Fault Tolerance Technique in Huffman Coding applies to Baseline JPEG

Cung Nguyen and Robert G. Redinbo
Department of Electrical and Computer Engineering
University of California, Davis, CA
email: cunguyen, redinbo@ece.ucdavis.edu

Abstract

Faults due to the incorrect functioning of the computation system, or the transmission errors of the internal data, could corrupt the output code stream of the Huffman encoder. In this paper, a fault detection method is proposed for the Huffman encoding system, which is implemented in the JPEG image compression standard [1]. The detection method based on the information input and the code stream output is described. The practical test results show the improvement of the reconstructed image quality.

Index Terms: Huffman coding, zigzag sequence, runlength coding.

I. INTRODUCTION

Huffman coding [2] is a very effective technique for compression data; saving of 20% to 90% are typical, depending on the characteristics of the data being compressed. Huffman coding starts by assign the shorter code words for more probable symbols and longer codewords for the less probable symbols. This coding technique is adopted by JPEG image compression standard. In JPEG image compression standard, two statistical models are used in the Huffman DCT-based sequential mode. First, the DCT basis functions are almost completely decorrelated [3] (pp. 150-57), such that they can be compressed independently without concern about correlation between coefficients. Second, the 63 AC coefficients arranged in zigzag order of decreasing probability of occurrence. Based on the statistical models, code tables for DC and AC coefficients are generalized and stored in memory storages before the bit stream is produced. Those codes will be selected to retrieve when the compression is in progressing.

It has long been recognized that error codes can seriously degrade any data have been compressed whether using lossless or lossy methods [4], [5]. Combined source-channel coding techniques have been developed integrating the compression steps with transmission error-correcting coding [6]. Such techniques are useful for modern video compression methods [7]. However, there are other sources of errors resulting from failures in the computing resources that compress and transmit the data.

Huffman coder is performed using computing resources that potentially affected by failures: the unexpected change of data bits due to physical faults in the system; the transient faults due to the noise, such as absorption of alpha particles in space [8] and electromagnetic interference, X-ray from the critical equipments, overheating. These source of errors could corrupt the output code stream. For these reasons,
fault tolerance becomes the main issue to be solved. Fault tolerance technique for the Huffman coder is based on the extra hardware components and extra time to combine with the original tasks. In this work, redundancy check bits are generated and used to compare with parity bits that are generated at the output code stream. When an error is detected, the system will request for a repeat encoding the defected data block.

The technique discussed in this paper is motivated The rest of this paper is divided into three sections. The first section overviews the Huffman encoding algorithm and data structures support the coding processes. In second section, a redundancy subsystem is proposed to detect errors occur in the Huffman code stream output. The third section provides the experiment results and the improvements in terms of of the image quality and image reconstruction error.

II. Overview the JPEG Huffman Coding

To simplify the problem, only the fault tolerance JPEG Huffman entropy coder implementations for the baseline mode operation is discussed in this paper. However the design principle can be applied to the extended systems [1]. Baseline sequential coding is for images with 8-bit samples and uses Huffman coding only, and its decoder can store only one AC and DC Huffman table. Prior to entropy coding, there usually are few nonzero and many zeros-valued coefficients. The task of entropy coding is to encode these few coefficients efficiently. The description of Baseline sequential entropy coding is given in two steps: conversion of quantized DCT coefficients into an intermediate sequence of symbols and assignment of variable-length codes to the symbols.

In the intermediate symbol sequence, each nonzero AC coefficient is represented in combination with the “runlength” of zero-valued AC coefficients which precede it in the zigzag. Each such runlength-nonzero coefficient combination is represented by a pair of symbols:

\[
\text{symbol-1} \leftrightarrow (\text{RUNLENGTH}, \text{SIZE}) \quad (1)
\]

\[
\text{symbol-2} \leftrightarrow (\text{AMPLITUDE}) \quad (2)
\]

symbol-1 represents two pieces of information, RUNLENGTH and SIZE; symbol-2 represents the single piece of information designated AMPLITUDE, which is simply the amplitude of the nonzero AC coefficient. RUNLENGTH is the number of consecutive zero-valued AC coefficients in the zigzag sequence preceding the nonzero AC coefficient being represented. SIZE is the number of bits used to encode AMPLITUDE. RUNLENGTH represents zero-runs of length 0 to 15. Actual zero-runs in the zigzag sequence can be greater than 15, so the symbol-1 value (15, 0) is interpreted as the extension symbol with runlength = 16. There can be up to three consecutive (15, 0) extensions before the terminating symbol-1 whose RUNLENGTH value complete the actual runlength. The terminating symbol-1 is always followed by a single symbol-2 except for the case in which the last run of zeros include the last \(63^{rd}\) AC coefficient. In this frequent case, the special symbol-1 value (0, 0) means EOB (end-of-block) symbol, is used to terminate the \(8\times8\) sample block.

