Fast Split Arithmetic Encoder Architectures and Perceptual Coding Methods for Enhanced JPEG2000 Performance[1]

Krishnaraj M. Varma

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Electrical Engineering

Committee:

Dr. Amy E. Bell, Chairperson
Dr. A. A. (Louis) Beex
Dr. Chris L. Wyatt
Dr. Ting-Chung Poon
Dr. Roger W. Ehrich

March 17, 2006
Blacksburg, VA

Keywords: JPEG2000, Perceptual weighting, EBCOT, Split Arithmetic Encoder, Multithreaded software, FPGA hardware

Fast Split Arithmetic Encoder Architectures and Perceptual Coding Methods for Enhanced JPEG2000 Performance

Krishnaraj M. Varma

(ABSTRACT)

JPEG2000 is a wavelet transform based image compression and coding standard. It provides superior rate-distortion performance when compared to the previous JPEG standard. In addition JPEG2000 provides four dimensions of scalability—distortion, resolution, spatial, and color. These superior features make JPEG2000 ideal for use in power and bandwidth limited mobile applications like urban search and rescue. Such applications require a fast, low power JPEG2000 encoder to be embedded on the mobile agent. This embedded encoder needs to also provide superior subjective quality to low bitrate images. This research addresses these two aspects of enhancing the performance of JPEG2000 encoders.

The JPEG2000 standard includes a perceptual weighting method based on the contrast sensitivity function (CSF). Recent literature shows that perceptual methods based on subband standard deviation are also effective in image compression. This research presents two new perceptual weighting methods that combine information from both the human contrast sensitivity function as well as the standard deviation within a subband or code-block. These two new sets of perceptual weights are compared to the JPEG2000 CSF weights. The results indicate that our new weights performed better than the JPEG2000 CSF weights for high frequency images. Weights based solely on subband standard deviation are shown to perform worse than JPEG2000 CSF weights for all images at all compression ratios.

Embedded block coding, EBCOT tier-1, is the most computationally intensive part of the JPEG2000 image coding standard. Past research on fast EBCOT tier-1 hardware implementations has concentrated on cycle efficient context formation. These pass-parallel architectures require that JPEG2000's three mode switches be turned on. While turning on the mode switches allows for arithmetic encoding from each coding pass to run independent of each other (and thus in parallel), it also disrupts the probability estimation engine of the arithmetic encoder, thus sacrificing

coding efficiency for improved throughput. In this research a new fast EBCOT tier-1 design is presented: it is called the Split Arithmetic Encoder (SAE) process. The proposed process exploits concurrency to obtain improved throughput while preserving coding efficiency. The SAE process is evaluated using three methods: clock cycle estimation, multithreaded software implementation, a field programmable gate array (FPGA) hardware implementation. All three methods achieve throughput improvement; the hardware implementation exhibits the largest speedup, as expected.

A high speed, task-parallel, multithreaded, software architecture for EBCOT tier-1 based on the SAE process is proposed. SAE was implemented in software on two shared-memory architectures: a PC using hyperthreading and a multi-processor non-uniform memory access (NUMA) machine. The implementation adopts appropriate synchronization mechanisms that preserve the algorithm's causality constraints. Tests show that the new architecture is capable of improving throughput as much as 50% on the NUMA machine and as much as 19% on a PC with two virtual processing units. A high speed, multirate, FPGA implementation of the SAE process is also proposed. The mismatch between the rate of production of data by the context formation (CF) module and the rate of consumption of data by the arithmetic encoder (AE) module is studied in detail. Appropriate choices for FIFO sizes and FIFO write and read capabilities are made based on the statistics obtained from test runs of the algorithm. Using a fast CF module, this implementation was able to achieve as much as 120% improvement in throughput.

ॐ असतोमा सद्गमय ।
तमसोमा ज्योतिर्गमय ।
मृत्योर्मा अमृतंगमय ।
ॐ शान्तिः शान्तिः शान्तिः ॥

From illusion lead me to the real;

From darkness lead me to light;

From death lead me to immortality;

And may there be Peace..Peace..Perfect Peace.

To my parents for

all their love,

all their patience, and

all their encouragement.

Acknowledgement

I would like to express my most sincere gratitude to Dr. Amy Bell for her guidance during the course of this research work and my academic career at Virginia Tech. Without her invaluable advice, help, suggestions and planning, this dissertation work would not have been possible. Working in the Digital Signal Processing and Communications Lab (DSPCL) at Virginia Tech has been an invaluable experience. My experience at DSPCL has improved my technical knowledge and research skills and broadened my understanding of many subjects, not just electrical engineering. For this opportunity I am deeply indebted to Dr. Bell. Dr. Bell has also been an inspiration for me on learning the virtues of planning and time management. I would also like to thank her for the financial assistantship that I was offered during the course of my doctoral research. There have been many occassions when she went out of her way to help me administratively and personally and make my stint at Virginia Tech as comfortable as possible. Above all, I am deeply thankful for her friendship.

This research would not have been possible without the collaboration of our hardware partners from the University of Akron. Dr. Joan Carletta and her student Himabindu Damecharla have been extremely helpful and cooperative in helping us understand the intricacies of hardware implementations on FPGAs. In addition their work in implementing the various EBCOT tier-1 process models on FPGA and running simulations to obtain throughput results were central to the success of this research. Dr. Godmar Back from the Computer Science department at Virginia Tech provided me with invaluable help during the multithreaded software implementation of the split arithmetic encoder process. His timely insights into multithreading were extremely useful in making informed decisions for designing the software architectures and for measuring the performance of the implementation.

I would like to offer special thanks to Dr. A. A. (Louis) Beex for his guidance and help during the first two years of my graduate study at Virginia Tech and for offering me an opportunity to teach the DSP course in Spring 2005. I have to admit that the most I have learned about signal processing concepts was when I taught that course. Many thanks also to Dr. Chris L. Wyatt, Dr. Ting-Chung Poon, and Dr. Roger W. Ehrich for being on my graduate committee and reviewing this work.

I would like to express my appreciation for the endless hours of discussion, technical and other-

wise, that I have had with my colleagues and friends Kishore Kotteri, Satyabrata Rout, and Takeshi Ikuma. These discussions have helped shape my graduate research during my tenure at Viginia Tech and their friendship is something I will cherish for the rest of my life. In addition, I would like to thank Sunil Mallik, Vidya Rajagopalan, Ramya Ramanath, Anbumani Subramanian and Sheela Ramamoorthy for their wonderful friendship without which my stay in Blacksburg would have been far less memorable.

My parents have been there for me throughout my good and bad times. They have always provided me with encouragement, love, patience and guidance. Without their support my entire professional life would have been impossible. Therefore I dedicate this dissertation with love to my parents. Finally I would like to thank my wife Reshmi whose love and help have been indispensable in the last few weeks of writing up this dissertation. She has been very kind and understanding and I hope that some day I can be even half as supportive to her as she has been to me.

Krishnaraj M. Varma

Contents

List of Figures

List of Tables

Chapter 1

Introduction

JPEG2000 is a new standard [2] for the coding of bi-level, gray-level, color and multicomponent images. Similar to the original JPEG standard, the JPEG2000 standard was born out of a collaboration between the International Standards Organization (ISO) and the United Nations based International Telecommunications Union (ITU). JPEG2000 offers improved performance while coding different types of images like natural, remote-sensing, scientific and medical images as well as text and rendered graphics. The standard also allows for different image application models like: client/server, real-time transmission, image library archival, limited buffer and bandwidth resources, etc. all within one unified coding system. JPEG2000's superior performance is evident in providing lower bit-rates at the same rate-distortion and subjective quality levels as JPEG [3].

JPEG2000 is ideally suited for image transmission in power or bandwidth limited mobile environments. An urban search and rescue (USR) effort is an example of such an environment. In the event of an urban disaster such as the collapse of a building, a small mobile rescue "agent" can capture and transmit images from voids too small or dangerous for rescue workers. A remote "client" station outside the void maneuvers the wireless mobile agent in its search for survivors. The client receives and processes the images from the mobile agent wirelessly. Since images are bandwidth intensive, the performance of the compression system becomes critical. JPEG2000 offers the best performance at low bitrates. In addition it provides several forms of scalability. Because of power and bandwidth related constraints in a mobile environment the client could request a new image, initially at either a lower resolution or lower quality. The client would like to obtain detailed high quality or high resolution images only when required. Most of the new images requested by the

client would not contain images of incapacitated human bodies and therefore would not be of any interest. It is only when the client receives an image of interest that it would need to request a higher resolution or higher quality image.

