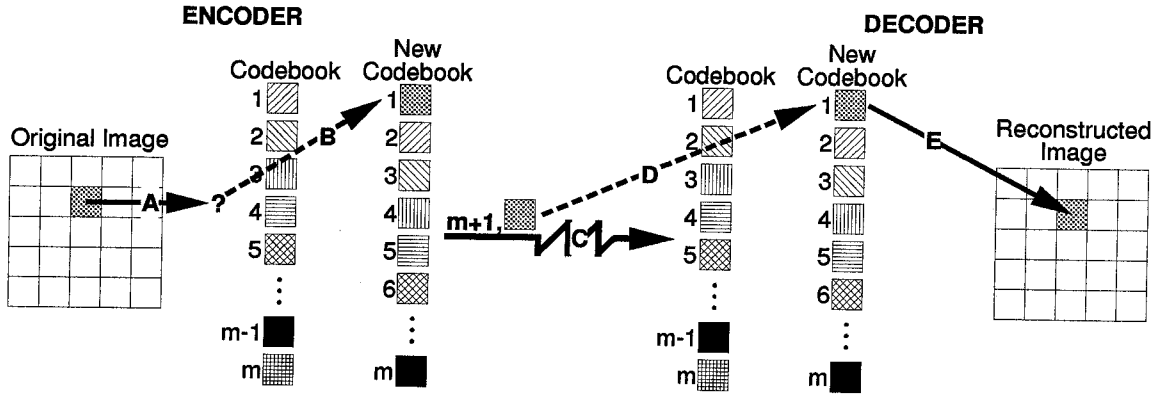

Figure 27: LAVQ without existing codeword. Encoder and decoder operation in the case where a codeword which approximates the input does not exist in the code book. An image block is compared to the entries in the code book and no match is found (A). The original image block is inserted at the top of the code book (B) and the index $m+1$ is transmitted, followed by the original block (C). On the decoder side, the special index $m+1$ is received (D); the raw data which is to be a new codeword is received immediately following. This block is placed at the top of the code book and in the reconstructed image (E).

advantages. First, because the code book is generated on-the-fly, it is suboptimal. This makes the rate-distortion performance of LAVQ inferior to algorithms such as LBG which spend more time optimizing the code book for lower distortion. Second, while detail regions are coded accurately with LAVQ, relatively constant regions are not. Because codewords are not optimized for the regions which they represent, the block boundaries are more easily visible. LAVQ, therefore, trades off speed and detail preservation for rate-distortion performance and smooth representation of constant regions. Various means to remove these deficiencies will be explored in subsequent sections.

4.2.1. An Analogy to DPCM

The LAVQ algorithm has similarities to the differential pulse coded modulation (DPCM) algorithm. In scalar or standard DPCM, a prediction is made of the present pixel based on neighboring pixels or other means. The difference between this prediction and the actual pixel value is then compressed, typically through quantization or an entropy coder. In vector DPCM, this prediction, difference, and coding procedure is conducted on a vector or block of pixels. The issue here is that for high dimensional vectors, the complexity of prediction and coding can become computationally intensive.

There is a great degree of similarity between LAVQ and vector DPCM in all three stages of prediction, difference, and coding. In vector DPCM, the prediction is based on spatially adjacent vectors, typically those which have been processed previously (and therefore the algorithm is causal). More than one vector can be used, and the prediction function can be quite complex. For LAVQ, the prediction is also based on previously encountered vectors; however, the prediction is based solely on one of those vectors, the closest matching vector in the code book.

The difference measure for vector DPCM is typically a vector of the per-pixel difference of the predicted and actual vectors. Thus, it is a direct measure of the accuracy of prediction in that the actual amount of error is determined. LAVQ, in contrast, finds the codeword which is closest to the input vector and only outputs the closest match. If no such match exists to within a preset error allowance, then the input vector is also output. Therefore, vector DPCM is analogous to an analog system, where a measure of error is output; LAVQ is analogous to a digital system, where either a close enough match is or is not found.

Coding for the vector DPCM is quantization and/or compression of the resulting error vector. Quantization is typically done on a per-pixel basis. Entropy codes used can include Huffman, arithmetic, and Lempel-Ziv codes. Likewise, coding for LAVQ also includes a per-pixel coding of the vector data when no close enough match exists; also, the indices of codewords used when a close enough match exists can be coded in a similar manner. Thus, the LAVQ algorithm contains many similarities to the vector DPCM algorithm; the LAVQ algorithm can be considered a special case of vector DPCM.

4.2.2. Experimental Results

Figure 28 shows a sample of the experimental results of software implementation of the basic LAVQ algorithm. In this figure, the image *lenna* is processed with the basic LAVQ algorithm and

the LBG algorithm; the absolute difference per pixel between the original image and the processed image (error image) is shown. The LBG algorithm was applied with parameters of 4×4 pixel blocks and 256 entry code books for a rate of 0.625 bit/pixel. To obtain the same mean squared error distortion using the basic LAVQ algorithm with 1×8 pixel blocks and 255 entry code book, 1.33 bits/pixel are needed.

The mean square error distortion measure is actually a normalized mean squared error per pixel, given by

$$MSE = \frac{1}{N_x N_y} \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} \left[\frac{p_{ij} - \hat{p}_{ij}}{255} \right]^2$$

for the monochrome 8-bit images used in this work. N_x and N_y are the dimensions of the image in number of pixels in the x and y direction, respectively; the original image is represented by pixels p_{ij} while the processed image is represented by pixels $\hat{p}_{ij}$.

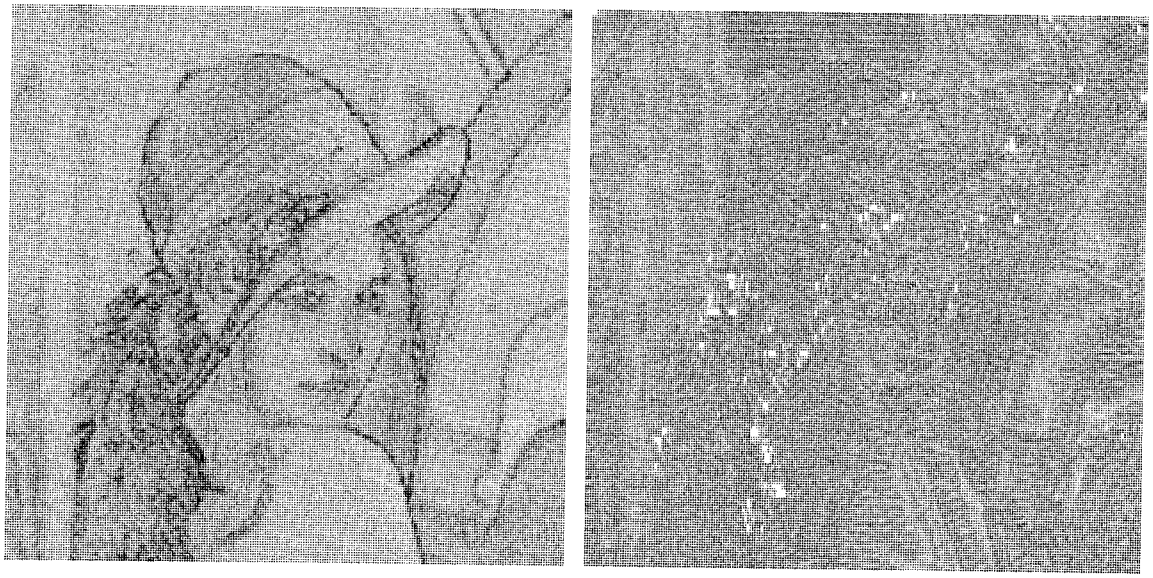

Figure 28: Error images, LAVQ and LBG. Basic LAVQ algorithm (error allowance of 0.0479, 1×8 pixel blocks, and 255 entry code book) and LBG compression (4×4 pixel blocks and 256 entry code book) applied to the *lenna* image. Left image shows error after processing with LBG algorithm; LBG obtained 0.625 bit/pixel compression at a pixel-wise mean squared error of 6.744×10^{-4}. Right image shows error after processing with basic LAVQ algorithm; to obtain a comparable error 6.700×10^{-4}, a rate of 1.33 bits/pixel is needed.

Figure 29 is a detail of the *lenna* image; here, enlarged portions of the right edge of the hat brim are shown for the original image, image processed with the basic LAVQ algorithm, and the image processed with the LBG algorithm. The LAVQ algorithm represents detailed areas and edges accurately, but it renders smooth, low detail areas with highly visible distortion. The LBG algorithm, by contrast, renders these smooth regions without much distortion; it represents high

detail areas and edges with poor quality. This effect can be seen readily in the error image (Figure 28) and in the enlarged figures.

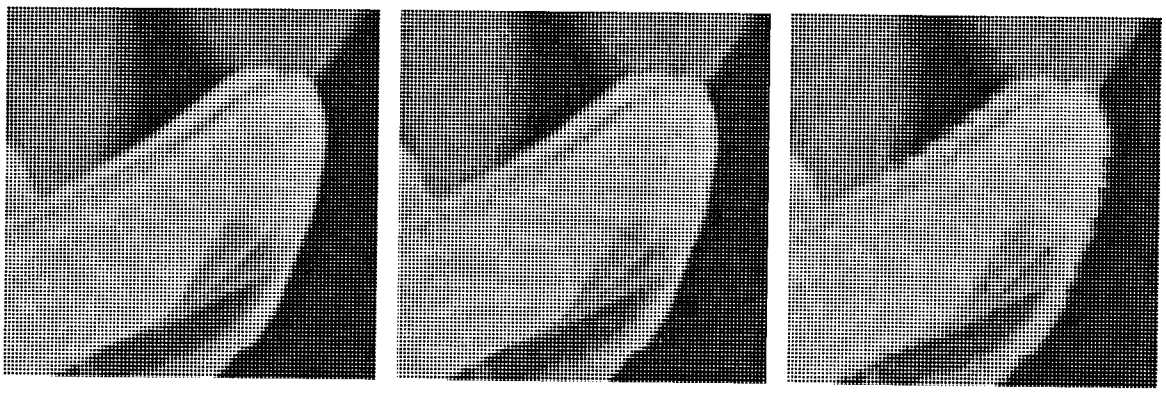

Figure 29: Basic LAVQ detail. Detail of upper right edge of hat brim from the *lenna* image. From left to right: original, basic LAVQ compression, and LBG compression.

The rate-distortion characteristics of the basic LAVQ algorithm on the *lenna* image are compared with the performance of the best case (best block and code book size) LBG algorithm in Figure 30. The basic LAVQ algorithm's performance is worse than that of the LBG. Its performance comes closer to that of the LBG algorithm for low distortion (high rate) compression than for high distortion (low rate) compression. This occurs because given the parameters used for LAVQ, an 8-pixel block is represented by an 8-bit index. These parameters limit performance to 1 bit/pixel. Changing the block and code book sizes can allow some degree of improvement; however, a similar rate limit will always exist. The LBG algorithm has no such limit and therefore can outperform the basic LAVQ algorithm.

One advantage of the basic LAVQ algorithm over the LBG algorithm is speed. The rate-distortion curves for both LAVQ and LBG algorithms were generated through simulation programs written in the C language on Sun Microsystems Sparcstation 1+ workstations. The LBG algorithm used speed enhancements such as partial distortion search and magnitude screening. Partial distortion search stops the squared error calculation when that partially computed error is larger than that of the best match found at that point. Magnitude screening disqualifies the codeword if the estimation of the distance, obtained by comparing the difference in magnitudes and the individual vector components between the codeword and the block being coded, exceeds that of the best match found at that point. The LAVQ algorithm used no such speed enhancements.

Despite this advantage which the LBG algorithm was given, the rate-distortion curve for the *lenna* image was produced through simulation which required over 200 hours of computation. In

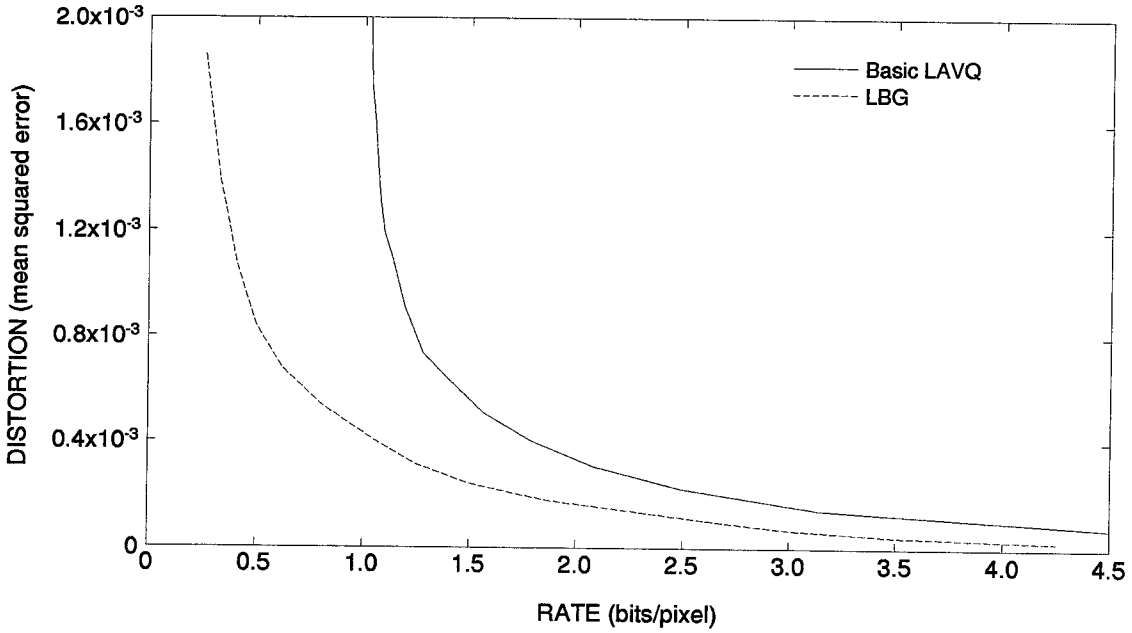

Figure 30: Basic LAVQ, rate-distortion curve. Rate-distortion curves for *lenna* image. Basic LAVQ algorithm used 1×8 pixel blocks and 255 codewords in code book; error allowance is varied to obtained curve. Best case LBG performance used various block and code book sizes (2×2, 2×4, and 4×4 pixel blocks and 16, 32, 64, 256, 512, 1024, 2048, 4096, and 8192 codeword books).

comparison, the curve for the basic LAVQ algorithm required less than 30 minutes of computation. Most of the time spent on encoding using the LAVQ algorithm is taken by finding the closest codeword in the code book and determining if that match is close enough. Rearranging the code book and sending the required index and possibly a new block can be done quickly in serial implementation using lookup tables, linked lists, and other software techniques. Other code book updating techniques can be used [96], but the lower complexity these methods offer is not significant. Methods used to shorten codeword search times for LBG-based algorithms can be applied to LAVQ as well.