A DC difference Huffman code is concatenated by two codes: The first code, \(d_{DC}\), so called difference category, defines the SIZE for the difference value. The second code, \(v_{DC}\), which denotes the additional
bits of the DC difference. Let $\triangle_{DC}$ denote the DC difference, $k$ denotes the SIZE of the DC difference, $B$ denotes the equivalent decimal value of the CODE. If $B \geq 2^{k-1}$, then $\triangle_{DC} = B$. Otherwise, $\triangle_{DC} = B - 2^k + 1$.

The possible range of quantized AC coefficients determines the range of values of both the AMPLITUDE and the SIZE information. If the input data are $N$-bit integers, then the non-fractional part of the DCT coefficients can grow by at most 3 bits. Baseline sequential has 8-bit integer source samples in the range $-2^7, 2^7 - 1$, so quantized AC coefficient amplitudes are covered by integers in the range $[-2^{10}, 2^{10} - 1]$. The signed-integer encoding uses symbol-2 AMPLITUDE codes of 1 to 10 bits in length, hence SIZE represents values from 1 to 10, and RUNLENGTH represents values from 0 to 15 as discussed previously. Figure 1 shows a general structure of the baseline Huffman encoder for DC difference and AC coefficients. The AC encoder is represented by block, which accepts the intermediate symbols, performs all computations include generation the addresses for accessing codes from the code tables. As a result, the encoder generated an appropriated compressed code streams represents for the AC coefficients. The input of the encoder includes a pair of intermediate symbols generated by the zigzag coder, once for each time. If the zero-runs is greater than 15 or the EOB (end of block) is reached, symbol-2 input is omitted. When the zero-runs is greater than 15, the symbol-1 value, (15,0), is interpreted as the extension symbol with runlength is 16. In the case of EOB, the symbol-1 value is (0,0). The AC encoder shares the DC difference code table with the DC coder, because the AC encoder uses this table to get the code for the AMPLITUDE symbol. The encoding for DC coefficients is similar to that of the AC coefficients, except the input for the DC part contains the DC difference category and the DC difference value. There are 11 DC difference categories (excluded the 0 category), and therefore, $2^{11}$ DC difference values. The general structure of the DC and AC Huffman encoders is shown in Figure 2. Figure 2 illustrates the procedure to encode an 8x8 block of DCT coefficients. The top array shows the 8x8 block of DCT coefficients arranged in zigzag order. The second array is the result in the conversion the block into a sequence of symbols. The last line in Figure 2 shows the output bit stream of the 8x8 DCT coefficients. In the presence of hardware or the instruction faults, the code stream output may contain bit errors. The errors may prevent 8x8 code block from decoded, or the reconstructed image for that block is degraded. These errors are due to the encoder itself and channel
control coding cannot correct them. To increase the encoder’s reliability and improve the image quality, the fault tolerance Huffman encoding system is proposed as shown in the next section.

**III. Fault Tolerance Model for Baseline JPEG Encoder**

![Diagram](image)

The method of inserting error-detecting features of the AC Huffman encoder is shown in Figure 3. Failures are modelled as corrupting one or more variable-length code vectors $s_1, s_2$ of the stream output. These corrupted binary code vectors can be expressed as flipped the correct bits randomly. The method is presented for fault detection throughout the data path so that the momentarily failure of any subsystem will not contaminated data to go undetected. The error detection for the code stream output is performed by generating the parity symbols via the spare subsystem connected parallels with the main encoder. These parity symbols are ready for checking as soon the output code stream is emerged. The goal of this fault tolerance design is to protect against at most one subsystem producing erroneous data. In the context of this paper, error detection of such situation is the only requirement and no correction methods will be addressed.