JPEG2000 is well suited for such embedded client-server applications. During the initial request for a new image, the mobile agent could capture and losslessly encode the image into a JPEG2000 compatible file, but could send only the topmost quality or resolution layer. If a repeated request for higher quality or higher resolution image is received, the mobile agent could send the rest of the file. This way the limited bandwidth and power of the mobile agent are utilized in the most optimal fashion. Such a system requires a fast and low power JPEG2000 encoder to be installed on the mobile agent. Since most of the initial requests would be made for low bit rate images, this encoder must ensure the best perceptual quality possible for such images. A field programmable gate array (FPGA) is an ideal hardware platform for implementing small, fast, low power architectures of a JPEG2000 encoder. The architecture design must maximize hardware performance (throughput, power, etc.) while maintaining JPEG2000's coding efficiency.

1.1 Wavelet Transform Based Image Coding – A Brief History

When the original JPEG standard was adopted in the 1980's, wavelet based image compression research was in an early stage. Almost all of the wavelet research that had been performed was based on subband coding techniques. The performance of wavelet transform based systems was, at best, comparable to the Discrete Cosine Transform (DCT) based subband coding systems. Since the DCT was well understood it was decided to adopt a DCT based system for the original JPEG standard.

Subband coding techniques involve the same basic steps whether it is DCT based or wavelet based. It involves the transformation of the image into a set of subbands using a linear, decorrelating and, if possible, energy-preserving transform. The coefficients in each subband are then quantized using a given number of bits. The number of bits allocated to all the coefficients in a subband is the same and is decided by some optimal bit-allocation algorithm that minimizes the mean squared error between the quantized and unquantized coefficients. The quantized coefficients are then entropy coded to generate the compressed bit stream. It was this kind of simplistic subband

coding technique that was limiting the performance of wavelet transform based systems.

When Shapiro introduced the Embedded Zero-tree Wavelet (EZW) coding algorithm [4], it revolutionized the performance of wavelet based image compression systems. The EZW algorithm was different from simple subband coding in that it recognized that significant coefficients are located in the same relative positions in each subband. This observation led to encoding only the significant coefficients. The overhead information required to locate the positions of the significant coefficients was efficiently coded by noting only the locations of the parents of significant trees. The introduction of EZW and later algorithms based on EZW, like Set Partitioning In Hierarchical Trees (SPIHT) [5], significantly improved the performance of wavelet based methods over DCT based methods. This precipitated the need for a new image coding standard based on wavelet transforms and the idea of JPEG2000 was born.

1.2 Perceptual Coding

The use of a wavelet transform in the new standard makes it well suited to perceptually optimize the compression of images. The initial processing of visual stimuli in the cortex of the human brain is modeled as a series of bandpass filters covering the entire spatial frequency spectrum in several different orientations. This allows the brain to separate information into these subbands in various orientations for further processing. A cortical model by Watson [6] approximates the non-overlapping filter bandwidths to 1 octave and the orientations to 45°. This means that properties of human vision, such as sensitivity to error, vary in frequency bands that are separated by 1 octave. Such a model fits well with a wavelet decomposition where the subbands are separated by 1 octave. Thus the properties of the human visual system (HVS) were easily incorporated into the JPEG2000 standard.

Two such HVS properties, namely, contrast sensitivity and visual masking were integrated into the JPEG2000 standard. Contrast sensitivity refers to the sensitivity of the HVS to spatial frequencies on a smooth background. Visual masking refers to the perceptual hiding or "masking" of errors in the vicinity of high activity within a scene. Contrast sensitivity was incorporated as a set of perceptual weights which scale the wavelet coefficients within a subband according to their HVS importance. This method is relatively straightforward to implement in hardware. On the other

hand visual masking was incorporated by additionally scaling each individual wavelet coefficient by a factor determined from the local activity surrounding the coefficient. Though this method improves the perceptual quality of images it increases the complexity of the encoder—particularly in the context of a hardware implementation.

The problem with the simple contrast sensitivity based scaling method is that when compression artifacts become visible at low bit rates, this method does not take into account the difference in sensitivity between the horizontal and vertical wavelet subbands. There is a need to design perceptual weights that are suitable for low bit rates and that account for contrast sensitivity as well as orientation sensitivity.

1.3 Fast EBCOT Architectures

The entropy coder for JPEG2000 is called the embedded block coder with optimal truncation (EBCOT). It differs from JPEG's encoder in that the division into independent non-overlapping code-blocks is done after the transform instead of before the transform. EBCOT is logically divided into two tiers. In tier-1 each code-block is independently entropy coded using a finite set of coding contexts and a binary multiplierless arithmetic encoder which uses a finite state machine to adapt symbol probabilities. In EBCOT's tier-2 each encoded bitstream is optimally truncated such that an overall desired bit rate is achieved. The entropy coder in EBCOT is the most computationally intense part of the JPEG2000 algorithm [7,8]. Consequently, significant effort has been devoted to developing fast hardware architectures for it.

Most of the research has concentrated on pass-parallel architectures in which the three coding passes of EBCOT are run in parallel. Such architectures take advantage of JPEG2000's mode switches in that they tradeoff speed (parallel implementation) for coding efficiency. Simulation results indicate that up to 0.7 dB of quality degradation is realized with pass parallel architectures. There is a need to develop architectures that are fast, but also preserve coding efficiency. Also, the previous research has concentrated on improving the cycle efficiency of the context formation (CF) module. Research on the complexities that arise from the unmatched rate of production of symbol-context pairs by the CF module and rate of consumption of symbol-context pairs by the arithmetic encoder (AE) module has been less common. This research aims to be more comprehensive by

solving problems that arise from this mismatch and also design architectures that maximize the overall throughput of the entire encoding system while maintaining coding efficiency.

1.4 Contributions

This research enhances the performance of JPEG2000 encoding in two ways: (1) design new perceptual weights that are easy to implement and that improve the perceptual quality of the compressed image and (2) design new EBCOT tier-1 architectures that improve encoding throughput while not sacrificing coding efficiency. The following are the specific contributions of this work.

1. New perceptual weights that improve the subjective quality of images compressed by JPEG2000 while maintaining simplicity of implementation for fast hardware implementations.

2. The new Split Arithmetic Encoder (SAE) process model for EBCOT tier-1 that improves encoding throughput while preserving a substantial portion of JPEG2000's coding efficiency.

3. A multithreaded software realization and evaluation of the SAE process.

4. Results that indicate that the software implementation of the SAE process achieves approximately 50% improvement in throughput (when compared to the conventional serial implementation) on a multi-processsor computing system.

5. An FPGA based multirate hardware realization of the SAE process (in collaboration with our hardware research partners). This implementation demonstrates 120% improvement in throughput compared to the conventional serial implementation.

1.5 Organization

This document is organized as follows. Chapter 2 presents an overview of the JPEG2000 image compression system. It lists the features of the standard and also provides brief details of the steps involved in the compression process. Chapter 3 delves into details of perceptual coding techniques. It concentrates on the three broad categories of methods that have been used in the literature to design perceptual weights for wavelet based compression systems. Chapter 4 discusses the details

of the design of new perceptual weights that improve the subjective quality of compressed images and presents the human subject test results obtained using the new weights.

Chapter 5 provides a detailed description of the EBCOT entropy coding algorithm and reviews some of the important hardware implementation research. This chapter also discusses some of the challenges involved in designing hardware—specifically the loss of coding efficiency in pass-parallel architectures. Chapter 6 proposes the new parallel EBCOT process called the split arithmetic encoder (SAE) process that improves speed without much loss of coding efficiency. Chapter 7 describes a multithreaded software realization of the SAE process. The results from encoding test images are presented and compared to results from a conventional serial implementation. Chapter 8 describes an FPGA based multirate hardware realization of the SAE process along with processing time and throughput results. Finally Chapter 9 summarizes the conclusions from the research.

Chapter 2

Fundamentals of JPEG2000 Image Compression

This chapter presents the fundamental principles of JPEG2000 encoding. First it briefly describes the steps involved in the JPEG2000 encoding process and follows it with a more detailed high-level description of EBCOT. Finally a comparison of EBCOT with the earlier SPIHT is presented.