4.3. Speed Improvements: Fast Codeword Searching

As mentioned in the previous section, one main advantage of the LAVQ algorithm over other VQ-based image compression algorithms is high encoding speed. Three methods useful in increasing speed of code book searching in serial implementations will be explored here. The first method involves the method of searching; the other two are taken from techniques developed for the LBG algorithm.

The original basic LAVQ algorithm searches the entire code book for the best match between

the vector (block of pixels) being encoded and the vector entries of the code book. Instead of looking for the best possible match in the code book, one can search for the first occurrence of a codeword which is "good enough," that is, within the error allowance. This is the partial code book search algorithm for LAVQ. This improvement was utilized in [93].

4.3.1. Partial Distortion Search

Partial distortion search methods are used in VQ compression strategies based on the LBG algorithm [62,65]. In these algorithms, the search is stopped if the minimum squared distance found up to that point is exceeded. Therefore, after each pixel-wise difference is squared and summed, this partial sum is compared to the minimum squared distance found up to that point. If that value is exceeded, comparison between that codeword and the vector is aborted, and the next codeword is compared. In this way, the squared distance will not be completely computed (for all pixels in the block) unless the minimum squared distance decreases.

This algorithm is most effective if those codewords which are close to the image vector are at the top of the code book, since this means that subsequent, more distant codewords will be only partially checked before being discarded as having too great a squared distance. For the LBG algorithm, the code book can be sorted to accomplish faster searching, as done in [61]; for LAVQ, however, since the most recently used codewords are at the top of the code book and since the most likely candidates for best matches of codewords are those that have occurred in the immediate past, the code book can be considered partially sorted since the most likely codewords are at the top.

4.3.2. Magnitude Screening

One more approach to limiting codeword searching time is similar to one implemented by [63] and [67]. In these papers, lower bounds for distances between two vectors are estimated based on the components of the vectors themselves. This is used to rule out obviously distant codewords and therefore limit the search to more likely candidates.

Magnitude screening is used to filter out obviously distant codewords through an estimation of distance using the magnitudes and individual vector components of the two vectors. If the difference of the magnitudes and all vector components are less than the error allowance, then the distance between the image vector and the codeword is computed in the conventional manner. This is done at the cost of originally computing the magnitudes of all blocks (which also means the codeword magnitudes are calculated, since the codewords are derived directly from the image

vectors) and computing the component-wise and magnitude differences for each codeword examined in the comparison process.

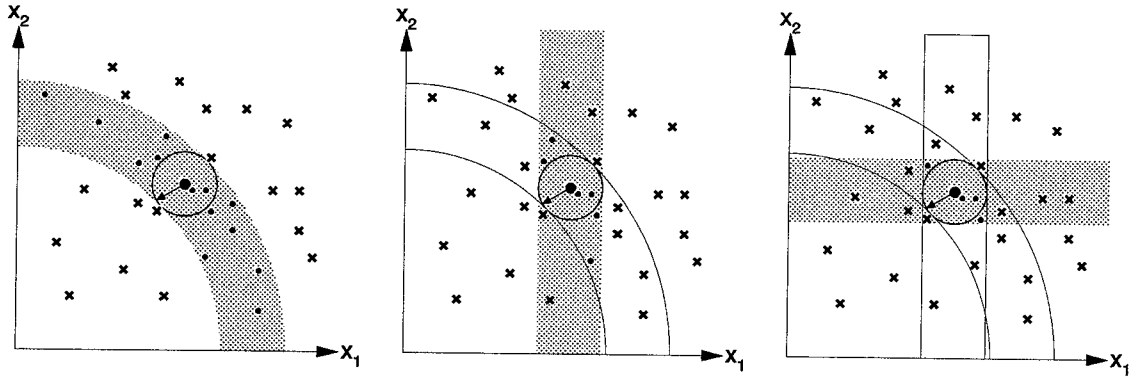

Figure 31: Magnitude screening for two-dimensional vectors. From left to right, eliminating unlikely codewords based on magnitude, based on vector component x_1, and based on vector component x_2. Codewords still considered are represented by dots; codewords eliminated as unlikely are represented by crosses. At each stage the search area is diminished further.

4.3.3. Experimental Results

The algorithms for codeword searching were applied to the *lenna* image; the implementation was checked for the number of pixel-wise squared difference operations (computational complexity), the average number of codewords checked before a match is declared (code book search depth), and the rate-distortion performance. In this analysis, there are several factors which must be noted prior to analysis. First, because the code book search depth is only dependent on the criteria for a match, *i.e.* best match or first match within error allowance, use of partial distortion search and magnitude screening has no effect on it. Thus, codeword search depth need be studied for the two cases of best match or first match codeword searching algorithms. Second, in the case that full search procedures are used, the relative computational complexity of the first match algorithm (in relation to the best match algorithm) is equivalent to its relative code book search depth. Therefore, studying the first match and best match algorithms gives an indication of the computational complexity of these two algorithms as well. Third, for magnitude screening, the complexity of the algorithm must include the initial determination of image block magnitudes.

Computational complexity and codeword search depth for four codeword search improvements are plotted in Figure 32. Here, the computational complexity and codeword search depth are relative to the basic LAVQ algorithm. The dashed curve denotes the relative code book search depth for the first match codeword searching algorithm as well as the relative computational complexity of

the full search, first match codeword searching algorithm. The dotted curve is the relative computational complexity of the partial distortion search, first match codeword searching algorithm; the solid curve is the relative computational complexity of the magnitude screened, partial distortion search, first match codeword searching algorithm; the dot-dashed curve represents the relative computational complexity of the magnitude screened, partial distortion search, best match codeword searching algorithm. All values are plotted against the compression rate.

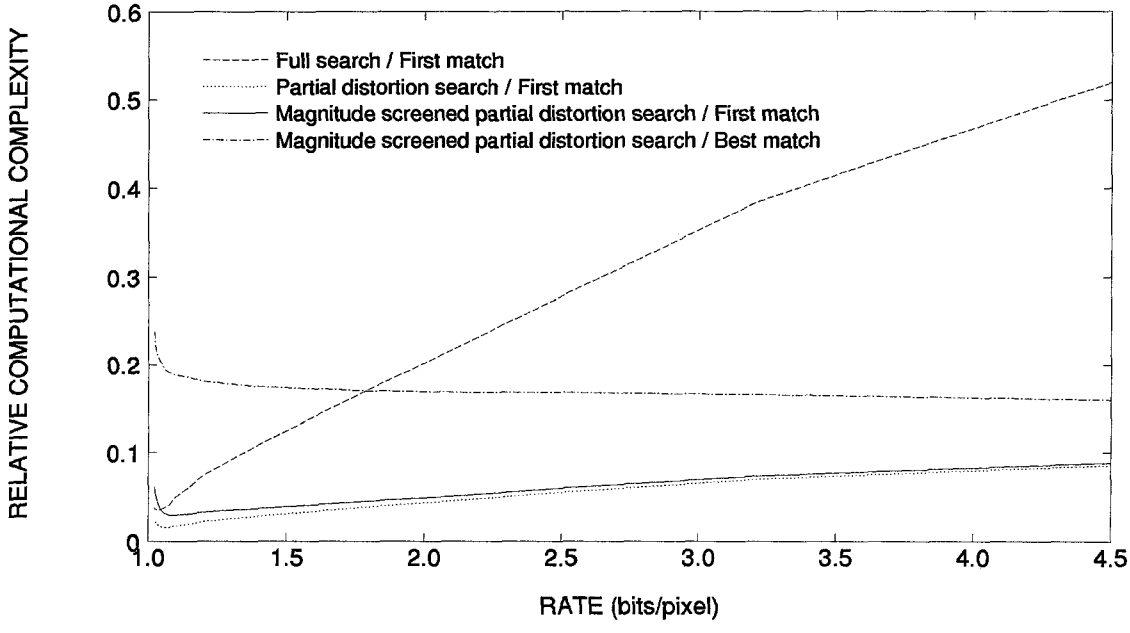

Figure 32: Searching algorithm complexity. Computational complexity of various codeword searching algorithms for the *lenna* image with respect to full search / best match algorithm.

The advantage obtained by first match codeword searching is significant; the number of computations (and the number of codewords checked) is less than half of the full search, best match algorithm for most of the range of compression rates. The tradeoff which may occur, and which will be examined later, is a decrease in rate-distortion performance. If partial distortion search is done, without any degradation in rate-distortion performance, the computational complexity can be decreased to about one-tenth of the full search, best match algorithm. Therefore, the speed of compression improves by roughly an order of magnitude with partial distortion searching. Adding in magnitude screening does not improve this performance; it is slightly detrimental, thus providing evidence that the partial distortion search, first match algorithm will be difficult to outperform.

The first match codeword searching algorithm affects the rate-distortion performance only slightly, while providing the advantage of increasing speed by an order of magnitude or more. A

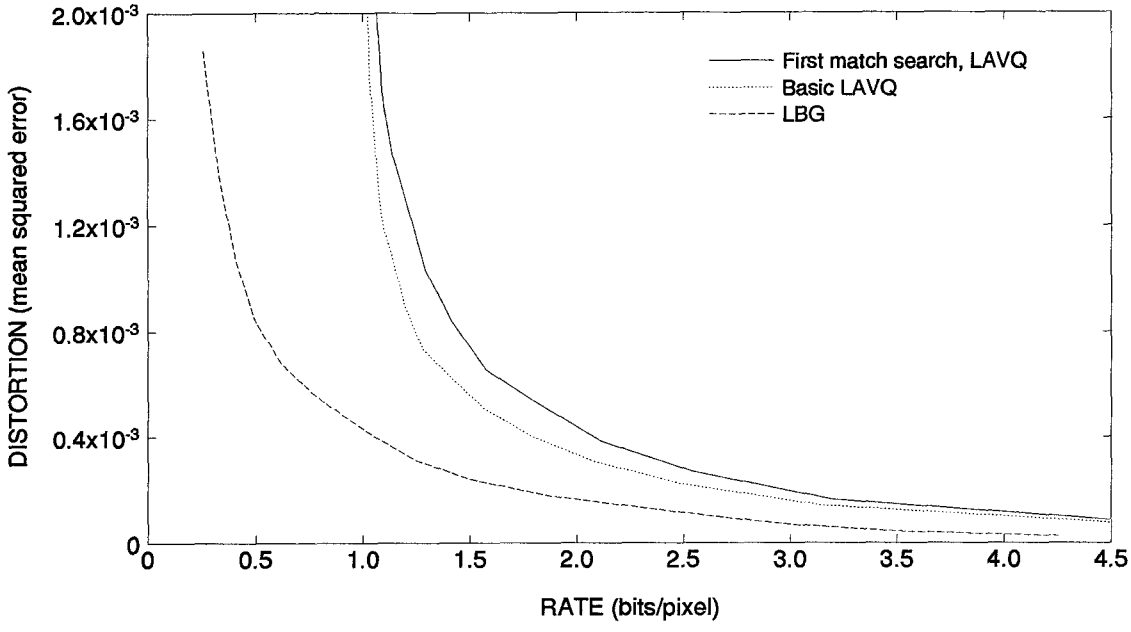

Figure 33: Searching algorithms, rate-distortion curve. Rate-distortion curves for various searching algorithms used on the *lenna* image. Parameters are identical to the curves in Figure 30.

magnitude screened partial distortion search, best match algorithm can still do respectably well in the low distortion (high rate) region; indeed, it does much better than the full search first match algorithm here. It requires, however, at least twice the computations of the partial search first match algorithm. The tradeoff, as mentioned earlier, is rate-distortion performance: Because a best match is not used, the performance can be expected to degrade for the first match algorithm. This degradation is small but nontrivial, as the rate-distortion curve in Figure 33 shows. Subsequent improvements can help offset this degradation, and the skewing of index usage toward lower values (which occurs since first match search will more likely than not output a low-valued index) will be advantageous for these improvements. Therefore, in subsequent sections of this work, the first match algorithms will be used.

4.4. Code Book Compression

For low distortion (high rate) image compression cases, the code book is a significant fraction of the total compressed image. This code book is derived from the vector data sent when no codeword matches the vector being coded within the error allowance; therefore, when the error allowance is low the resulting image will have low distortion and have many new codewords entered into the code

book. Rate-distortion performance can be improved by reducing the number of bits required to send this codeword data. The two approaches to this reduction addressed in this work are lossless data compression and bit stripping.

4.4.1. Entropy Coding

Lossless data compression, or entropy coding, of the codewords can yield better compression without image quality degradation. Compression is obtained by taking advantage of the frequency of occurrence of each pixel value and is especially useful for images with repetitive details: The vector may not accurately represent these details because the detail may not be of the right size for the vector or the detail may be oriented in a direction incompatible with the scanning direction. However, the pixel values themselves may repeat often; these statistics will be utilized by the lossless compression algorithm.

There are a number of considerations for data compression algorithms used in codeword compression. First, since high speed is an advantage of the LAVQ algorithm, this advantage cannot be negated by a slow compression algorithm. Second, when actually implemented, the data compression algorithm is best applied as a separate module. Thus, it should be a single-pass algorithm that requires no training time or buffer to store the complete, compressed image. Third, since the statistics of the pixel values vary with its location in the image, an adaptive algorithm which takes into account these local variations is necessary.

These considerations are served by the adaptive arithmetic and the Lempel-Ziv algorithms. The first takes advantage of local pixel statistics to compress the indices; the second uses spatial redundancies in pixel values. The compressed pixel values can be easily interspersed with the vector indices being sent to represent other blocks. No large buffers other than what the entropy coding algorithm requires for the coding process are needed at both the encoder and decoder. The order in which the indices and codeword data is transmitted will differ from that of the standard algorithm, but this will still be decodable at the receiver. The arithmetic coder used is similar to the adaptive algorithm presented in [97]. The code starts with the assumption that all symbols are equally likely; as symbols are used, the statistical model is updated to reflect the actual usage of symbols. The UNIX *compress* and *uncompress* commands were used for Lempel-Ziv compression.

4.4.2. Experimental Results of Entropy Coding

The improvement in rate-distortion performance with entropy coding of the code book is

not significant. The rate-distortion curve for the *lenna* image is presented in Figure 34. The adaptive arithmetic and Lempel-Ziv coders are compared against the LBG and basic LAVQ algorithms. The adaptive arithmetic coder gives a slight improvement over the uncoded case; the Lempel-Ziv coder, however, does worse. The adaptive arithmetic coder relies on statistical redundancies, not spatial redundancies like Lempel-Ziv coding, to achieve compression; since the code book pixels are not highly correlated spatially, the Lempel-Ziv coder does not perform well.