Let $s_1 = b_1 b_2 \ldots b_n$ denotes the code of (RUNLENGTH,SIZE) symbol; $s_2 = c_1 c_2 \ldots c_n$ denotes the code of AMPLITUDE symbol. To reduce the delay in computation the parity symbols, the parity table for the code $s_1$ is created and maintained in memory storage throughout the encoding process. Assume the code table for $s_1$ was created by using the statistic model [1] (pp. 510-16). The correspond parity for $s_1$ is shown in Tab. I. A parity symbol can be retrieved quickly when a (RUNLENGTH,SIZE) symbol is valid at the address decoder for the memory storage (I). The details of how to implement the address decoder is not discussed in this paper, however, based on the (RUNLENGTH,SIZE) code table, the address is well known to be unique.
TABLE I
PARIITY SYMBOL TABLE OF THE CODES OF (RUNLENGTH, SIZE) SYMBOLS

<table>
<thead>
<tr>
<th>RUNLENGTH</th>
<th>SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 1 1 0 0 0 1 1 0 0 0 0 0 0</td>
<td>0 0 1 0 1 1 0 0 0 1 1 0 1 1 1 0 0</td>
</tr>
<tr>
<td>1 1 1 0 0 1 0 1 1 1 1 0 1 1 0 1 1</td>
<td>1 1 1 1 0 1 0 1 0 0 1 1 1 0 0 1 0 0</td>
</tr>
<tr>
<td>1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 1</td>
<td>0 0 0 1 1 0 1 0 0 1 1 1 0 0 1 0 0 0</td>
</tr>
<tr>
<td>1 1 0 1 0 1 0 0 0 1 1 1 0 1 1 1 1 0 1 1 0</td>
<td></td>
</tr>
<tr>
<td>0 1 1 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0</td>
<td></td>
</tr>
<tr>
<td>1 0 0 1 0 1 0 1 0 1 1 0 0 0 0 0 1 1 1 1</td>
<td></td>
</tr>
</tbody>
</table>

The memory storage (II) shares the same address decoder with the storage (I), because the memory storage (II) contains the length of concatenated code $s_1s_2$. Both have the same number of items and have a one-by-one relationship. As far the (RUNLENGTH, SIZE) symbol is known, the length of the $s_1s_2$ code is determined by

$$\text{length}(s_1s_2) = \text{length}(s_1) + \text{length}(s_2)$$

$$= \text{length}(s_1) + \text{SIZE}$$

In this context, the SIZE represents for the code length of nonzero AC coefficient. The length parameter is used to determine the parity check points for the code stream output. When the code stream for an 8x8 block is continuously coming out, it is virtually partitioned into code segments following a certain rule. The length memory storage is used to determine the endpoints of these variable length codes.

The storage (III) is used to hold the parity of $s_2$ codes. There are 2047 different codes for $s_2$, which represent for the AC values in the range [-1023, 1023] [1] (pg. 445). The code $s_2$ is computed similar to that of the code $v_{DC}$ [1] (pg.191). The computed parities for the $s_2$ codes are stored in the storage (III). Figure 4 shows the construction of the code $s_2$. The zero AC value is unused in this construction due to the zigzag encoding result. Therefore the memory cell at the address 1023 is empty. However the existence of this cell will save some operations for the address decoder. The parity of the code

AC values: 1023 0 1 2 3 4 5 6 7
Code length: 10 1 1 1 1 1 1 1 0
Code value: 0 1 1 1 1 1 1 1 1
Parity bit: 0 1 1 1 1 1 1 1 1

Fig. 4. The parity bit structure of non-zero AC coefficients code. This is the pre-computed parity bits are stored in memory and will be retrieved uniquely by the AMPLITUDE symbol.

stream represents for (RUNLENGTH, SIZE)+AMPLITUDE symbol pair is determined by combine the parity retrieved from the memory storages (I) and (III). This parity is used to compare with the parity of the code stream generated by the Huffman encoder. The check point is determined by the length parameter retrieved from the memory storage (II). The AC Huffman encoder is used to compute the code stream for the (RUNLENGTH, SIZE)+AMPLITUDE pair using the Huffman code table and the built-in arithmetic...
The operation principle of the AC Huffman encoder is known from the JPEG compression standard.

Let \( s_1 = b_1 b_2 \ldots b_{n_1} \) be the code of a (RUNLENGTH,SIZE) symbol; \( s_2 = c_1 c_2 \ldots c_{n_2} \) be the code of an AMPLITUDE symbol. The parity symbols of these codes are pre-computed and stored in memory (I) and (III) as shown below:

\[
p_1 = b_1 \oplus b_2 \oplus \ldots \oplus b_{n_1} \quad (4a)
\]
\[
p_2 = c_1 \oplus c_2 \oplus \ldots \oplus c_{n_2} \quad (5a)
\]

where \( 2 \leq n_1 \leq 16 \) and \( 1 \leq n_1 \leq 10 \). Thus there is required five bits to store the total code length \( n \) (\( n = n_1 + n_2 \)). The overall parity of the code sequence \( b_1 b_2 \ldots b_{n_1} c_1 c_2 \ldots c_{n_2} \) is determined by \( p = p_1 \oplus p_2 \).