2.1 Features of JPEG2000

1. **Superior Low Bit-Rate Performance**: The standard provides superior performance (MSE and perceptual quality) at low bit-rate (below 0.25 bits-per-pixels or bpp) when compared to existing standards. On the average JPEG2000 outperforms the older JPEG standard by 2 dB peak signal to noise ratio (PSNR) for several images across all compression ratios [3].

2. **Continuous-Tone and Bi-level Compression**: The standard compresses both bi-level and continuous-tone images. Bi-level images use one bit per pixel indicating whether the pixel is black or white. Scanned versions of legacy text publications are a good example. Continuous tone images use several bits per pixel. The standard is also capable of handling images with these kinds of regions mixed together, like a natural image with text in a corner.

3. **Lossy and Lossless Compression**: The standard provides for both lossless (completely reversible) and lossy (irreversible) compression.

4. **Scalability**: JPEG2000 allows encoders to code the image once and allows decoders to reconstruct the image in a scalable fashion. For example the decoder can reconstruct the image at lower resolutions, lower bit-rates, individual color components or interested regions.

5. **Region of Interest Coding**: This feature allows the user to define certain regions in the image that are of more interest than other regions, thus allowing these regions to be transmitted at a higher quality.

6. **Robustness to Bit Errors**: This feature allows the images to be encoded so that the important portions of the bit stream are protected from bit errors when it is being transmitted through a lossy channel like a wireless channel.

7. **Protective Image Security**: Copyright protection of digital images can be achieved by means of methods like watermarking, labeling, stamping, encryption etc. The JPEG2000 standard allows for such copyright protection to be included.

2.2 Algorithm Steps

Figure 2.1 shows the JPEG2000 encoding algorithm in block diagram form. A brief description of each block is given below.

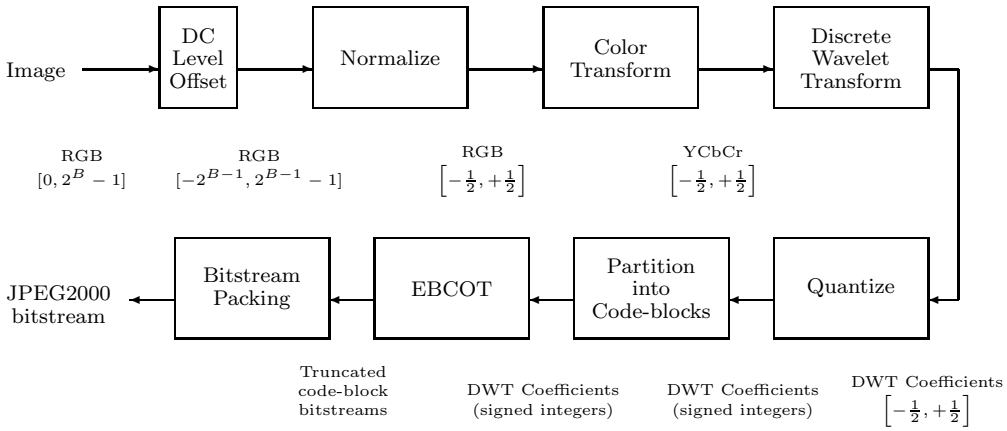


Figure 2.1: *Block diagram of the JPEG2000 coding algorithm: an image is converted into a JPEG2000 encoded, embedded bitstream.*

1. **DC Level Offset**: If a color component of the image has an unsigned positive dynamic range of B bits, then a value of $2^{(B-1)}$ is subtracted from each pixel value of that color component. This DC level shifting is done independently for each color component. The bit depth, B, is stored in the compressed codestream. This DC level shifting is done so that all subbands

(including the LL) have both positive and negative values. Otherwise the LL subband would be an exception to this. However the DC level offset is an optional, though typical feature in the standard.

2. **Normalization**: This step involves the scaling of values in each component by dividing them by 2^B. This converts all the pixel values to the range $\left[-\frac{1}{2}, \frac{1}{2}\right]$. This step is only done when using the irreversible wavelet transform because the transform is linear. The non-linearities in the reversible (integer) wavelet transform prevent us from scaling the pixel values. This kind of scaling makes the compressed data independent of the original bit depth of the image. For example an original 12 b/p image that was compressed using JPEG2000 can be decoded as an 8 b/p image by changing the value of B while decoding.

3. **Color Transform**: This step converts the normalized RGB color components (if image is in color) to a luminance and two color-difference (chrominance) components. The three-element RGB values for each pixel can be converted to three-element YCbCr values using a matrix multiplication. The output YCbCr values are guaranteed to be in the range $\left[-\frac{1}{2}, \frac{1}{2}\right]$. But in the decoding phase, the quantized YCbCr values are not guaranteed to be in the same range. Also, even if the YCbCr values are within range, the reverse color transform to RGB is not guaranteed to be within this range. The standard does not discuss how to deal with these out-of-range values, but decoders can clip these values to the nearest values within range in the RGB domain. For the case of the reversible compression, a different integer color transform has been defined which works on the DC level shifted (un-normalized) pixel values and gives transformed integer values.

4. **Wavelet Transform**: The JPEG2000 standard uses 2-dimensional, separable, non expansive, symmetric extension wavelet transforms. In the case of irreversible compression, the standard uses the bi-orthogonal 9/7 wavelet and in the case of reversible compression it uses the 5/3 integer wavelet to perform the transforms. The number of levels of wavelet decomposition can range from 0 to 32 with 5 being the typical value.

5. **Quantization**: This step refers to the conversion of subband coefficients, typically real numbers in the range $\left[-\frac{1}{2}, \frac{1}{2}\right]$, to a set of integers. The quantization is called deadzone scalar quantization. A quantization step size is selected for each subband. The magnitudes of the

coefficients of that subband are divided by the step-size and fixed to the nearest integers. The signs of the coefficients are then multiplied to the integers. The step-size for the zero bin is twice the size of the other bins. Step sizes are transmitted in the codestream in two ways. Either the step-size for each subband is transmitted explicitly or only the step-size for the LL subband is transmitted and all other step-sizes are derived from this. A good review of the quantization strategy in JPEG2000 is provided by Marcellin et. al. [9]

6. **Partitioning**: After we obtain the quantized transform coefficients, we divide the coefficients at each resolution into precincts. The precincts contain the coefficients from the same regions of each of the subbands at that resolution. The origins and sizes of each precinct are stored in the compressed bit-stream. This is a conceptual division which will be used later to pack data in useful ways. Each subband is also divided into non-overlapping code-blocks. Thus each precinct at a particular resolution contains a code-block from each of the three subbands.

7. **EBCOT**: Embedded Bit-plane Coding with Optimal Truncation (EBCOT) is the name of the algorithm that encodes the quantized coefficients into a codestream of desired length. The bit-planes from each code-block are coded independently using a modified Q–arithmetic coder (MQ coder). A bit-plane is the set of bits at the same bit-position for all the quantized coefficients in the code-block. The independent bitstreams corresponding to each code-block are optimally truncated so that the total length is equal to or less than the desired length. Details of this coding algorithm are given in Section 2.3.

8. **Codestream**: The codestream consists of a header and a number of data-blocks called layers. The header contains general information about the image, like its size, bit depth, type of transform, number of levels of decompositions etc. Each layer contains all the coded bits required to decode the image at a particular quality. Each succeeding layer adds to the quality of the reconstructed image. A layer is made up of a number of smaller data-blocks called packets. Several bit-planes from all code-blocks from each precinct at each resolution from each color component forms a particular packet. Different code-blocks may have different number of bit-planes coded up within a packet. The distribution of the number of bit-planes across layers for each code-block is determined by the optimal bit-allocation algorithm. The distribution of coded bit-planes into layers and packets as described above is only one of

several ways to order the data. This order provides distortion scalability. Ordering the data in other ways can provide other kinds of scalability.

2.3 EBCOT

Embedded Block Coding with Optimized Truncation (EBCOT) is the name given to the entropy encoder in the JPEG2000 standard. EBCOT accepts as input a set of signed integers representing the quantized wavelet coefficients. Each subband of the image in the wavelet domain is divided into a set of code-blocks. Typical sizes of such code-blocks are 32×32 or 64×64. Figure 2.2 depicts the block diagram of the EBCOT algorithm.