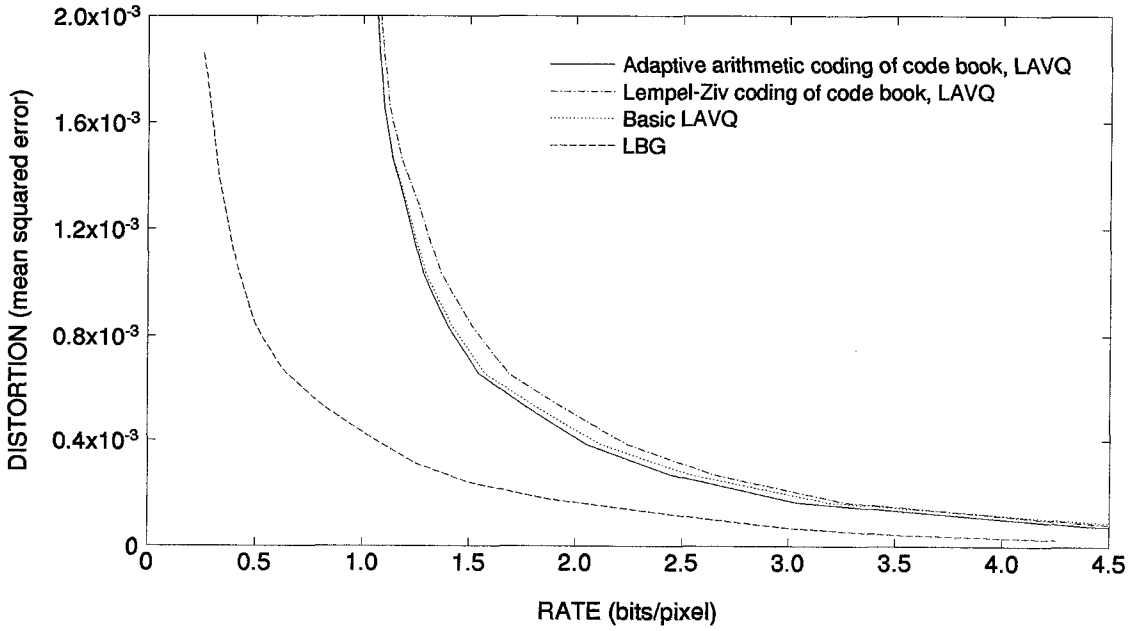

Figure 34: Entropy coded code book compression, rate-distortion curve. Rate-distortion curves for *lenna* image. LAVQ code books were compressed with an adaptive arithmetic coder and a Lempel-Ziv coder. All curves were generated using parameters identical to those used in Figure 30.

Limited improvement in performance is observed because the effect of entropy coding is to shift the rate-distortion curves slightly to the left. The shift is greater in the low distortion region, where the curve is practically horizontal, than in the high distortion region, where the curve is vertical and where the shift would be more apt to be visible. In the low distortion (high rate) region, there are many new codewords being sent; therefore, the entropy coders have large numbers of pixel values to compress, and a large fraction of the compressed file is composed of new codeword elements. However, at high distortion (low rate), the number of codewords being sent is limited, and adaptive entropy coders cannot perform as well with a small number of elements to code. Furthermore, since the majority of the compressed file size is composed of indices in this region, the relative advantage obtained by entropy coding of the codewords is not significant.

4.4.3. Bit Stripping

As mentioned in the previous section, for low distortion (high rate) image compression cases, the code book is a significant fraction of the total compressed image; this code book can be reduced in size through entropy coding, a method explored earlier, and bit stripping, which will be examined here.

Bit stripping refers to the process of removing the b least significant bits from each pixel and replacing them with random bits. This does not introduce significant distortion because the least significant bits also tend to be random for most images. In LAVQ, this bit stripping is done just prior to sending the data to the receiver; the random bits are introduced after the image is reconstructed. Note that though lossless compression is guaranteed to maintain the original distortion at the cost of uncertain compression, bit stripping guarantees a certain advantage in rate with a corresponding unknown potential change in distortion. Bit stripping operates under the assumption that the least significant bits of any image are random. Thus, if the two least significant bits are stripped off the new codewords, then, for 8-bit monochrome images, this entails a potential reduction in the compressed image by 6/8, or an improvement of 25%. The algorithm promises a simple means for reducing file size.

However, there are several disadvantages to this method. First, in actual implementation, the advantage of bit stripping is reduced by the overhead required in sending index values for each vector. Furthermore, since random bits replace the least significant bits of each pixel, there is potential for increased error. This error can be as great as twice the error allowance originally specified during coding. These combined disadvantages may reduce or eliminate the advantage of improved rate; therefore, bit stripping may not be an effective solution.

4.4.4. Experimental Results of Bit Stripping

The limitations of bit stripping as discussed in the previous subsection are visible in the rate-distortion performance of the *lenna* image in Figure 35; this figure shows the effects of stripping one, two, three, and four bits off the new codewords sent. Stripping one or two bits leads to a slight improvement in rate, but stripping more bits leads to an increase in distortion. The improvement in performance is small because bit stripping shifts the rate-distortion curves slightly to the left while shifting the curves upward (decreasing rate while increasing distortion). The leftward shift is greater in the low distortion region, where there are more codeword elements from which to strip bits; however, this also leads to an increase in distortion in a region where distortion is already low.

The resulting increase in distortion can be significant. In the high distortion (low rate) region, the number of codewords being sent is small, so the leftward shift obtained by stripping bits is small; the upward shift (increase in distortion) is large.

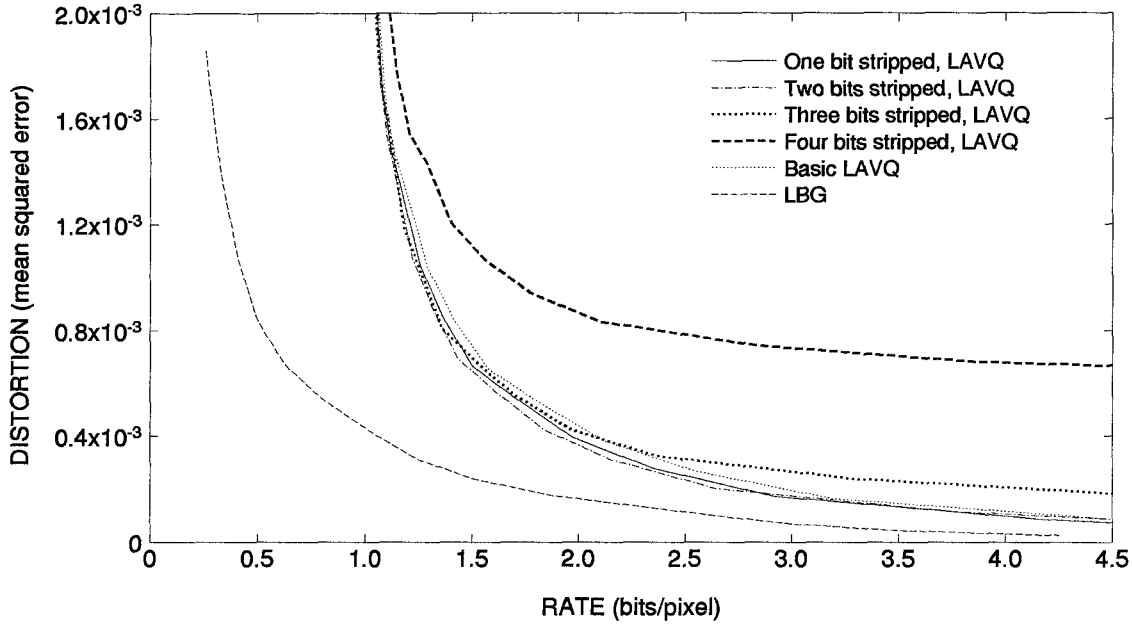

Figure 35: Bit stripping, rate-distortion curves. Rate-distortion curves for *lenna* image. LAVQ code books had their least significant bits stripped off. All curves were generated using parameters identical to those used in Figure 30.

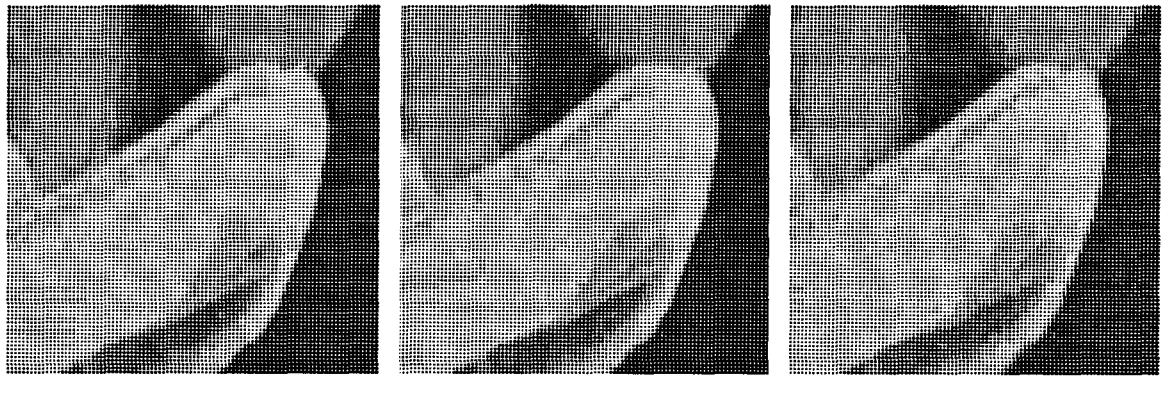

Figure 36: Bit stripping detail. Detail of upper right edge of hat brim from the *lenna* image using the basic LAVQ algorithm. From left to right: zero, two, and four bits stripped off the new codewords.

An additional effect of bit stripping is the reduction of blockiness in the processed image. Since the least siginificant bits are replaced by random bits at the decoder, the staircase effect resulting from having quantized blocks of pixels is reduced; block boundaries become less visible.

Figure 36 shows enlarged portions of the *lenna* image (detail of the right edge of the hat brim); these are, from left to right, the results of processing with the basic algorithm with zero, two, and four bits stripped off each new codeword. Some block boundaries are less visible when more bits are stripped off; the cost of this is increased distortion which is visible as a more grainy image and more visible block boundaries in different areas.

4.5. Index Compression: Entropy Coding

A byproduct of choosing the first codeword within the error allowance (instead of finding the best match within the code book) is that the frequency of use of low-valued indices is much higher than that of high-valued indices. For example, the indices 1, 2, 3, *etc.,* are expected to be used more often than 64, 65, 66, *etc.* Furthermore, if the images have regions of similar pixel intensity (that is, they are spatially correlated), then index use will also have spatial correlation. This suggests the use of lossless data compression algorithms on the indices to reduce size without affecting index data, just as in the case of code book data compression explored in the previous section. The codes used can exploit nonuniform index frequency statistics (such as the case with Huffman and arithmetic coders) or spatial redundancies (such as the case with Lempel-Ziv type algorithms).

There are a number of considerations for data compression algorithms used in this application; these are the same as those considerations explored for code book data compression; they were the necessity for algorithms that are fast, single-pass, and adaptive.

As before, these considerations are served by the adaptive arithmetic and the Lempel-Ziv algorithms. When implemented, these algorithms can intersperse the indices with the required new codeword data in one bit stream, negating the use of input and output buffers for the encoder; therefore, no buffers other than those needed for the entropy coding process are needed at both the transmitter and receiver. The arithmetic coder used is a variation of the adaptive algorithm presented in [97]; the UNIX *compress* and *uncompress* commands were used for Lempel-Ziv compression.

The adaptive arithmetic coder in [97] started with the assumption that all symbols are equally likely; as symbols are used, the statistical model is updated to reflect the actual usage of symbols. The Q-coder [98], based on this algorithm, uses a number of models to more accurately represent the characteristics of the source; it is effectively a state-dependent arithmetic coder since it switches models based on which symbols were used. This approach is used for this work: A total of five models are used; the coder switches models depending on the last symbol encoded. The decoder

duplicates this switching between multiple models for lossless decompression. Separate models are used if the index is 1, the maximum value $(m+1)$, in the range 2 to $(m/4)-1$, in the range $m/4$ to $(m/2)-1$, and in the range $m/2$ to m. This simple state dependency improves compression rate by accounting for some spatial redundancy.

4.5.1. Experimental Results

The improvement in rate-distortion performance when entropy compression is applied to the indices is significant. The rate-distortion curve for the *lenna* image is presented in Figure 37. The figure compares the adaptive arithmetic and Lempel-Ziv coders against the LBG and basic LAVQ algorithms. The Lempel-Ziv and adaptive arithmetic coders perform comparably, though the Lempel-Ziv algorithm does slightly better than the arithmetic coder. Coding of this type is most beneficial in the high distortion (low rate) regions.

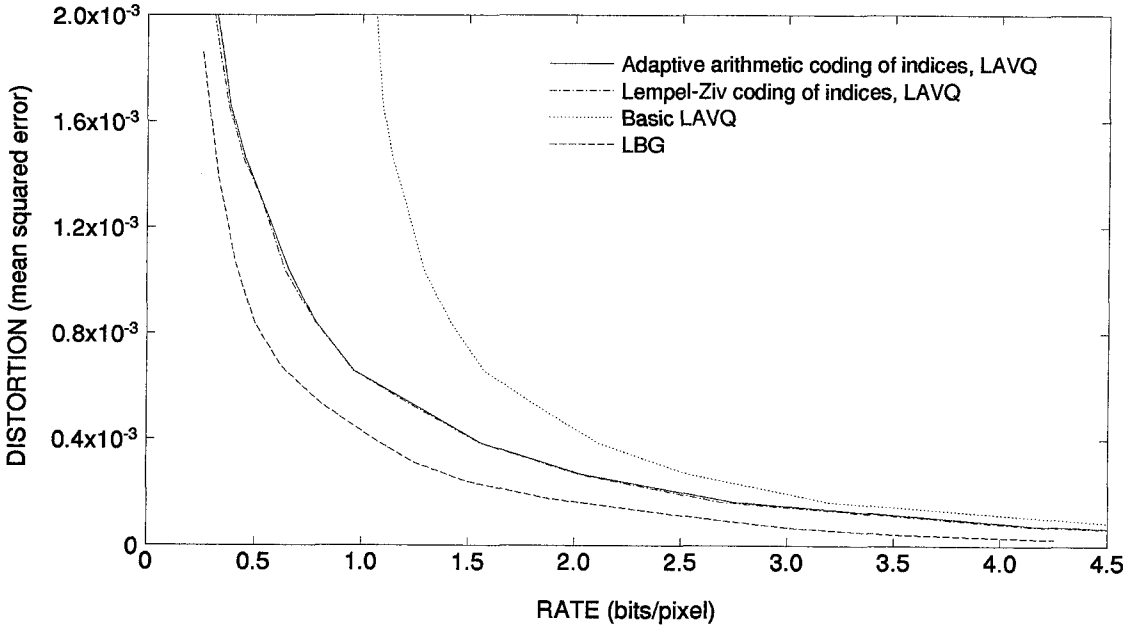

Figure 37: Entropy coding of indices, rate-distortion curve. Rate-distortion curves for *lenna* image. LAVQ indices were compressed with an adaptive arithmetic coder and a Lempel-Ziv coder. All curves were generated using parameters identical to those used in Figure 30.