Consider a specific pair of symbols (RUNLENGTH,SIZE) and AMPLITUDE resulting from the zero runlength coding of a zigzag sequence. As soon this symbol pair is available at the input of the Huffman encoder, the parity bit \( p \) and the length \( n \) is retrieved from the memory storages and ready for parity checking. If error free, the parity \( \tilde{p} \) of the code stream \( \tilde{b}_1 \tilde{b}_2 \ldots \tilde{b}_{n_1} \tilde{c}_1 \tilde{c}_2 \ldots \tilde{c}_{n_2} \) generated by the encoder must identical to the stored parity \( p \). When this code stream sequentially pulling out, the parity bit for that stream is dynamically computed. The control signal use the length \( n \) to set the comparison point for the code stream output, where the comparison is taken place. The error flag is only active at that check points. The structure of the error checking for the output bit stream is shown in Figure 5, where the parity

\[
\tilde{p}_1 = \tilde{b}_1 \quad (6a)
\]
\[
\tilde{p}_2 = \tilde{p}_1 \oplus \tilde{b}_2 \quad (6b)
\]
\[
\vdots \quad (6c)
\]
\[
\tilde{p}_{n_1} = \tilde{p}_{n_1-1} \oplus \tilde{b}_{n_1} \quad (6d)
\]
\[
\tilde{p}_{n_1+1} = \tilde{p}_{n_1} \oplus \tilde{c}_1 \quad (6e)
\]
\[
\vdots \quad (6f)
\]
\[
\tilde{p} = \tilde{p}_{n_1} \oplus \tilde{c}_{n_2} \quad (6g)
\]
When the length of the stream is matched with the pre-stored length, the comparison is conducted. Assume the synchronization is correct, then the two parity bits will be matched, if no error has occurred in the encoding process. Otherwise, the error flag is turned on and sent re-encode repeat request to the control center.

IV. EXPERIMENT RESULTS

The protection designed for Huffman encoder was implemented using a computer software resources. The computer program, first, performed the image reading, color component transformation, DCT and Huffman coding processes for both encoding and decoding. The error detection was implemented restricted only in the encoder side. The error source corrupted the system by changing data bits of the code stream at random locations. The error rate is varies is a typical range from $10^{-4}$ to $10^{-1}$. Since the error correction is not implemented in this paper, to improve the quality of image in the existence of error, the alternate form of error correction is repeat encoding for the corrupted blocks. The error correction performance is evaluated based on the Mean Square Error (MSE) between the reconstructed and the original data images. Figure 6 shows the quality performance curves of the Huffman encoder with and without error correction. The dash line and solid line curves present the reconstruction performance of image with and without repeat encoding request, respectively, versus error rate injection. As a matter of fact, the fidelity is much improved when the error correction system is implemented. Figure 7 shows the reconstructed of color images in the effects of corrupted blocks. The four particular error images with coding errors with error rates $10^{-4}$ (top-left), $10^{-3}$ (top-right), $3 \times 10^{-3}$ (bottom-left), and $3 \times 10^{-2}$ (top-right) are reconstructed. Each color square is a result of discarding the bad code blocks. In contrast with the above results, Figure 8 shows the reconstructed of images in the same error rates, but with repeat encoding request. Each corrupted block is discarded and replaced by a re-encoded block. This method reduced the number of error spots with the cost of delay.

![Figure 6. Quality performance curves with and without repeat encoding request.](image-url)
V. CONCLUSIONS

In this paper, the error detection and correction for JPEG Huffman entropy coding was implemented successfully in software. The computer program takes the color input images, performs color transform, DCT, scalar quantization and Huffman coding. Although the implementation only detects single error occurred in a code word, the design principle can be expanded to solve for the complicated problems. In the future research, error control coding techniques will be employed in order to generate more sophisticated coding techniques for detection and correction difference types of errors. The simulation results show that when repeat encoding process is performed on the corrupted part of the code stream, the significant visual quality improvement is perceived in the decoded image.

REFERENCES