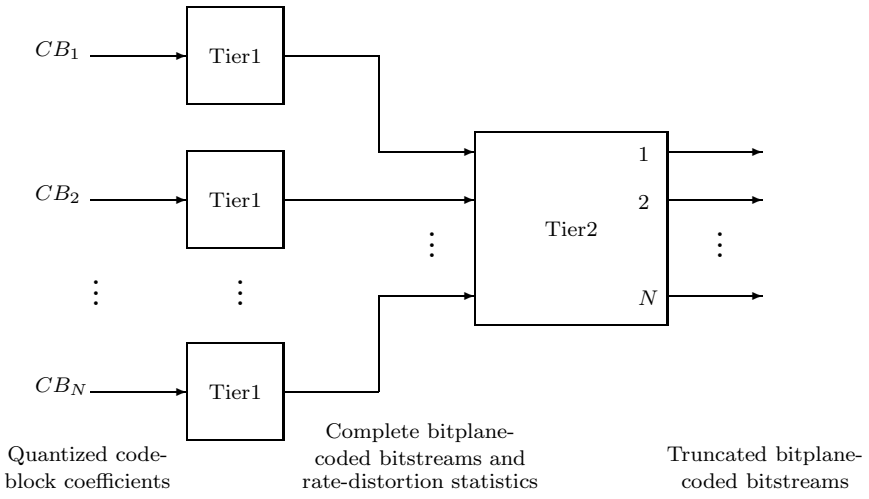


Figure 2.2: *Block diagram of EBCOT: the quantized wavelet coefficients from each code-block are independently encoded in tier-1 and the overall rate-distortion statistics are used by tier-2 to optimally truncate each of the tier-1 bitstreams.*

EBCOT tier-1 accepts the quantized code-block coefficients and outputs a complete coded bitstream as well as rate-distortion statistics. EBCOT tier-1 independently processes each code-block and generates its bitstream. EBCOT tier-2 inputs the bitstreams and rate-distortion statistics from all the code-blocks and generates optimally truncated bitstreams (for each code-block) such that the total length of all the bitstreams is within the desired rate.

2.3.1 EBCOT Tier-1: Bit Plane Coding

The block diagram of EBCOT tier-1 is depicted in Figure 2.3. The quantized coefficients from a code-block are input to the context-adaptive bitplane coder and converted to symbols and context labels. This is referred to as the context formation (CF) task. Next, the MQ coder, a type of arithmetic coder, uses an adaptable, finite set of probabilities to code the symbols. This is referred to as the arithmetic encoder (AE) task. A symbol is either a 0 or a 1 and it represents the bitwise information that is being encoded. For example in certain coding passes the symbol represents the sign of a coefficient and in other coding passes it represents the value of a certain bit in a coefficient. Context labels are numbers that represent one of 19 coding contexts that are used to update the probabilities stored in internal state registers. The context state file contains information useful to perform the probability updates.

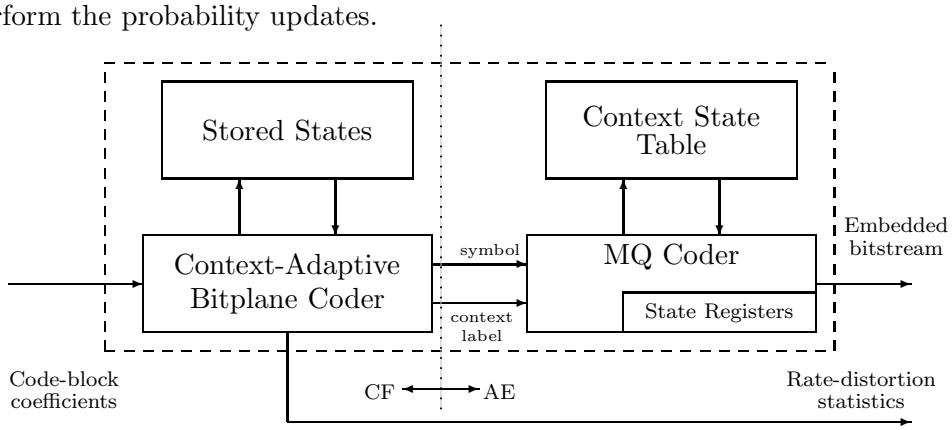


Figure 2.3: *Block diagram of EBCOT tier-1. CF is the context formation task and AE is the arithmetic encoding task.*

The Context Formation Task

The CF task traverses the code-block coefficients one bitplane at a time starting at the highest (most significant) and moving to the lowest. Bitplane K is defined as the set of the K^{th} bits in the binary representation of the code-block coefficients. To understand the concept of a bitplane, consider a two dimensional array of coefficients from a code-block. Assume that each coefficient is represented in binary form using a specific number of bits. Assume that these bits form the third (vertical) dimension of the representation of the code-block coefficients with the most significant

bits at the top. Then each horizontal cross-section of this representation consists of a plane of bits and is called a bitplane.

For each bitplane, the coder executes three coding passes: significance propagation, magnitude refinement, and cleanup. Each coding pass follows a fixed traversal scheme to process all the bits in the bitplane. The traversal rule is illustrated in Figure 2.4. Each bitplane is divided into stripes. Each stripe is a set of 4 rows of coefficient bits. The bitplane is traversed one stripe at a time from top to bottom. Each stripe is traversed one column at a time from left to right. Each column, in turn, is traversed one bit at a time from top to bottom.

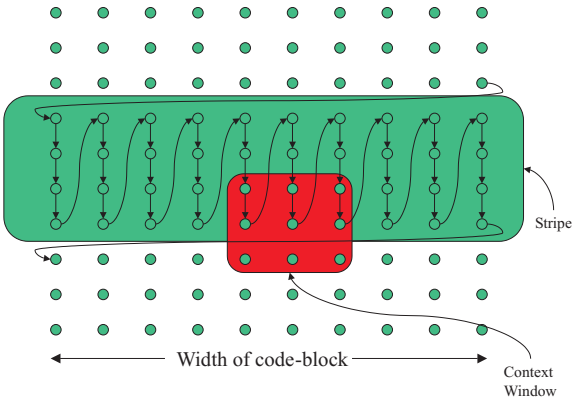


Figure 2.4: *Traversal within a code-block. Code-block: one bitplane at a time from highest to lowest; bitplane: one stripe at a time from top to bottom; stripe: one column at a time from left to right; column: one bit at a time from top to bottom.*

The context-adaptive bitplane coder processes each bitplane based on coefficient significance. A coefficient becomes significant at bitplane K if its bit value at bitplane K is the most significant non-zero bit of the coefficient. The stored states, shown in Figure 2.3, are used to keep track of coefficient significance information. Each coding pass is briefly described below.

0) Significance Propagation Pass (Pass 0): This pass processes all previously insignificant coefficients. The significance of a coefficient is determined based on the significance of its eight nearest neighbors in the context window (shown in Figure 2.4). The coefficient is marked significant if at least one of its neighbors is significant. A symbol of 1 (significant) or 0 (insignificant) and a context-label are sent to the AE task. In addition, if the coefficient has become significant, its sign (0 for positive or 1 for negative) along with a context-label is also sent to the AE task.

1) Magnitude Refinement Pass (Pass 1): For all coefficients that became significant in a previous bit-plane, a magnitude refinement bit and a context-label are sent to the AE task. The magnitude refinement symbol is the bit value of the coefficient at the current bitplane.

2) Cleanup Pass (Pass 2): After the first two passes, coefficients that have just become significant, but whose eight neighbors are still insignificant, have not yet been coded. This pass codes these coefficients and completes the information needed to reconstruct the bitplane. The information sent to the AE task is the same as in the significance propagation pass. In addition, this pass has a "run" mode which performs run-length encoding when all four coefficients present in one stripe column are insignificant.

The Arithmetic Encoder Task

The AE task performs binary, multiplierless arithmetic coding to efficiently encode the symbols that it receives [2]. The context label lends extra meaning to the symbol being encoded. For example, a 1 symbol that originates from a significance propagation pass is different from a 1 symbol generated by a magnitude refinement pass. Consequently these two 1 symbols are accompanied by distinct context labels and are encoded differently due to the disparate coding contexts. The previous sequence of 1s and 0s dictates the current state of a coding context. These context states are stored in the context state table. The coding context state uniquely determines the probability of the "less" probable symbol used by the arithmetic encoder. The state of a coding context is updated using a fixed state transition table. This probability estimation and arithmetic encoding method is called MQ coding.

2.3.2 EBCOT Tier-2: Optimal Truncation

After the entire code-block has been encoded, the next step is to find a set of optimal truncation points for all the code-blocks. One optimal truncation point is assigned to each code-block. The set of optimal truncation points gives an overall (i.e. all code-blocks combined) bitstream length that is within the desired rate. The desired bitstream length is obtained from the desired compression ratio or from the channel bandwidth characteristics.