This improvement in performance is observed because the effect of entropy coding on the indices is to shift the rate-distortion curves to the left. The amount of shift is comparable in both the low and high distortion regions since the number of indices is the same regardless of the distortion. This is in contrast to the case of code book entropy compression, where the number of new codewords decreases as the distortion rises. Thus, entropy compression of the indices has the most effect on

rate-distortion performance in the high distortion region, where the curves are practically vertical and where the indices are a major fraction of the compressed image size. Furthermore, since images tend to be highly correlated, the good performance obtained through Lempel-Ziv coding and the adaptive arithmetic coder used here is not surprising.

4.6. Differential Coding with Nonlinear Quantization

One significant error type or artifact which the LAVQ algorithm causes in processed images is blockiness arising from quantization of vectors, especially at higher compression rates. This occurs from using discrete vectors to represent the image; therefore, if the pixels are slowly varying across the image, as is the case with most images, the algorithm substitutes these with regions of uniform intensity and relatively large jumps in pixel values between these regions. Differential coding attempts to limit these jumps and improve compression by utilizing this assumption of slowly varying image regions.

Scalar differential encoding uses the assumption that the present element being coded has a similar value to the previously coded element or elements: Only the difference between the actual value and a prediction based on the previously coded element values is sent. If this assumption is true, the distribution of difference values will be biased toward lower values; therefore, nonlinear quantization and/or entropy coding of these difference values will compress the data. Here, a similar assumption is made, but the algorithm will be extended to vectors. Therefore, the assumption made for LAVQ coding is that the present vector being coded has the same mean value as the previous vector, and the differences or errors being sent are the difference of individual elements with respect to this mean value. This difference vector is what is stored in the code book and used for encoding. After each vector is processed, this mean value is updated with what the decoder will perceive as the next mean value. The decoder reinserts this mean value after receiving and decoding each vector. Continual updating of the mean assures that both the encoder and decoder will have the same mean value to remove and reinsert into the image for each processed vector.

Because difference values for images tend to be small, nonlinear quantization and entropy coding of new codeword values can be used to improve performance. Small difference values are most common; therefore, the quantization levels should be fine enough to record these accurately, while larger values need not be quantized with as many levels. The choice of quantization step sizes is not clearly defined, as the image statistics are unknown to the encoder and cannot be easily transmitted

8-bit Quantization

x	Q(x)
$0 \le x \le 63$	x
$64 \le x \le 127$	$2\lfloor x/2 \rfloor$
$128 \le x \le 255$	$4\lfloor x/4 \rfloor+1$

7-bit Quantization

x	Q(x)
$0 \le x \le 31$	x
$32 \le x \le 63$	$2\lfloor x/2 \rfloor$
$64 \le x \le 95$	$4\lfloor x/4 \rfloor+1$
$96 \le x \le 127$	$8\lfloor x/8 \rfloor+4$
$128 \le x \le 159$	$16\lfloor x/16 \rfloor+8$
$160 \le x \le 191$	175
$192 \le x \le 255$	222

5-bit Quantization

x	Q(x)
$0 \le x \le 5$	x
$6 \le x \le 7$	6
$8 \le x \le 10$	8
$11 \le x \le 14$	11
$15 \le x \le 20$	16
$21 \le x \le 27$	23
$28 \le x \le 39$	32
$40 \le x \le 54$	45
$55 \le x \le 77$	64
$78 \le x \le 108$	91
$109 \le x \le 255$	128

6-bit Quantization

x	Q(x)
$0 \le x \le 15$	x
$16 \le x \le 17$	16
$18 \le x \le 20$	19
$21 \le x \le 24$	23
$25 \le x \le 29$	27
$30 \le x \le 34$	32
$35 \le x \le 41$	38
$42 \le x \le 49$	45
$50 \le x \le 58$	54
$59 \le x \le 69$	64
$70 \le x \le 82$	76
$83 \le x \le 98$	91
$99 \le x \le 117$	108
$118 \le x \le 139$	128
$140 \le x \le 165$	152
$166 \le x \le 197$	181
$198 \le x \le 255$	215

4-bit Quantization

x	Q(x)
$0 \le x \le 2$	x
$3 \le x \le 5$	4
$6 \le x \le 11$	8
$12 \le x \le 22$	16
$23 \le x \le 45$	32
$46 \le x \le 255$	64

Table 3: Quantization tables for nonlinear quantization. Tables correspond to eight, seven, six, five, and four bit quantization (255, 127, 63, 31, and 15 levels used, respectively). Only positive values are listed; $Q(x)$ is symmetrical about the origin: $Q(-x) = -Q(x)$.

to the decoder when the image is to be compressed in only one pass. Therefore, the quantizer used in this work is not adaptive, and an arbitrary choice is made beforehand. Quantizer design has some criteria, however. Initial step sizes (near zero) should be small, and subsequent sizes should increase. Based on this assumption, a fixed quantizer was constructed with a quasi-logarithmic step size progression. The quantizer function values are listed in Table 3.

4.6.1. Experimental Results

The rate-distortion curve for the *lenna* image for LAVQ coding with nonlinear quantization and codeword entropy coding is presented in Figure 38. Difference coding with 8-, 7-, 6-, 5-, and 4-bit quantization is used; best results were obtained by using 5-bit quantization. Higher resolution yielded slightly worse performance due to increased rate; lower 4-bit resolution also had worse

performance due to quantization errors. In addition, entropy coding is applied to both indices and codewords to exploit the advantage which difference coding provides. Entropy coding algorithms are identical to those used for code book and index compression in the preceding sections. Indices are compressed with the Lempel-Ziv algorithm, while codewords are compressed with the adaptive arithmetic coder. The improvement in rate-distortion performance is significant; the performance of the LAVQ now approaches that of the LBG algorithm.

The computational complexity involved in recording, removing, and restoring mean values is not significant compared to the task of searching the code book for the appropriate codeword.

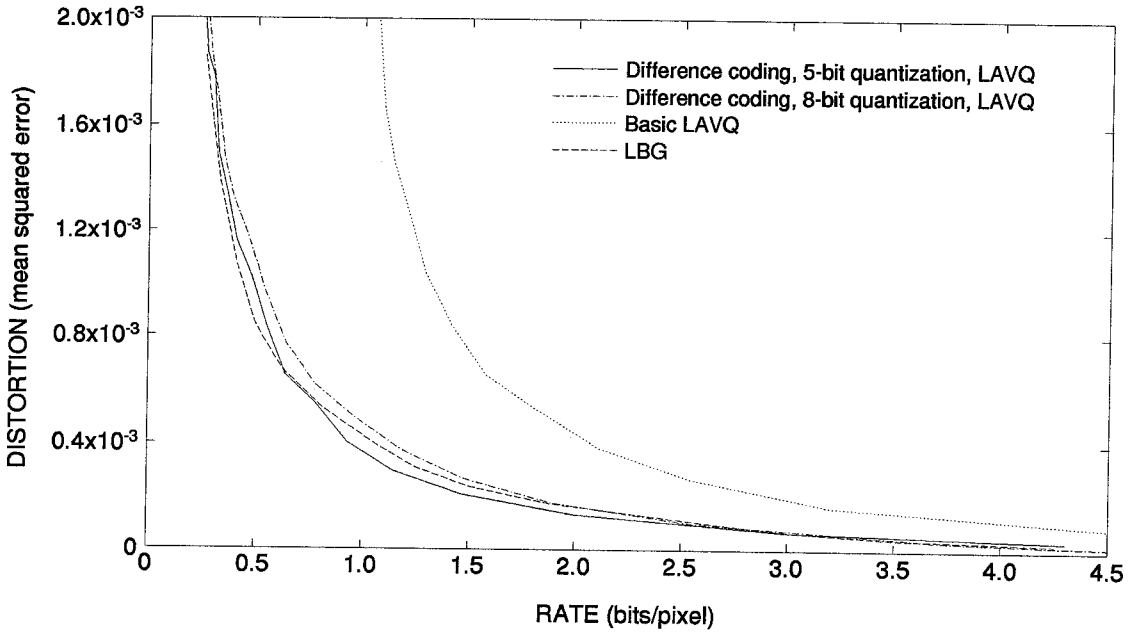

Figure 38: Difference coding, rate-distortion curve. Rate-distortion curves for *lenna* image. LAVQ coding was applied to the vectors with mean values removed. All curves were generated using parameters identical to those used in Figure 30.

4.7. Image and Block Scanning

The improvements covered so far concentrate on decreasing file size (improving compression rate) with little or no degradation in image quality (distortion). Furthermore, these techniques were applied directly to the encoding and decoding process or were applied after the basic encoding process was completed. The improvements of this and the following sections deal with methods to decrease distortion. These methods are also pre- and post-processing techniques, since they are applied prior to encoding or after decoding with the LAVQ algorithm. Therefore, these are

pipeline-able processes which do not increase complexity of the algorithm in actual implementation. Furthermore, the techniques outlined here are not computationally complex compared to the basic LAVQ algorithm.

The first approach to improving performance lies in the method by which the image is scanned to generate the vectors, or blocks, necessary for LAVQ encoding. In most standard vector quantization algorithms the order in which the blocks are scanned into memory is not significant since these algorithms are memoryless; that is, they do not take into account physical location when encoding the blocks. (One such algorithm which is state-dependent is tree search VQ.) Typically, these algorithms employ a standard raster scan of blocks; that is, they order the blocks in rows from the top, scanning left to right. LAVQ has a long string of memory; it is dependent on those blocks which immediately precede it. Therefore, a scan which only traverses left to right may not encode the image as well as a scan which traverses in a "snake scan." An example of a snake scan is shown in Figure 39.

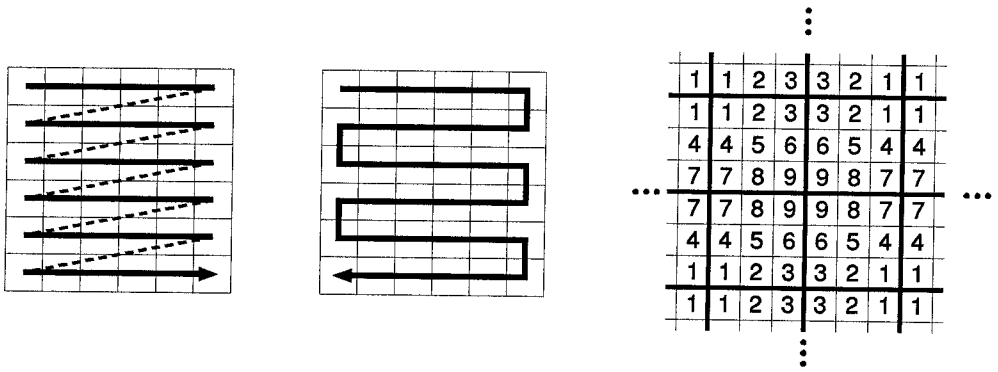

Figure 39: Scanning methods. From left to right, standard raster scan, snake scan, and block flipping methods. For block flipping, numbers denote pixel order for forming nine-element vectors from 3×3 blocks.

Block flipping can be applied to any block ordering method; it refers to the pixel ordering applied when vectors are formed from the pixels. Figure 39 shows an example of block flipping for 3×3 pixel blocks. By flipping each alternating column and each alternating row of blocks, the pixels are organized in vector format such that each pixel along the edges of adjacent blocks is stored in the same vector element. Therefore, this has potential to decrease the blockiness or staircase effect. The appproach has been explored in [99].

4.7.1. Experimental Results

Performance of snake scan and block flipping methods does not provide much improvement in rate-distortion performance. Figure 40 shows slight improvement using snake scan and block flipping methods, especially in the high distortion region. This is expected, as the slight decrease in the number of new codewords sent will not be advantageous except when the total number of new codewords is small. Furthermore, block flipping will not be of much utility where the blocks are taken 1×8 pixels. The cost of adding these algorithms, however, is inconsequential; therefore, they will be used in subsequent sections.

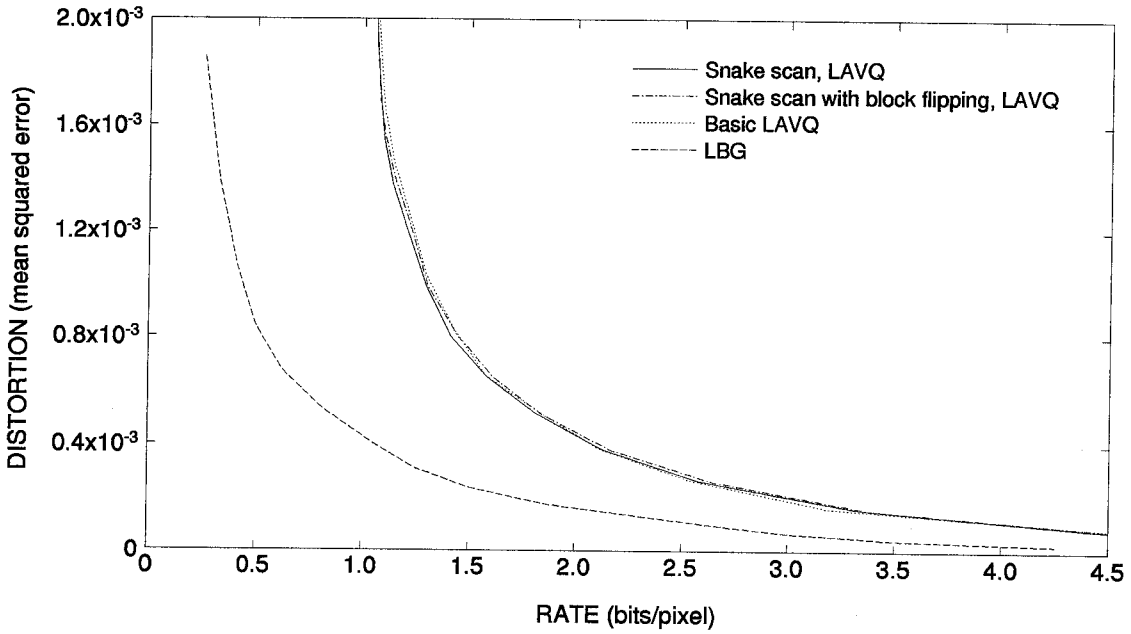

Figure 40: Scanning methods, rate-distortion curve. Rate-distortion curves for *lenna* image. LAVQ applied to blocks generated with raster scan, snake scan, and snake scan with block flipping. Parameters used for generating the curves are identical to those used in Figure 30.

4.8. Filtering with Block Replacement

The principal source of noise or distortion in the LAVQ algorithm is blockiness in the image. This occurs since there are mismatches at the edges of each block which can be repetitive and large enough to be visible. Thus, with the parameters being used in this work, the errors are in the vertical direction with a spatial frequency being 8 pixels (since 1×8 pixel blocks are being used). Since the noise will occur at regular intervals in known positions, a static filter can remove or at least reduce

this noise. This approach was used in [46].

To filter out block boundaries, a bandstop filter with stop bands corresponding to the spatial block frequency and its harmonics is used. The filter must have smooth transitions to avoid "ringing" at edges in the image, yet must have sufficient selectivity in stopband attenuation to avoid attenuating too many frequencies. Excessive filtering will also cause the image to appear fuzzy or out of focus in the high detail regions and edges. The filter used here is the raised sinusoid function,

$$f(\omega_x) = 1 - a_0 \sin^4\left[\frac{\pi\omega_x n_x}{2N_x}\right] \tag{88}$$

where

a_0 = filtering attenuation value
ω_x = spatial frequency along the x-axis
n_x = size of a block in the x-direction
N_x = size of the image in the x-direction

A similar function exists for filtering along the y-direction. The filtering process is implemented here with a fast Fourier transformation, application of the filter function, and inverse fast Fourier transformation. For the images and LAVQ parameters used in this work, since the blocks are taken 1×8, there is no need for filtering along the x-direction; $n_y = 8$, $N_y = 512$, and a_0 is variable.

Filtering with a high attenuation a_0 will cause the image to appear fuzzy in high detail regions and edges. However, insufficient attenuation will not filter the block edges sufficiently. A solution to allowing relatively high filter attenuation values while maintaining sharp features is obtained through block replacement, where blocks of the filtered image are replaced with the original blocks of the unfiltered image.

Block replacement is possible because the LAVQ algorithm differentiates between low and high detail areas. The LAVQ algorithm represents a vector in the image with one which is stored in its code book; if no sufficiently close match exists, then that vector is entered into the code book as a new codeword. New codewords tend to be entered when high detail regions or edges are recorded. Regions of low detail have the same codeword repeatedly utilized. Therefore, by replacing those blocks in the filtered image which represent these high detail regions with the original coded image's blocks, edges and details can be preserved while low detail regions have their block boundaries filtered out. A vector is considered to be in a high detail region if it is not represented by the same codeword as is the previously coded vector.

The entire operation (transform, filter, inverse transform, and block replacement) is computationally complex; however, all of this is done at the receiver or decoder. The LAVQ encoder is

more computationally complex than the decoder: The encoder must find a match between the vector and the codewords in the code book, a computationally complex task, and update the code book. The decoder, on the other hand, only needs to update the code book and reconstruct the image. The computational bottleneck, therefore, occurs at the encoder. Additional complexity placed on the decoder does not worsen this bottleneck. Indeed, this type of processing is suitable for cases where the encoder must be as simple and small as possible while the decoder has less stringent limits on size and computational power, such as for deep space probes: Power, size, weight, and reliability issues force spacecraft computers to be limited in capacity. In comparison, ground-based units have no such restrictions and can therefore add more processing to received image data.

4.8.1. Experimental Results

Filtering with block replacement applied to the *lenna* image shows improvement in image quality in high detail regions and along edges. The processed images have less blockiness in low detail regions while maintaining image quality in high detail regions and along edges. As an example consider Figures 41 and 42. Figure 41 shows the locations of the non-repeating blocks; this confirms that details and edges tend to be covered by different vectors. The benefit of filtering with block replacement is dependent on the number of nonrepeating vectors available in the coded image; therefore, at low rates there may not be sufficient numbers of nonrepeating blocks to be beneficial. Furthermore, at high rates, though there are many nonrepeating blocks for block replacement, filtering may be of limited utility since most filtered blocks will be replaced, negating the effectiveness of the filtering process.

Figure 41: Nonrepeating codeword locations. Nonrepeating codeword locations for the *lenna* image. Note that the LAVQ algorithm picks out edges and areas of high detail well. Coding parameters were identical to those used in Figure 28.

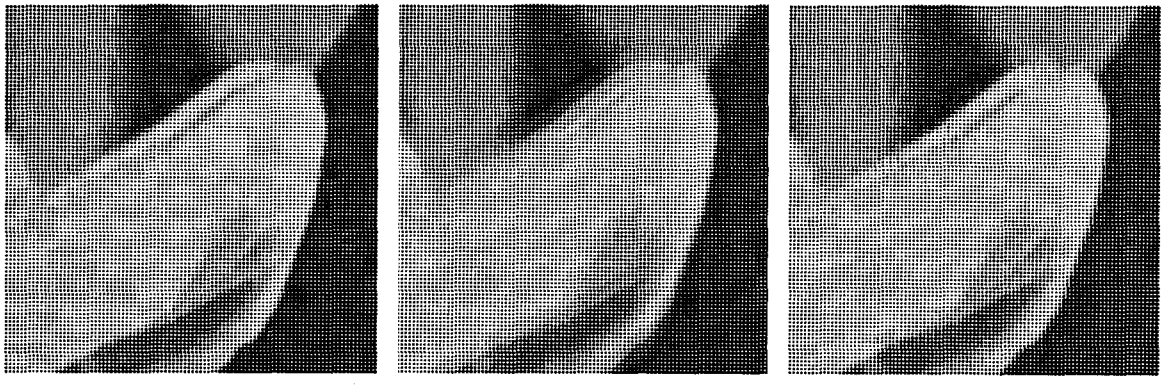

Figure 42: Filtering with block replacement detail. Detail of upper right edge of hat brim from the *lenna* image. From left to right: basic LAVQ algorithm, filtered image (with attenuation $a_0 = 1.0$), and filtered image with block replacement.

The effects of filtering and block replacement are shown in Figure 42. In both figures, the block boundaries of vectors are clearly visible for the basic LAVQ implementation. The edges also remain quite sharp in these cases. Filtering can help remove the blockiness in low detail areas, as shown in the same figures, but the cost is blurred edges and ringing effects. Ringing refers to the shadows and ghost images which appear; this occurs from the filter's removing of some frequency elements. The block boundaries, however, are attenuated significantly. Finally, through block replacement, ringing and blurred details are reduced without reintroducing visible block boundaries. Thus, the image has improved quality in both low and high detail regions.

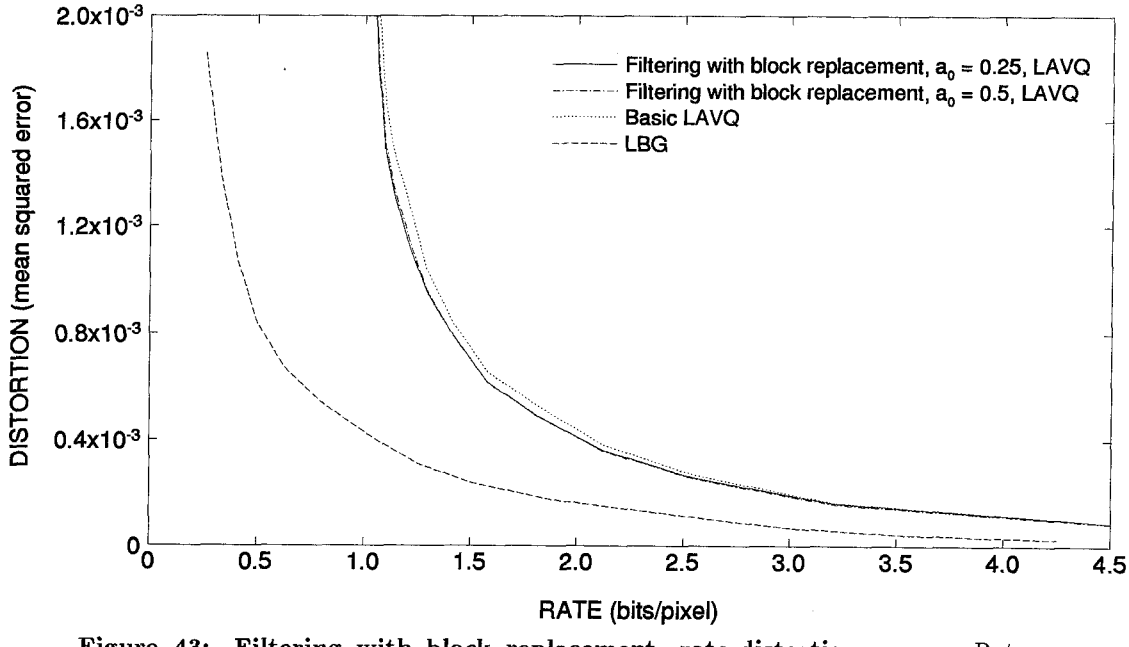

Figure 43: Filtering with block replacement, rate-distortion curve. Rate-distortion curves for *lenna* image. LAVQ image was processed using the filtering with block replacement strategy. All curves were generated using parameters identical to those used in Figure 30.

However, though the subjective image quality is considerably improved, the rate-distortion performance is only slightly improved. Figure 43 shows little improvement by using filtering with block replacement. The eye sees significant improvement in performance; however, the measure of distortion, mean squared error, cannot differentiate between accuracy of details and instead weighs each pixel and each squared pixel difference equally. In terms of rate-distortion performance, filtering with smaller values of a_0, typically $0.10 < a_0 < 0.25$, provides greater improvement in performance; in terms of subjective quality, however, larger values of a_0, in the range of $0.25 < a_0 < 0.50$, provide better subjective image quality improvement.

4.9. Interpolation

The previous section dealt with filtering along the vertical block boundaries to improve image quality. Another means by which image quality can be improved is by horizontal interpolation. As blocks are scanned into the encoder along horizontal lines, there are often areas where the same block is repeated. This region is one of low detail, and therefore of slowly varying pixel values. When the error allowance is exceeded, however, a new block is used. This process can cause low detail areas to be represented by step-like visible changes in pixel intensity. Interpolating between the first occurrences of differing vectors along a scanned image, therefore, can limit step intensity and improve subjective appearance.

Using differing vectors as delimiters for the interpolation function, however, can prevent interpolation from occurring for some pixel values, as is the case when another element in the vector changes. Also, difference coding does a simple form of horizontal interpolation: The interpolation function is similar to the linear function explored in the next subsection. This can limit the effectiveness of interpolation because the interpolation has already been applied.

A solution to these two problems is to apply the interpolation on a pixel-wise basis: The function is applied based on the pixel values of each row of the decoded image, scanned in the same direction as the decoding algorithm. Differing pixel values now form the delimiters for the interpolation function; this allows for interpolation where needed.

4.9.1. Linear Interpolation

Linear interpolation is a simple method of smoothing the "quantization noise" which occurs from the visible changes in pixel intensity occurring from slowly varying pixel values being mapped

against an error allowance and existing codewords. This interpolation, as do the others, uses the decoded image. The occurrences of differing pixel values in the horizontal direction are marked in the image in a manner similar to what is done for filtering with block replacement, a strategy explored in the previous section. The decoded image is then scanned again; those pixels between marked pixels are filled in with a pixel-wise linear interpolation along the horizontal direction of the two boundary pixels. The interpolation function between pixels is

$$x_k = x_i + \frac{x_j - x_i}{j - i}(k - i) \qquad \text{for} \quad i < k < j$$

where

i = position of first occurrence of repeating pixel value
j = position of first occurrence of differing pixel value
x_k = pixel of k-th position

Figure 44 depicts what the effect of linear interpolation would be in an image. The most serious drawback of linear interpolation is representation of edges; this makes it of limited utility. With linear interpolation, sharp edges in the image will be blurred horizontally; this will cause any image with sharply defined vertical lines to have smudges replace these features. Avoiding this by limiting the amount of interpolation allowed does not remove the difficulty arising from less sharp images, where edges may not be as sharply defined. Furthermore, as filtering with block replacement has shown, sharply defined features contribute to a subjectively more appealing image; linear interpolation destroys these features. A possible solution to this problem is addressed in the next subsection.

4.9.2. Exponential Interpolation

To avoid edge smearing caused by linear interpolation, exponential interpolation, based on an exponential decay of linear tracking, can be used. The function used is therefore

$$x_k = x_i + [x_j - x_i]\frac{e^{-\tau(j-k)} - e^{-\tau(j-i)}}{1 - e^{-\tau(j-i)}} \qquad \text{for} \quad i < k < j \tag{89}$$

where τ is an interpolation parameter. The final slope of the interpolation function is determined by τ; the slope of pixel values per pixel position at the end of the interpolation region ($k = j - 1$) will be the step size multiplied by τ. Greater values of τ correspond to sharper cutoffs; for $\tau \gg 1$ the result is similar to no interpolation at all. For $\tau \ll 1$ the result is similar to linear interpolation.