The optimal truncation point is chosen from the candidate truncation points (L_i^z) and corresponding distortions (D_i^z). Here i represents a code-block and z represents an index to the

truncation point. The candidate truncation points are the lengths of the AE bitstreams at the end of each coding pass. The corresponding distortions are the mean squared errors between the original quantized coefficients of the code-block and the lower-precision coefficients obtained by decoding the AE bitstream of length, L_i^z.

The steps involved in finding the optimal truncation points are the following:

1. Compute a set of (L_i^z, D_i^z) at all candidate truncation points for each code-block. (This step can be done during the tier-1 processing.)
2. Select a set of feasible truncation points from the candidate truncation points for each code-block.
3. Select one optimal truncation point for each code-block from the sets of feasible truncation points.

The subsequent subsections discuss the selection processes (steps 2 and 3) in detail.

Selection of Feasible Truncation Points

The selection of feasible truncation points for a code-block is performed based on rate-distortion theory. Feasible truncation points are those points that are most effective in decreasing distortion with smallest increase in bitstream length. The method is best explained using Figure 2.5. The figure shows candidate truncation lengths plotted on the x-axis against corresponding distortions on the y-axis. This is the distortion-rate curve for a particular code-block.

Each point on the plot (solid or otherwise) represents a candidate truncation point, L_i^z. There are a total of 11 candidate points shown here. Notice that out of these truncation points, some have the property to decrease distortion (for unit increase in bitstream length) more than other truncation points. These points form the convex-hull of the distortion-rate curve and are shown by the dotted line joining the solid points. These solid points (h_i^1, h_i^2, $\cdots$) form the set of feasible truncation points. The figure shows 5 feasible truncation points.

Selection of Optimal Truncation Points

At this point, for each code-block, a set of feasible truncation points and corresponding distortions have been identified. Also for each feasible truncation point we have a corresponding distortion-rate

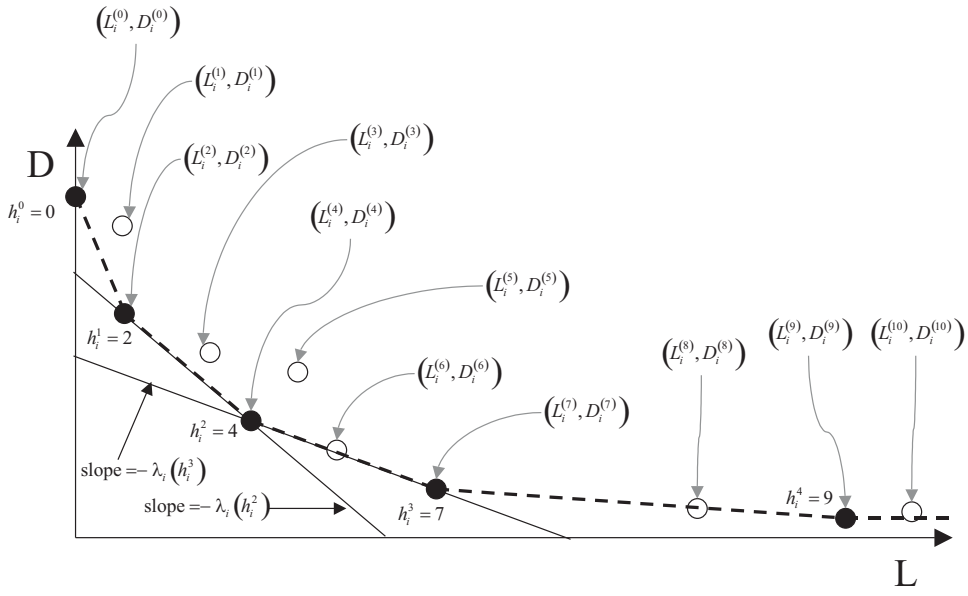


Figure 2.5: *Convex Hull of the Distortion-Rate curve.*

slope (e.g. $\lambda(h_i^1)$ in Figure 2.5). The distortion-rate slope is a measure of the amount by which the truncation point decreases distortion (from the previous feasible truncation point) for a unit increase in length of the coded bitstream. The algorithm searches for an optimal distortion-rate slope. The selection algorithm optimizes the problem by maximizing the quantity

$$J = \sum_i \lambda(h_i^{z_i}) \tag{2.1}$$

while being subject to the constraint

$$\sum_i L_i^{z_i} \leq L_d \tag{2.2}$$

where L_d is the desired total bitstream length.

The optimal truncation points of all code-blocks for a particular target length form a quality layer. If we need to provide distortion scalability, then we can have several layers of truncation points for each code-block. Each successive quality layer decreases the distortion while increasing the length of the bit-stream.

2.3.3 EBCOT vs SPIHT

The primary similarity between EBCOT and SPIHT is that both employ bitplane coding. In both cases the magnitudes of the coefficients are compared with a threshold corresponding to each bit plane. Then either a significance coding pass or a magnitude refinement pass is performed to code the information. Both methods have distortion scalability, albeit to different degrees. This means that in both cases one can reconstruct all the coefficients at a lower precision from a truncated version of the original bit-stream.

The most important difference between SPIHT and EBCOT is the fact that EBCOT divides the wavelet coefficients into relatively small code-blocks and codes each code-block independently. On the other hand, SPIHT works on all the subbands and codes the coefficients by following a tree structure. The tree structure of SPIHT is quite rigid and therefore this offers SPIHT only one degree of scalability—in distortion. Since each code-block is independent in EBCOT, it requires extra bits to code overhead information. This allows EBCOT more degrees of freedom and therefore more scalability: distortion, resolution, spatial and color component. Although EBCOT requires extra overhead bits, the optimal truncation scheme and MQ coder more than compensate.

A second important difference between SPIHT and EBCOT is the use of a separate arithmetic encoder (MQ) in EBCOT. The optimal truncation method used in EBCOT is based on the lengths of the bitstreams generated by the MQ coder. Therefore the MQ coder is an integral part of EBCOT. On the other hand, SPIHT does not have a separate arithmetic encoder. In fact SPIHT itself is an arithmetic encoder. The MQ coder is a more sophisticated arithmetic encoder refining its probability maps using information from both the coefficient and its neighborhood. The SPIHT arithmetic coder uses only the parent-child relationships between coefficients in different subbands to code information.

A third important difference between SPIHT and EBCOT is the post compression optimal truncation method adopted in EBCOT. SPIHT does not provide much control over the precision of the coefficients in different subbands. It is determined by the relative sizes of the coefficients and the path followed by SPIHT. EBCOT uses a more sensible approach. After coding each code-block independently, EBCOT determines the truncation of each code-block, in order to achieve the desired overall length. This truncation is optimal in that it achieves the lowest distortion that can be achieved for a given rate based on the underlying rate-distortion curve.

The computational costs involved in implementing EBCOT are greater than that of SPIHT. EBCOT requires more computations (adds and multiplys) in order to compute the rate distortion statistics. In addition EBCOT requires more sophisticated book-keeping during the bitplane coding in tier-1. This is a natural consequence of the fact that we are providing greater scalability with EBCOT.

2.3.4 Perceptual Weighting in EBCOT

Perceptual weights are used in most subband image compression systems to either enhance or suppress the transform coefficients in certain subbands. These perceptual weights are usually derived from measurements of human sensitivity for certain visual frequencies. EBCOT uses the perceptual subband weights in one of two ways. In the first method, the quantization step sizes of the subbands are modified based on the perceptual weights.

$$\Delta_{b,\text{weighted}} = \frac{\Delta_b}{W_b} \tag{2.3}$$

Here b represents a subband index, Δ_b is the original quantization step-size for the subband and W_b is the perceptual weight for the subband. This means that if the perceptual weight for a particular subband is large, then its quantization step-size is made small. This gives more precision to the signed integers representing the samples of that subband. This method has two main disadvantages. First, the modification of step size will not show much effect on the precisions of the subband coefficients unless the weights are restricted to be powers of two. When the weights are different powers of 2, then the precisions of the quantized subband values are changed by an integer number of bits. Secondly, when the quantization step sizes are fixed at some static value, it cannot be later modified to meet the needs of different applications in a scalable manner.

EBCOT uses the perceptual weights in a second more effective manner. As was described in the section on optimal truncation, candidate truncation lengths and corresponding distortions (MSEs) are saved during the bit-plane coding process. Instead of using raw MSEs, JPEG2000 uses weighted MSEs.

$$D^{(z)}_{i,\text{weighted}} = W_b^2 \sum_{\mathbf{j}\in\mathcal{B}_i} (\hat{y}_i[\mathbf{j}] - y_i[\mathbf{j}])^2 \tag{2.4}$$

Here the sum computes the squared error between the original coefficients and the coefficients generated from a bit-stream truncated at z. To this MSE, W_b^2 is multiplied to generate a weighted

MSE. This weighted MSE is then used in the post compression optimal truncation algorithm. The effect of this is as follows. If the perceptual weight of a subband is large, then the weighted MSE for that subband becomes large. This causes the distortion-rate slopes for that subband to be large. Thus clearly more bits have to be allocated to that subband for a given optimal distortion-rate slope. This makes more bit-planes from code-blocks belonging to this subband available in the top quality layers.

Chapter 3

Perceptual Coding Techniques

Improving the perceptual quality of compressed images has received considerable attention in the literature. Perceptual methods that exploit the varying sensitivity of the Human Visual System (HVS) to different spatial frequencies are among the most effective techniques. Recent research indicates that there have been three primary approaches to the effective design of frequency-based perceptual methods. The first method uses a contrast sensitivity function (CSF) to design perceptual weights as a function of spatial frequency [1, 10, 11]. Each weight is multiplied with all of the discrete wavelet transform (DWT) coefficients in the subband with the corresponding spatial frequency: the DWT coefficients are increased at the more sensitive frequencies and are decreased at the less sensitive frequencies. The JPEG2000 standard–Part 1 [12] includes perceptual weights derived from a CSF [1].

The second method is based on the maximum amount of quantization that can be present in a subband without being detected by humans [13]. This dictates the maximum quantization step size for each subband that will result in visually lossless compression. However, this method does not allow direct control of the bit rate. The method can be adapted to include multiplication factors on the quantization step sizes in order to achieve a prescribed bit rate; unfortunately, iteration is required to find the multiplication factors that give the desired bit rate for a particular image.

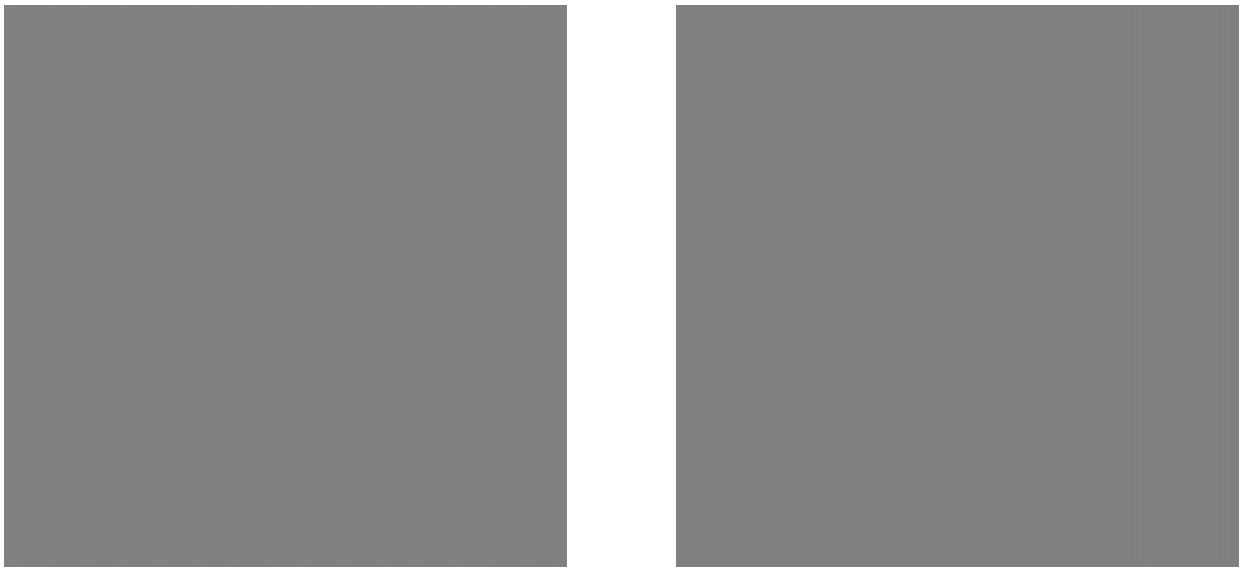
The third approach showed that the multiplication factors in the second method were sub-optimal when quantization artifacts were visible at very low bit rates [14]. This third method uses subband standard deviation (SSD) to derive a weight for each subband. Here the weights are not only frequency dependent, but also orientation dependent (distinguishing it from the previous two

methods). This approach has not been compared to weights derived from a CSF.

The choice of the transform used determines the effectiveness of the HVS based perceptual method. In general HVS tends to change properties in subbands whose ranges change by a factor of two. This is a fortunate coincidence since the subband structure into which a DWT divides the image is also dyadic in nature. Therefore HVS properties can be matched very well with wavelet subbands. This is not necessarily true for the DCT. In addition the DWT coefficients also carry spatial information which can be easily used to perform localized masking of compression artifacts. Due to these reasons perceptual coding techniques based on the HVS have more or less exclusively concentrated on DWT based compression systems.

3.1 Methods Based on CSF

The sensitivity of human vision changes over spatial frequency. As an example consider the two different spatial frequencies shown in Figure 3.1 (a) and (b). The former shows a horizontal spatial frequency of 0.03 cycles/pixel whereas the latter shows a horizontal spatial frequency of 0.2 cycles/pixel.



(a) $f = 0.03$ cycles/pixel. (b) $f = 0.2$ cycles/pixel.

Figure 3.1: *Two different spatial frequencies displaying different sensitivities.*

Both images were created with the same contrast value. This means that both sinusoids have

the same peak to peak variation in intensity. In spite of this, notice that the human eye is more sensitive to the lower frequency. The higher frequency is barely visible whereas the lower frequency can be clearly observed. Human sensitivity to contrast variations is a function of the spatial frequency. This function is called the contrast sensitivity function (CSF). One of the earliest empirical approximations of the CSF was presented by Mannos and Sakrison [10]. This work was performed only for the luminance color plane and therefore was ideally suitable for grayscale images.

Beegan et al. [15] used this CSF to derive four separate sets of perceptual weights to be used in an image compression system using the DWT and the SPIHT encoding algorithm. The perceptual weights for each wavelet subband were derived from the value(s) of the contrast sensitivity from that subband. For example one of the sets of weights was derived by averaging the contrast sensitivities over the subband. These perceptual weights were used in the encoding system by multiplying them with the DWT coefficients from the corresponding subband. Larger average contrast sensitivities over a subband resulted in larger perceptual weight for that subband. This in turn made the weighted or "masked" coefficients from that subband larger, thus allocating more bits during the encoding of that subband. Therefore the subbands with larger average contrast sensitivities could be reproduced with higher quality from the compressed image. Beegan et al. used the perceptual weights only on the luminance color plane. In the case of color images, Beegan et al. performed asymmetric compression, which means that the chrominance color planes were always compressed at 64:1 without any perceptual weights and the luminance plane were compressed at the desired compression ratio with the perceptual weights turned on.

Beegan et al. tested the new perceptual weights using a variety of grayscale and color images. Both objective (peak signal to noise ratio, PSNR) as well as subjective (using expert and non-expert human observers) measures were used to determine compressed image quality. The tests showed substantial subjective quality improvement with the use of the perceptual weights.

A new and more comprehensive measurement and modeling of human CSF was performed by Nadenau [1]. The actual contrast sensitivity was measured by asking observers to look at sinusoidal gratings (similar to the ones in Figure 3.1) and finding out the contrast at which the grating just becomes visible. The inverse of this minimum contrast is a measure of the contrast sensitivity at that frequency. This measurement was repeated for a range of frequencies and averaged over several observers to arrive at an average measured CSF. Finally a curve was fit on this measured

CSF to determine an empirical CSF function model. The three CSFs for the three color planes are shown in Figure 3.2 [1]. The luminance CSF (Y) is a bandpass filter whereas the two chrominance CSFs (Cb, Cr) are lowpass filters. This corroborates similar, though less extensive, measurements performed by Mullen [11].