Exponential interpolation allows a smoother transition than is possible without interpolation, but without as much smearing of edges as linear interpolation. Exponential interpolation in effect fixes the slope of the interpolation near the end of the interpolation region. This allows only

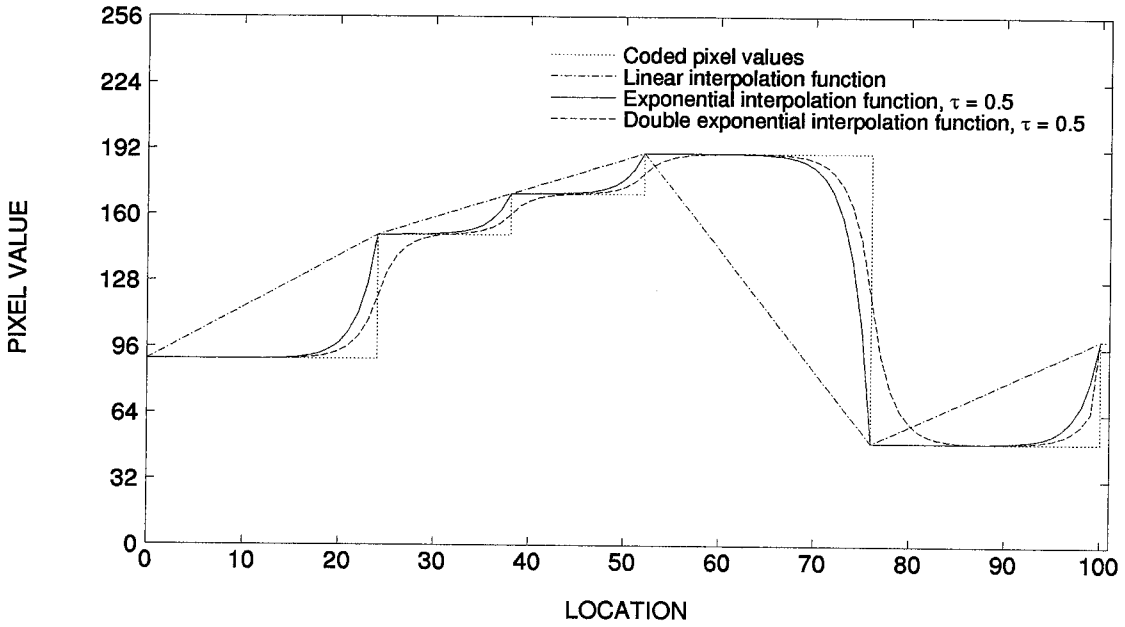

Figure 44: Interpolation functions. Sample of interpolation functions for a horizontal slice of pixel values.

a small change in pixel value near the start of the interpolation region, thus limiting the smearing of sharp edges. (See Figure 44.) If the distance between the two differing pixels is small, the interpolation function appears more like a linear interpolation; if the distance is large, then the limited smoothing properties become evident.

4.9.3. Double Exponential Interpolation

Exponential interpolation does not remove the sharp edges which can yield more visible block boundaries; another attempt to remove these is a double exponential interpolation technique. The exponential interpolation function is applied twice to smooth the step: Both the "tread-to-riser" corner (the horizontal-to-vertical corner as seen from left to right, smoothed by the exponential interpolator of the previous subsection) and the "riser-to-tread" corner (the vertical-to-horizontal corner as seen from left to right) will be interpolated with an exponential function.

$$x_k = x_i - \left[\frac{x_i - x_h}{2}\right] \frac{e^{-\tau(k-i)} - e^{-0.5\tau(j-i)}}{1 - e^{-0.5\tau(j-i)}} \qquad \text{for} \quad i < k < \tfrac{i+j}{2}$$

$$x_k = x_i + \left[\frac{x_j - x_i}{2}\right] \frac{e^{-\tau(j-k)} - e^{-0.5\tau(j-i)}}{1 - e^{-0.5\tau(j-i)}} \qquad \text{for} \quad \tfrac{i+j}{2} < k < j$$

where

i = position of first occurrence of repeating pixel value
j = position of first occurrence of differing pixel value
h = position of last previous differing pixel value

x_k = pixel of k-th position
τ = interpolation parameter

As in the exponential interpolation case, the slope near the vertical component of the step is determined by τ; the slope of pixel values per pixel position at the transition regions ($k = i$ and $k = j - 1$) will be the step size multiplied by τ. Also, greater values of τ correspond to sharper cutoffs; for $\tau \gg 1$ the result is similar to no interpolation at all. For $\tau \ll 1$ the result is similar to linear interpolation except that the slope discontinuity points are no longer at the original step discontinuity points. (See Figure 44.)

The advantage of double exponential interpolation is smoother interpolation of step discontinuities than is possible with exponential interpolation but without the edge smearing of linear interpolation. However, because the step riser-to-tread corner, which represents a more accurate representation of the image, is smoothed, the mean squared error is increased with this interpolation technique. Therefore, though the double interpolation function removes block boundaries very well without much edge smoothing, the penalty in distortion is high.

4.9.4. Experimental Results

The three above interpolation functions are applied to the *lenna* image; the resulting rate-distortion curves are shown in Figure 45. Various values of τ for the exponential and double exponential interpolation functions were applied. Small values of τ result in performance similar to the linear interpolation case; large values give performance similar to no interpolation. Good results are obtained by setting $\tau = 1.0$ for both exponential and double exponential interpolation.

Linear interpolation yields rate-distortion performance worse than what is obtained without interpolation. As Figure 44 shows, linear interpolation smears edges excessively, resulting in images with subjectively worse quality as well as worse rate-distortion performance. Double exponential interpolation does only marginally better than no interpolation because this function interpolates the riser-to-tread corner, which is usually more accurately represented by the LAVQ algorithm. The exponential interpolation function does best in improving rate-distortion performance, though this gain is small: Interpolation serves to decrease distortion, especially in the high distortion region; however, here the curve is near vertical, so downward shifts are less observable.

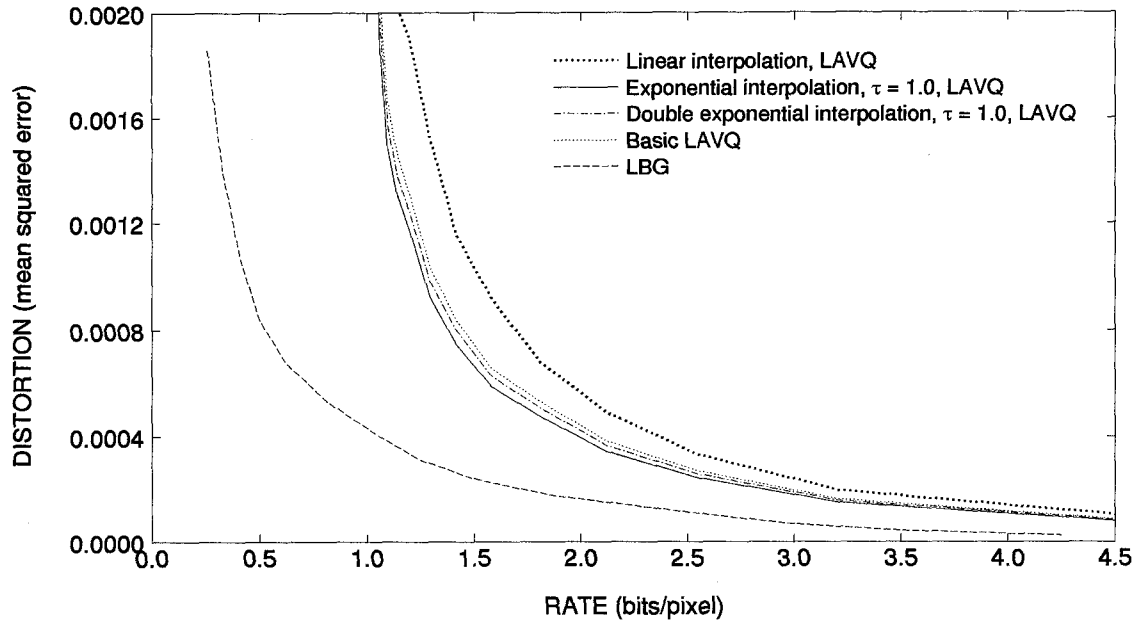

Figure 45: Interpolation, rate-distortion curve. Rate-distortion curves for *lenna* image. LAVQ image was processed using three interpolation functions. All curves were generated using parameters identical to those used in Figure 30.

4.10. Best Case Improvements

All of the improvements of this chapter are incorporated here to provide the best case performance of the LAVQ algorithm. The algorithm was invoked upon the four images *lax*, *lenna*, *med01*, and *saturn1*; for comparison purposes, the basic LAVQ algorithm and the LBG algorithm were also applied. Various parameters were attempted to obtain a good solution; however, best results were obtained for all four images with similar parameter values. This is encouraging in that if there exists a set of parameters which yields good results for most images, then there is little need to optimize for each given situation.

As expected from the previous sections, difference coding and entropy compression yielded the greatest improvement in rate; other methods added incrementally to performance. The parameters used for all images are as follows:

- 1×8 pixel codewords
- 255-entry code book
- Snake scanning of image
- First match code book search algorithm
- Difference coding with 6-bit nonlinear quantization
- Adaptive arithmetic code book compression
- Lempel-Ziv index compression
- Exponential horizontal interpolation as given by equation (89) with $\tau = 2.0$
- Filtering with block replacement, with filter function given by equation (88)
  attenuation factor: *lax*: $a_0 = 0.3$ *med01*: $a_0 = 0.1$
  lenna: $a_0 = 0.2$ *saturn1*: $a_0 = 0.3$

With the exception of the filter attenuation parameter a_0, the parameters for the best case results are the same for all four images. Setting a_0 the same at approximately 0.25 for all images yielded results which were only slightly worse than the best case.

4.10.1. Experimental Results

The rate-distortion characteristics of the best case LAVQ algorithm on the four images are compared with the performance of the best case (best block and code book size) LBG algorithm in Figures 46-49. In these figures, the basic LAVQ algorithm's performance is also plotted. With the exception of the *lax* image, the improved LAVQ algorithm performs as well or better than the LBG algorithm through most of the rate-distortion range plotted. In the *lax* image, the improved LAVQ algorithm has performance which approaches that of the LBG algorithm.

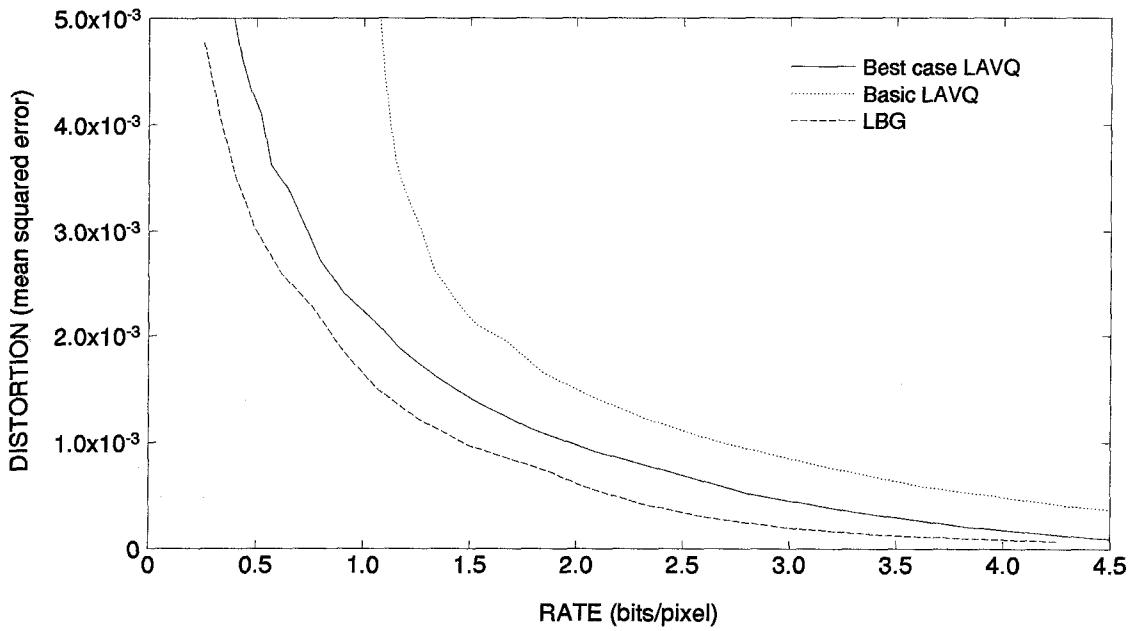

Figure 46: Best case LAVQ, rate-distortion curve, *lax*. Rate-distortion curves for *lax* image coded with parameters for LAVQ algorithm giving best results.

The improved LAVQ algorithm still maintains its detail preserving features. Figures 50 and 51 show detailed portions of the *lax* and *lenna* images, respectively; with similar distortion values, the resulting images show more detail after processing with the improved LAVQ algorithm than with LBG. The *lax* image, though, is not as well compressed with the improved LAVQ algorithm as it is with the LBG: LBG obtains a mean squared error of 2.572×10^{-3} at 0.625 bit/pixel; improved LAVQ requires 0.849 bit/pixel to obtain a comparable distortion (2.565×10^{-3}). However, unlike

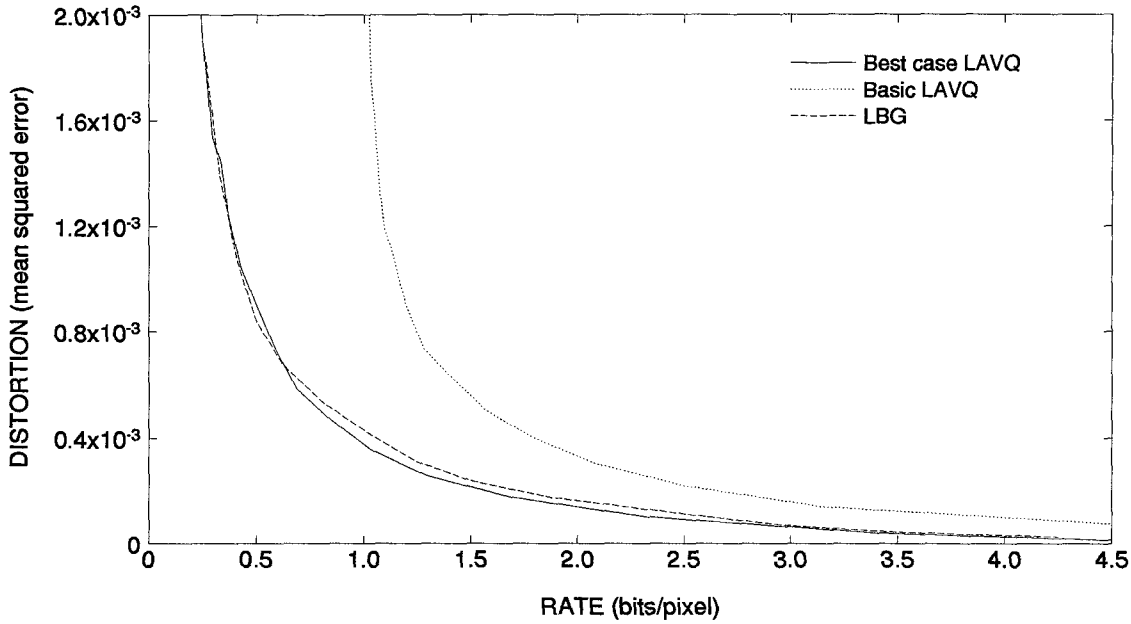

Figure 47: Best case LAVQ, rate-distortion curve, *lenna*. Rate-distortion curves for *lenna* image coded with parameters for LAVQ algorithm giving best results.