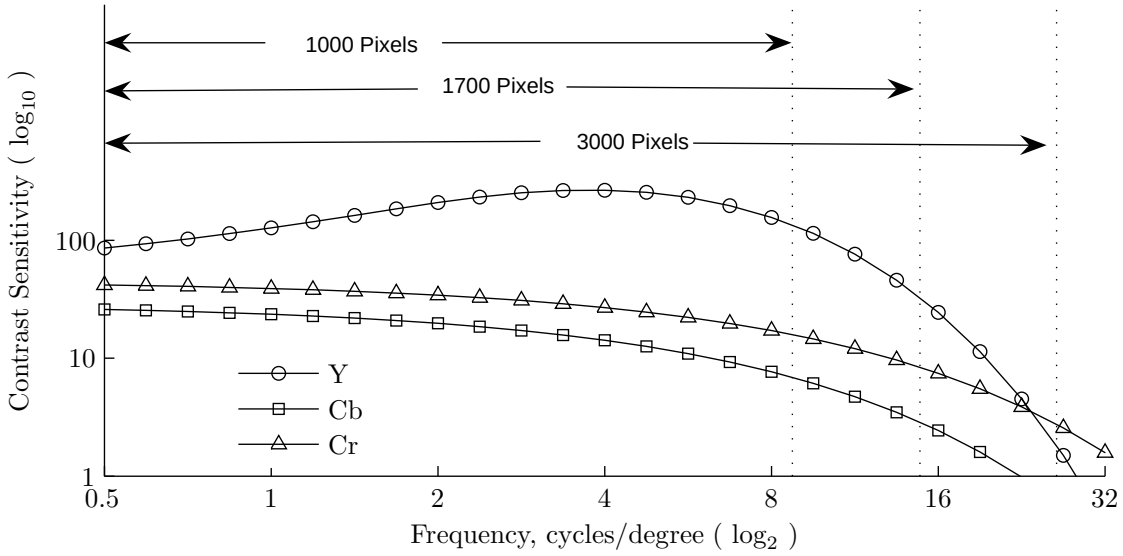


Figure 3.2: *Human contrast sensitivity function as modeled by Nadenau [1].*

In Figure 3.2 the spatial frequency has been expressed as cycles/degree instead of cycles/pixel. The spatial frequency in cycles/degree is defined as the number of cycles of the sinusoid present in one degree of viewing angle as shown in Figure 3.3. This provides a method to express the CSF independent of the viewing distance.

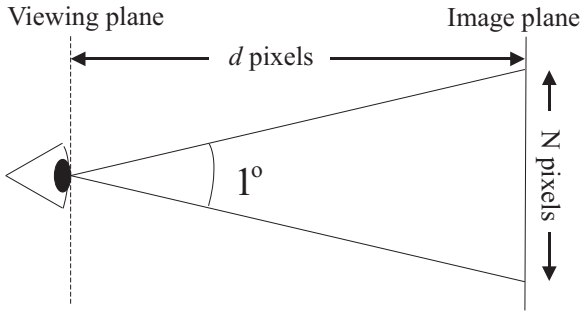


Figure 3.3: *Cycles/degree as the number of sinusoidal cycles in one degree of angular view.*

Past research indicates that the contrast sensitivity to spatial discrete frequency is, in general,

dependent on the distance from which the image is viewed. The CSF exhibits a peak at the most sensitive spatial discrete frequency. The most sensitive spatial discrete frequency shifts lower and lower as the viewing distance increases. Therefore there is the need to incorporate the viewing distance into the CSF. Expressed in cycles/degree, spatial frequency is no longer discrete, but is analog. Depending on the viewing distance, a certain lower band of the analog spatial frequency is mapped to the discrete frequency range of 0 to 0.5. For example in Figure 3.2, for a viewing distance of 1000 pixels, the lower frequency band from 0 cycles/degree to just above 8 cycles/degree is mapped to the discrete frequency range of 0 to 0.5 cycles/pixel. Similarly, for a viewing distance of 1700 pixels, the lower band from 0 cycles/degree to just under 16 cycles/degree is mapped to the discrete frequency range of 0 to 0.5 cycles/pixel. Expressing the viewing distance in terms of pixels makes this whole discussion independent of the resolution of the display.

Part 1 of the JPEG2000 standard uses perceptual weights that were derived from the CSF as modeled by Nadenau [1]. The standard provides three separate tables of perceptual weights for the three viewing distances of 1000 pixels, 1700 pixels and 3000 pixels. The standard computes the perceptual weights as an average of the CSF within the spatial frequency subband.

$$W_C(s) = \frac{\int_{f^l_{s,v}}^{f^h_{s,v}} S(f)df}{f^h_{s,v} - f^l_{s,v}} \times \frac{\int_{f^l_{s,h}}^{f^h_{s,h}} S(f)df}{f^h_{s,h} - f^l_{s,h}} \tag{3.1}$$

Here $W_C(s)$ is the computed perceptual weight for the s^{th} band, $S(f)$ is the CSF, $f^h_{s,v}$ is the higher edge vertical frequency for the s^{th} band, $f^l_{s,v}$ is the lower edge vertical frequency for the s^{th} band, $f^h_{s,h}$ is the higher edge horizontal frequency for the s^{th} band and $f^l_{s,h}$ is the lower edge horizontal frequency for the s^{th} band. The frequencies are analog spatial frequencies in *cycles/degree*. This formula computes the average value of the CSF in the horizontal and vertical directions and uses their product as the perceptual weight.

3.2 Methods Based on Suppressing Quantization Artifacts

Watson et al. [13] conducted research on the effect of DWT coefficient quantization on the reconstructed image. All visible compression artifacts are caused by the quantization of DWT coefficients. This work determined the maximum allowable amount of quantization that may be introduced in a wavelet subband before which the quantization artifacts become visible. This was done by intro-

ducing an impulse of a certain magnitude in the DWT coefficients of an image which are otherwise all zero and computing the inverse DWT. An observer was then asked whether the artifact was observable. This experiment was repeated using impulses with increasing amplitude until the observer was just able to see the artifact on an otherwise blank screen. The amplitude of the impulse stimuli was assumed to be the quantization threshold for that observer for the subband that was used. This experiment was repeated for all subbands and averaged over several observers to obtain average values of measured quantization thresholds for all the subbands. These average measured quantization thresholds were then fit to a model to empirically obtain expressions for quantization values in different subbands in a wavelet transform. This method is simple and image independent and almost always guarantees a compressed image that is visually lossless.

The disadvantage with this form of perceptual coding is the inability to control the bitrate of the compressed image. Since quantization in each subband is fixed, the size of the compressed image depends on the redundancy characteristics of the actual image being compressed.

3.3 Methods Based on Subband Standard Deviations

The work of Watson focused on obtaining visually lossless compression by keeping the quantization levels in each subband just under the threshold of visibility. This is called *sub-threshold* compression. Ramos and Hemami [14, 16] performed some psychophysical experiments to determine good perceptual coding techniques in the cases where the compression is so high that the quantization artifacts are visible (*supra-threshold* compression). Theoretically the quantization steps derived by Watson could be scaled to be used with supra-threshold compression.

However Ramos and Hemami proved that such scaled application of these quantization step sizes would be sub-optimal. In this work observers were asked to compare two images, one the original and the other an image reconstructed with a certain amount of quantization in a particular subband. The observers were asked to increase the quantization in the subband until a difference between the two images was noticed. This amount of quantization was assigned as the first minimum noticeable distortion step size (MNDSS_1) since this was the step size that gave one visual distortion unit (VDU) of difference between the reference image and the test image. Next the reference image was replaced by the test image and the quantization in the test image was increased further until

again the observer noticed a difference between the two images. This quantization was assigned as MNDSS_2 since it resulted in 2 VDUs of difference. This was continued until the subband was completely removed from the reconstruction process. The process was repeated for all subbands averaged over several observers and images. This resulted in the construction of average MNDSS sequences for each subband of each image.

Several observations were made on these MNDSS sequences. First it was found that within each subband for each image the MNDSS for higher VDUs were multiples of MNDSS_1. The implication was that results obtained for 1 VDU could be scaled to obtain results for any other VDU. Secondly it was found that MNDSS values for the LH and HL subbands were not the same. This meant that when artifacts became visible the orientation that contained more information was more sensitive to visual distortion. These observations led to the conclusion that for optimal supra-threshold compression, the quantization step sizes for each subband of the image must be proportional to the corresponding MNDSS_1.