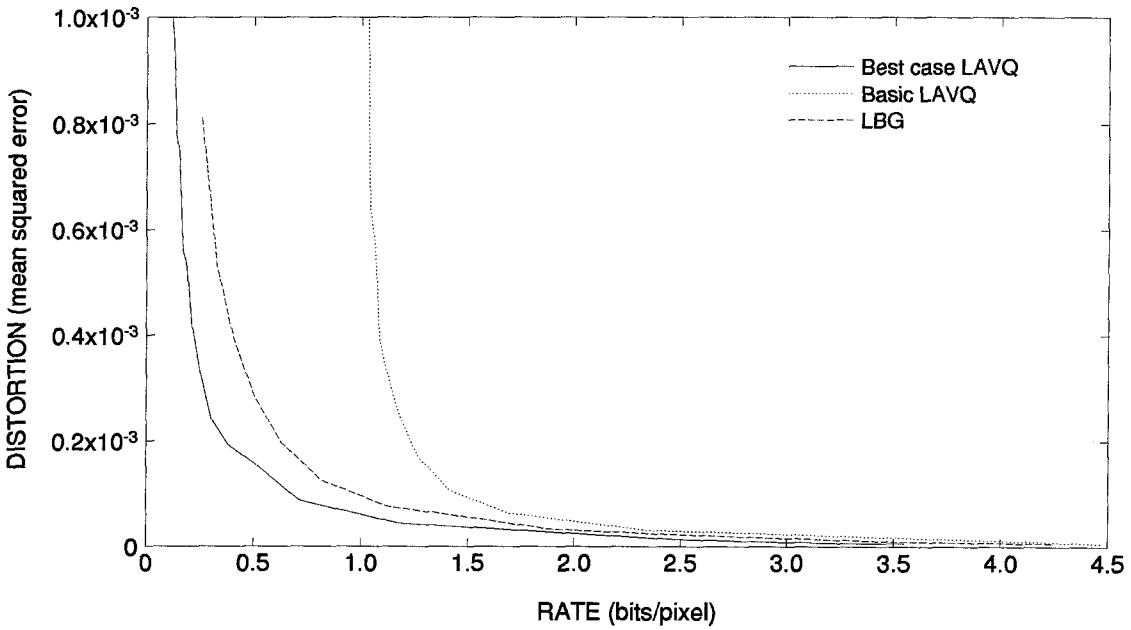

Figure 48: Best case LAVQ, rate-distortion curve, *med01*. Rate-distortion curves for *med01* image coded with parameters for LAVQ algorithm giving best results.

the LBG algorithm, the improved LAVQ algorithm renders the aircraft, vehicles, and terminal with far greater detail than the LBG algorithm. The *lenna* image is compressed to about the same rate with both algorithms: At 0.625 bit/pixel, LBG has a mean squared distortion of 6.744×10^{-4}, while

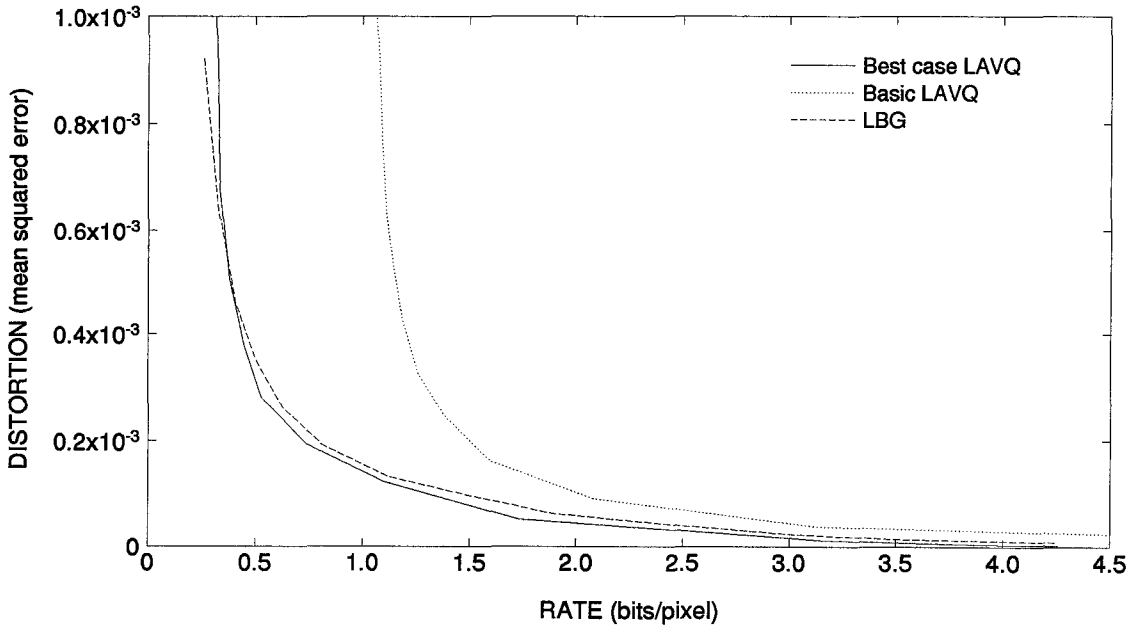

Figure 49: Best case LAVQ, rate-distortion curve, *saturn1*. Rate-distortion curves for *saturn1* image coded with parameters for LAVQ algorithm giving best results.

for a comparable distortion of 6.701×10^{-4}, the improved LAVQ does slightly better in terms of rate at 0.610 bit/pixel. The hat edges are clearly defined.

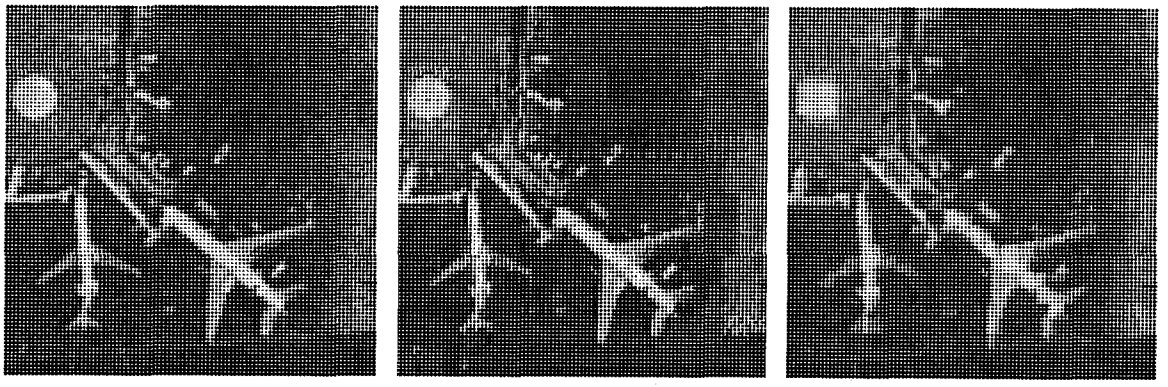

Figure 50: Best case LAVQ detail, *lax*. Detail of lower right of upper terminal from the *lax* image. From left to right: original, best case LAVQ compression, and LBG compression. Rate-distortion values are: LBG, 0.625 bit/pixel at 2.572×10^{-3} mean squared error; LAVQ, 0.849 bit/pixel at 2.565×10^{-3} mean squared error.

In general, the improved LAVQ algorithm has the most difficulty matching the performance of the LBG algorithm at high distortion (low rate) levels. Here, since the image is being compressed drastically, the representation of a large number of pixels is assigned to relatively fewer bits. The extra computation time spent by the LBG algorithm to develop a good representation (a good code book) pays off in good performance. The exception to this is the *med01* image, which, with its

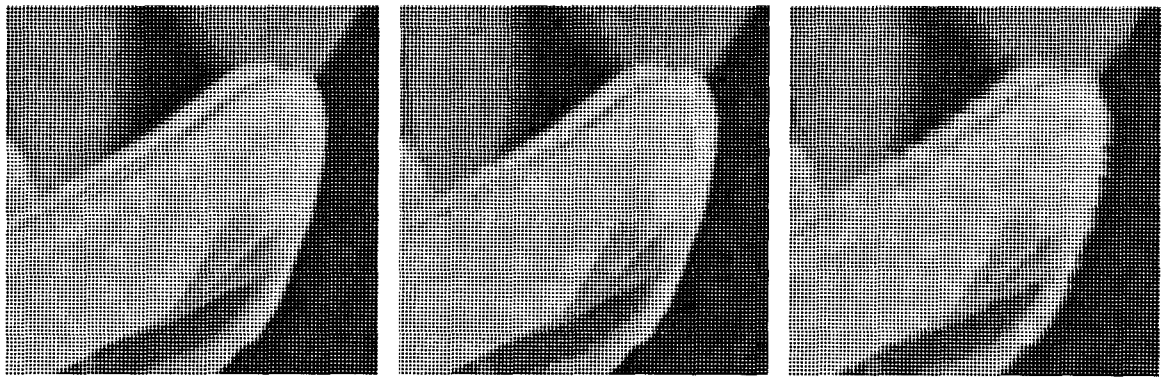

Figure 51: Best case LAVQ detail, *lenna*. Detail of upper right edge of hat brim from the *lenna* image. From left to right: original, best case LAVQ compression, and LBG compression. Rate-distortion values are: LBG, 0.625 bit/pixel at 6.744×10^{-4} mean squared error; LAVQ, 0.610 bit/pixel at 6.701×10^{-4} mean squared error.

large low detail region, is easily compressed by the LAVQ algorithm, which easily allocates very few bits to this area while maintaining a higher number of bits for the high detail areas for an accurate rendition of the image.

Despite the various post-processing functions applied to the LAVQ algorithm, this algorithm is still much faster than the LBG algorithm. With the faster search algorithms that the improved LAVQ algorithm uses, the data for the rate-distortion curves for the images were generated in a little under 20 minutes for each curve. Each curve contains approximately 25 data points. The LBG algorithm required over 200 hours for each curve, as noted earlier. Thus, the improved LAVQ algorithm obtains results comparable to the LBG algorithm at over 600 times the speed while better preserving detailed regions.

4.11. Conclusion

Various improvements to the basic LAVQ image compression algorithm were explored. These improvements are fast searching strategies, image scanning techniques, entropy coding of both the code book and the indices, differential encoding with nonlinear quantization, filtering with block replacement, and horizontal interpolation. The improved LAVQ algorithm maintains its ability of preserving fine details within the compressed image. Even with these improvements, the algorithm has a computationally simple encoder and is a fast, one-pass image compression algorithm. Its rate-distortion performance is comparable to standard vector quantization algorithms such as the LBG algorithm, but at a fraction of the computational complexity. In serial software implementation, this algorithm is faster than the LBG algorithm by a factor of over 600.

REFERENCES

[1] D. Y. Koo and H. B. Chenoweth, "Choosing a Practical Model for ECC Memory Chip," *Proc. 1984 IEEE Reliability and Maint. Symp.*, pp. 255-261, 1984.

[2] C. L. Chen and M. Y. Hsiao, "Error Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review," *IBM Journal of Research and Development*, vol. 28, pp. 124-134, 1984.

[3] T. Fuja and C. Heegard, "Focused Codes for Channels with Skewed Errors," *IEEE Transactions on Information Theory*, vol. IT-36, pp. 773-783, 1990.

[4] T. C. May and M. H. Woods, "Alpha-Particle-Induced Soft Errors in Dynamic Memories," *IEEE Transactions on Electron Devices*, vol. ED-26, pp. 2-9, 1979.

[5] M. Horiguchi, *et al.*, "An Experimental Large-Capacity Semiconductor File Memory Using 16-Levels/Cell Storage," *IEEE Journal of Solid-State Circuits*, vol. SC-23, pp. 27-33, 1988.

[6] D. Marston, "Memory System Reliability with ECC," Intel Application Note AP-73, Intel Corp., 1980.

[7] T. Fuja and C. Heegard, "Row/Column Replacement for the Control of Hard Defects in Semiconductor RAM's," *IEEE Transactions on Computers*, vol. C-35, pp. 996-1000, 1986.

[8] T. Fuja, "Coding for the Address-Defect Channel," *The 24th Annual Conference on Information Sciences and Systems*, Princeton, NJ, 21-23 March 1990.

[9] T. Fuja, "The Performance of Random Access Memory Systems Employing On-Chip and Board-Level Error Control," *IEEE International Symposium on Information Theory*, Kobe, Japan, 1988.

[10] Micron Technology Inc., "Effect of On-chip ECC on System Soft Errors," Micron Technology Inc. MT1256 Data Sheet.

[11] M. Asakura, *et al.*, "An Experimental 1-Mbit Cache DRAM with ECC," *IEEE Journal of Solid State Circuits*, vol. SC-25, pp. 5-10, 1990.

[12] T. Chiueh, R. M. Goodman, and M. Sayano, "A 2K x 1 Static RAM Chip with On-Chip Error Correction," *IEEE Journal of Solid-State Circuits*, vol. 25, pp. 1290-1294, 1990.

[13] T. Fuja, C. Heegard, and R. M. Goodman, "Linear Sum Codes for Random Access Memo-

ries," *IEEE Transactions on Computers*, vol. C-37, pp. 1030-1042, 1988.

[14] L. Levine and W. Meyers, "Semiconductor Memory Reliability with Error Detecting and Correcting Code," *Computer*, vol. 9, pp. 43-50, 1976.

[15] W. F. Mikhail, R. W. Bartoldus, and R. A. Rutledge, "The Reliability of Memory with Single-Error Correction," *IEEE Transactions on Computers*, vol. C-31, pp. 560-564, 1982.

[16] R. A. Rutledge, "Models for the Reliability of Memory with ECC," *Proc. 1985 IEEE Reliability and Maint. Symp.*, pp. 57-62, 1985.

[17] H. Vinck and K. Post, "On the Influence of Coding on the Mean Time to Failure for Degrading Memories with Defects," *IEEE Transactions on Information Theory*, vol. IT-35, pp. 902-906, 1989.

[18] R. M. Goodman and R. J. McEliece, "Lifetime Analyses of Error-Control Coded Semiconductor RAM Systems," *IEE Proceedings*, vol. 129E, pp. 81-85, 1982.

[19] R. M. Goodman and R. J. McEliece, "Hamming Codes, Computer Memories, and the Birthday Surprise," *Proc. 20th Allerton Conference on Communication, Control, and Computing*, 1982.

[20] M. Blaum, "Error-Correcting Codes for Computer Memories," Ph.D. Thesis, California Institute of Technology, 1985.

[21] M. Blaum, R. M. Goodman, and R. J. McEliece, "The Reliability of Single-Error Protected Computer Memories," *IEEE Transactions on Computers*, vol. C-37, pp. 114-119, 1988.

[22] A. M. Saleh, J. J. Serrano, and J. H. Patel, "Reliability of Scrubbing Recovery-Techniques for Memory Systems," *IEEE Transactions on Reliability*, vol. 39, pp. 114-122, 1990.

[23] R. Krishnamoorthy and C. Heegard, "Reliability and Yield: Error Control in Semiconductor RAMs," *IEEE Transactions on Information Theory*, in press.

[24] S. Lin and D. J. Costello, *Error Control Coding: Fundamentals and Applications*, Prentice Hall, 1983.

[25] P. Elias, "Error-Free Coding," *Transactions of the IRE Professional Group on Information Theory*, vol. IT-4, pp. 29-37, 1954.

[26] E. N. Gilbert, "A Problem in Binary Encoding," *Proceedings of the Symposium of Applied Mathematics*, vol. 10, pp. 291-297, 1960.

[27] P. G. Neumann, "A Note on Gilbert Burst-Correcting Codes," *IEEE Transactions on Information Theory*, vol. IT-11, pp. 377-384, 1965.

[28] L. R. Bahl and R. T. Chien, "On Gilbert Burst-Error-Correcting Codes," *IEEE Transactions*

on Information Theory, vol. IT-15, pp. 431-433, 1969.

[29] L. R. Bahl and R. T. Chien, "Single- and Multiple-Burst-Correcting Properties of a Class of Cyclic Product Codes," *IEEE Transactions on Information Theory*, vol. IT-17, pp. 594-600, 1971.

[30] H. O. Burton and E. J. Weldon, "Cyclic Product Codes," *IEEE Transactions on Information Theory*, vol. IT-11, pp. 433-439, 1965.

[31] P. G. Farrell and S. J. Hopkins, "Burst-Error-Correcting Array Codes," *The Radio and Electronic Engineer*, vol. 52, pp. 188-192, 1982.

[32] M. Blaum, P. G. Farrell, and H. C. A. van Tilborg, "A Class of Burst Error-Correcting Array Codes," *IEEE Transactions on Information Theory*, vol. IT-32, pp. 836-839, 1986.

[33] W. Zhang and J. K. Wolf, "A Class of Binary Burst Error-Correcting Quasi-Cyclic Product Codes," *IEEE Transactions on Information Theory*, vol. IT-34, pp. 463-479, 1988.

[34] M. Blaum, P. G. Farrell, and H. C. A. van Tilborg, "Multiple Burst Error-Correcting Array Codes," *IEEE Transactions on Information Theory*, vol. IT-34, pp. 1061-1066, 1988.

[35] M. Blaum, "A Family of Efficient Burst Error-Correcting Array Codes," IBM Internal Research Report RJ6732, 1989.

[36] Z. Zhang, "Limiting Efficiencies of Burst-Correcting Array Codes," *IEEE Transactions on Information Theory*, vol. IT-37, pp. 976-981, 1991.

[37] K. N. Sivarajan, R. J. McEliece, and H. C. A. van Tilborg, "Burst-Error-Correcting and Detecting Codes," *IEEE International Symposium on Information Theory*, San Diego, California, 14-19 January 1990.

[38] R. M. Goodman and M. Sayano, "Size Limits on Phased Burst Error Correcting Array Codes," *Electronics Letters*, vol. 26, pp. 55-56, 1990.

[39] P. Calingaert, "Two-Dimensional Parity Checking," *Journal of the ACM*, vol. 8, pp. 186-200, 1961.

[40] M. Blaum, "A Class of Byte-Correcting Array Codes," IBM Internal Research Report RJ5652, 1987.

[41] P. Ribenboim, *The Book of Prime Number Records*, 2nd ed., Springer-Verlag, 1989.

[42] Y. Shiloach, "A Fast Equivalence-Checking Algorithm for Circular Lists," *Information Processing Letters*, vol. 8, pp. 236-238, 1979.

[43] M. Blaum and R. M. Roth, "New Array Codes for Multiple Phased Burst Correction," IBM Internal Research Report RJ8303, 1991.

[44] Y. Linde, A. Buzo, and R. M. Gray, "An Algorithm for Vector Quantizer Design," *IEEE*

Transactions on Communications, vol. COM-28, pp. 84-95, 1980.

[45] N. M. Nasrabadi and R. A. King, "Image Coding Using Vector Quantization: A Review," *IEEE Transactions on Communications*, vol. COM-36, pp. 957-971, 1988.

[46] A. Makur, *Low Rate Image Coding Using Vector Quantization*, Ph.D. thesis, California Institute of Technology, 1990.

[47] R. M. Gray, "Vector Quantization," *IEEE ASSP Magazine*, vol. 1, no. 2, pp. 4-29, April 1984.

[48] J. Foster, R. M. Gray, and M. O. Dunham, "Finite-State Vector Quantizers for Waveform Coding," *IEEE Transactions on Information Theory*, vol. IT-31, pp. 348-359, 1985.

[49] R. Aravind and A. Gersho, "Image Compression Based on Vector Quantization with Finite Memory," *Optical Engineering*, vol. 26, pp. 570-580, 1987.

[50] R. F. Chang, W. T. Chen, and J. S. Wang, "A Fast Finite-State Algorithm for Vector Quantizer Design," *European Transactions on Telecommunications and Related Technologies*, vol. 2, pp. 21-44, 1991.

[51] T. R. Fischer, M. W. Marcellin, and M. Wang, "Trellis-Coded Vector Quantization," *IEEE Transactions on Information Theory*, vol. IT-37, pp. 1551-1566, 1991.

[52] N. M. Nasrabadi and Y. Feng, "A Multilayer Address Vector Quantization Technique," *IEEE Transactions on Circuits and Systems*, vol. CS-37, pp. 912-921, 1990.

[53] N. M. Nasrabadi and Y. Feng, "Image Compression Using Address-Vector Quantization," *IEEE Transactions on Communications*, vol. COM-38, pp. 2166-2173, 1990.

[54] N. M. Nasrabadi and Y. Feng, "Dynamic Address-Vector Quantization of RGB Colour Images," *IEE Proceedings-I*, vol. 138, pp. 225-231, 1991.

[55] B. Ramamurthi and A. Gersho, "Classified Vector Quantization of Images," *IEEE Transactions on Communications*, vol. COM-34, pp. 1105-1115, 1986.

[56] C. W. Liao and J. S. Huang, "An Edge Preserved Image Compression Technique," *Image and Vision Computing*, vol. 7, pp. 246-252, 1989.

[57] A. Kubrick and T. Ellis, "Classified Vector Quantisation of Images: Codebook Design Algorithm," *IEE Proceedings-I*, vol. 137, pp. 379-386, 1990.

[58] C. H. Hsieh, P. C. Lu, and W. G. Liou, "Adaptive Predictive Image Coding Using Local Characteristics," *IEE Proceedings-I*, vol. 136, pp. 385-390, 1989.

[59] W. H. Equitz, "A New Vector Quantization Clustering Algorithm," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-37, pp. 1568-1575, 1989.

[60] C. K. Ma and C. K. Chan, "Maximum Descent Method for Image Vector Quantisation," *Electronics Letters*, vol. 27, pp. 1772-1773, 1991.

[61] J. S. Koh and J. K. Kim, "Fast Sliding Search Algorithm for Vector Quantisation in Image Coding," *Electronics Letters*, vol. 24, pp. 1082-1083, 1988.

[62] K. K. Paliwal and V. Ramasubramanian, "Effect of Ordering the Codebook on the Efficiency of the Partial Distance Search Algorithm for Vector Quantization," *IEEE Transactions on Communications*, vol. COM-37, pp. 538-541, 1989.

[63] M. R. Soleymani and S. D. Morgera, "A Fast MMSE Encoding Technique for Vector Quantization," *IEEE Transactions on Communications*, vol. COM-37, pp. 656-659, 1989.

[64] S. H. Huang and S. H. Chen, "Fast Encoding Algorithm for VQ-based Image Coding," *Electronics Letters*, vol. 26, pp. 1618-1619, 1990.

[65] C. H. Hsieh, P. C. Lu, and J. C. Chang, "Fast Codebook Generation Algorithm for Vector Quantization of Images," *Pattern Recognition Letters*, vol. 12, pp. 605-609, 1991.

[66] J. Ngwa-Ndifor and T. Ellis, "Predictive Partial Search Algorithm for Vector Quantization," *Electronics Letters*, vol. 27, pp. 1722-1723, 1991.

[67] S. W. Ra and J. K. Kim, "Fast Weight-Ordered Search Algorithm for Image Vector Quantisation," *Electronics Letters*, vol. 27, pp. 2081-2083, 1991.

[68] N. Moayeri, D. L. Neuhoff, and W. E. Stark, "Fine-Course Vector Quantization," *IEEE Transactions on Signal Processing*, vol. SP-39, pp. 1503-1515, 1991.

[69] E. A. Riskin, T. Lookabaugh, P. A. Chou, and R. M. Gray, "Variable Rate Vector Quantization for Medical Image Compression," *IEEE Transactions on Medical Imaging*, vol. MI-9, pp. 290-298, 1990.

[70] E. A. Riskin and R. M. Gray, "A Greedy Tree Growing Algorithm for the Design of Variable Rate Vector Quantizers," *IEEE Transactions on Signal Processing*, vol. SP-39, pp. 2500-2507, 1991.

[71] Y. Yamada and S. Tazaki, "Recursive Vector Quantization for Monochrome Video Signals," *IEICE Transactions on Communications, Electronics, Information, and Systems*, vol. E74, pp. 399-405, 1991.

[72] D. J. Vaisey and A. Gersho, "Variable Block-Size Image Coding," *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 1051-1055, 1987.

[73] L. Corte-Real and A. P. Alves, "Vector Quantization of Image Sequences Using Variable Size and Variable Shape Blocks," *Electronics Letters*, vol. 26, pp. 1483-1484, 1990.

[74] I. Dinstein, K. Rose, and A. Heiman, "Variable Block-Size Transform Image Coder," *IEEE*

Transactions on Communications, vol. COM-38, pp. 2073-2078, 1990.

[75] L. M. Po and C. K. Chan, "Hierarchical Mean-Residual Image Vector Quantiser Using Gradient-Laplacian Subspace Distortion," *Electronics Letters*, vol. 26, pp. 1362-1364, 1990.

[76] P. Seitz and G. K. Lang, "A Practical Adaptive Image Compression Technique Using Visual Criteria for Still-Picture Transmission with Electronic Mail," *IEEE Transactions on Communications*, vol. COM-38, pp. 947-949, 1990.

[77] B. Marangelli, "A Vector Quantizer with Minimum Visible Distortion," *IEEE Transactions on Signal Processing*, vol. SP-39, pp. 2718-2721, 1991.

[78] Z. Xie and T. G. Stockham, "Previsualized Image Vector Quantization with Optimized Pre- and Postprocessors," *IEEE Transactions on Communications*, vol. COM-39, pp. 1662-1671, 1991.

[79] A. Gersho, "Optimal Nonlinear Interpolative Vector Quantization," *IEEE Transactions on Communications*, vol. COM-38, pp. 1285-1287, 1990.

[80] V. R. Udpikar and J. P. Raina, "BTC Image Coding Using Vector Quantization," *IEEE Transactions on Communications*, vol. COM-35, pp. 352-356, 1987.

[81] M. Kamel, C. T. Sun, and L. Guan, "Image Compression by Variable Block Truncation Coding with Optimal Threshold," *IEEE Transactions on Signal Processing*, vol. SP-39, pp. 208-212, 1991.

[82] P. Nasiopoulos, R. K. Ward, and D. J. Morse, "Adaptive Compression Coding," *IEEE Transactions on Communications*, vol. COM-39, pp. 1245-1254, 1991.

[83] B. Zeng, "Two Interpolative BTC Image Coding Schemes," *Electronics Letters*, vol. 27, pp. 1126-1128, 1991.

[84] W. Y. Chan and A. Gersho, "Constrained-Storage Quantization of Multiple Vector Sources by Codebook Sharing," *IEEE Transactions on Communications*, vol. COM-39, pp. 11-13, 1991.

[85] H. Sun and M. Goldberg, "Adaptive Vector Quantization for Image Sequence Coding," *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 339-342, 1985.

[86] M. Goldberg and H. Sun, "Image Sequence Coding Using Vector Quantization," *IEEE Transactions on Communications*, vol. COM-34, pp. 703-710, 1986.

[87] M. Goldberg and H. Sun, "Frame Adaptive Vector Quantization for Image Sequence Coding," *IEEE Transactions on Communications*, vol. COM-36, pp. 629-635, 1988.

[88] H. M. Hang and J. W. Woods, "Predictive Vector Quantization of Images," *IEEE Transactions on Communications*, vol. COM-33, pp. 1208-1219, 1985.

[89] M. Goldberg, P. R. Boucher, and S. Shlien, "Image Compression Using Adaptive Vector

Quantization," *IEEE Transactions on Communications*, vol. COM-34, pp. 180-187, 1986.

[90] A. Gersho and M. Yano, "Adaptive Vector Quantization by Progressive Codevector Replacement," *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 133-136, 1985.

[91] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A Locally Adaptive Data Compression Scheme," *Communications of the ACM*, vol. 29, pp. 320-330, 1986.

[92] K. M. Cheung and V. K. Wei, "A Locally Adaptive Source Coding Scheme," *Proceedings of the Bilkent Conference on New Trends in Communications, Control, and Signal Processing*, Ankara, Turkey, 2-5 July 1990.

[93] K. M. Cheung and V. K. Wei, "A Locally Adaptive Source Coding Scheme," submitted *IEEE Transactions on Communications*, 1990.

[94] B. Y. Ryabko, "A Locally Adaptive Data Compression Scheme," *Communications of the ACM*, vol. 30, p. 792, 1987.

[95] R. N. Horspool and G. V. Cormack, "A Locally Adaptive Data Compression Scheme," *Communications of the ACM*, vol. 30, pp. 792-794, 1987.

[96] E. Makinen, "On Implementing Two Adaptive Data-Compression Schemes," *The Computer Journal*, vol. 32, pp. 238-240, 1989.

[97] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic Coding for Data Compression," *Communications of the ACM*, vol. 30, pp. 520-540, 1987.

[98] R. B. Arps, T. K. Truong, D. J. Lu, R. C. Pasco, and T. D. Friedman, "A Multi-Purpose VLSI Chip for Adaptive Data Compression of Bilevel Images," *IBM Journal of Research and Development*, vol. 32, pp. 775-794, 1988.

[99] J. S. Koh and J. K. Kim, "Simple Block-Effect Reduction Method for Image Coding with Vector Quantisation," *Electronics Letters*, vol. 23, pp. 713-714, 1987.