A very good regression fit between the MNDSS_1 values and subband standard deviations was found, suggesting a functional relationship between them. Using curve-fitting tools a good fit for a function between MNDSS_1 and the subband standard deviation, σ_s was also found.

$$\text{MNDSS}_1(s) = a\sigma_s^b \tag{3.2}$$

Here $a = 31.4752$, $b = 0.3432$ and σ_s is the standard deviation of subband s. This formula suggests that the larger the standard deviation of the subband is, the larger is the step size required to produce 1 VDU of difference. This suggests that the larger the standard deviation of the subband, the less sensitive it is to quantization errors.

This expression for MNDSS_1 was used to derive optimal quantization steps for supra-threshold compression. The final observed visual distortion is due to the combined quantization effect of all the subbands. In order to derive optimal quantization coefficients to obtain an overall difference of 1 VDU, the 1 VDU needs to be distributed among all the subbands. The authors advocate distributing the 1 VDU in inverse proportion to the standard deviations of the subbands. This makes intuitive sense since more visually important information is expected to come from subbands containing larger standard deviations. Therefore, if $\alpha_s\text{VDU}(< 1)$ of distortion is allocated to

subband s, then α_s may be computed as

$$\alpha_s = \frac{K}{\sigma_s} \tag{3.3}$$

such that

$$\sum_s \frac{K}{\sigma_s} = 1 \tag{3.4}$$

Since α_sVDU of distortion from subband s is caused by a quantization step size of α_sMNDSS$_1$, the optimal quantization step size allocation may be written as

$$\begin{aligned} Q(s) &= \alpha_s \text{MNDSS}_1 \\ &= \frac{K}{\sigma_s} a \sigma_s^b \\ &= K a \sigma_s^{b-1} \end{aligned} \tag{3.5}$$

Thus in order to obtain 1 VDU of distortion over the entire reconstructed image, each subband must be quantized using the above quantization step size. Notice here that overall quantization step sizes are inversely proportional to the subband standard deviation. This makes sense since subbands with larger standard deviations are expected to contain more information. These subbands therefore need to be reproduced at higher quality. But also note that the factor b works against this principle because subbands with larger standard deviations are also less sensitive to quantization errors. Therefore this allocation addresses both opposing effects of subband standard deviation.

Quantization step size allocations can be directly converted to perceptual weights by using values proportional to the inverse of the quantization step sizes. Therefore, a subband standard deviation based perceptual weight may be derived as follows.

$$W_{SSD}(s) = l \sigma_s^{1-b} \tag{3.6}$$

where l is an arbitrary constant that makes the largest weight unity. The use of such perceptual weights helps in the direct control of bit rates for supra-threshold compression of images. The authors used these perceptual weights along with the SPIHT encoder and observed substantial improvement in perceptual quality of compressed images.

Chapter 4

New Perceptual Weights

As described in Chapter 2, perceptual weights can be used to improve the subjective quality of compressed images. The JPEG2000 standard defines three sets of suggested perceptual weights for three viewing distances. These weights were developed based on human contrast sensitivity for different frequency subbands. This chapter describes new perceptual weight designs and their effectiveness for low bitrate image compression using JPEG2000 [17].

4.1 New Perceptual Methods and Comparisons

We developed two new sets of perceptual weights: C-SSD, which combines the weights from CSF and subband standard deviation; and, C-CSD which combines the weights from CSF and code-block standard deviation. We compared the performance of our new weights to the JPEG2000 CSF weights. We also compared Hemami et al's SSD weights [14] to JPEG2000's CSF weights. We expected that one of our new methods that combines the CSF and standard deviation weights would improve the perceptual quality of the compressed images over JPEG2000's CSF weights.

JPEG2000 allows the perceptual weights to be directly applied to the mean-squared-error (MSE) obtained for each MQ-encoded code-block. These weighted MSEs, defined as:

$$WMSE(s,c) = MSE(s,c)W_i^2(s,c) \tag{4.1}$$

alter JPEG2000's subsequent optimal bit allocation; (s,c) refers to functions of subband s and/or code-block c. $WMSE(s,c)$ is the weighted MSE and $MSE(s,c)$ is the unweighted MSE for code-block c belonging to subband s. $W_i(s,c)$ refers to the subband or code-block weights of method i;

the four methods are C for CSF, SSD for subband standard deviation, C-SSD for CSF and subband standard deviation, and C-CSD for CSF and code-block standard deviation.

The first set of weights was obtained from the standard deviations of the subbands [16]:

$$W_{SSD}(s) = l\sigma_s^{(1-b)} \tag{4.2}$$

where $b = 0.3432$, σ_s is the standard deviation of subband s, and l is a normalization factor that makes the largest weight unity. These weights have been demonstrated to improve subjective performance over no weights using SPIHT for quantization. However, $W_{SSD}(s)$ had not been compared with weights derived from a CSF. We compared the performance of $W_{SSD}(s)$ with JPEG2000's CSF weights, $W_C(s)$, using the JPEG2000 EBCOT quantization algorithm. In our evaluations $W_C(s)$ represents the JPEG2000 CSF perceptual weights for a viewing distance of 1700 pixels [2].

The second set of weights was based on our new method that combines the CSF weights with subband standard deviations:

$$W_{C-SSD}(s) = l\sigma_s^{(1-b)}W_C(s). \tag{4.3}$$

We compare the performance of $W_{C-SSD}(s)$ with $W_C(s)$.

The third set of weights was based on our new method that combines the CSF weights at the subband level with the standard deviations at the code-block level:

$$W_{C-CSD}(s,c) = l\sigma_c^{(1-b)}W_C(s) \tag{4.4}$$

σ_c is the standard deviation of code-block c in subband s.

4.2 Testing Methodology

Eight images from the standard image set were used in our subjective tests [18]. Fifteen 512×512 sub-images were chosen from these 8 images in order to represent a wide range of characteristics: frequency (smooth areas–highly detailed/textured areas), color (many colors–grayscale), and content (aerial photos of buildings–people). The sub-images were classified as high, medium or low frequency by thresholding the average number of edges detected by a Sobel operator.

The 15 images were compressed with JPEG2000's *Kakadu* coder Version 2.2 [2] using one of the four sets of perceptual weights: $W_C(s)$, $W_{SSD}(s)$, $W_{C-SSD}(s)$ and $W_{C-CSD}(s,c)$. The

images were compressed using 5 levels in the JPEG2000 lossy coder (biorthogonal-9/7 wavelet); 5 compression ratios were examined: 32:1, 50:1, 70:1, 90:1, and 120:1. This resulted in 75 (15 images $\times$ 5 compression ratios) test images for each perceptual weighting method. Again, the three comparisons performed in our subjective tests were: $W_C(s)$ vs. $W_{SSD}(s)$, $W_C(s)$ vs. $W_{C-SSD}(s)$, and $W_C(s)$ vs. $W_{C-CSD}(s, c)$.

We derived our testing methodology from the stimulus comparison testing procedure outlined in the standard for subjective assessment of the quality of television pictures [19]. During a subjective test session, the participant was asked to compare the perceptual quality of two candidate images. Both candidates were the same image compressed at the same compression ratio; however, each candidate was compressed using one of the two perceptual weighting methods under comparison: $W_i(s)$ or $W_C(s)$. Throughout each test session the image/compression ratio combination was randomly chosen from the set of 75 candidate images.

The two images were shown one at a time; the participant could switch between them as often as desired, but a gray, blank screen was displayed for 2 seconds during the switch. The participant was asked to fill the blank: "Image 1 is ___ compared to Image 2". The choices were (1) worse, (2) slightly worse, (3) same, (4) slightly better, or (5) better. (Note: the assignment of $W_i(s)$ and $W_C(s)$ to Image 1 and Image 2 was random throughout each test session).

When $W_i(s)$ was used in Image 1, choices (1)-(5) translated to scores of –2, –1, 0, +1, +2. However, when $W_C(s)$ was used in Image 1, choices (1)-(5) translated to scores of +2, +1, 0, –1, –2. This scoring method generates positive values if the participant decided that $W_i(s)$ performed better than $W_C(s)$; conversely, negative values indicate that the participant decided that $W_C(s)$ performed better than $W_i(s)$.

Each participant viewed 25 comparisons; the first 5 decisions were thrown away to account for the learning process. 45 participants took our test—15 participants for each perceptual weight comparison. Thus, each of the 3 perceptual weighting comparisons was evaluated using 300 decisions spread randomly across 75 image/compression ratio combinations.

4.3 Results

4.3.1 Comparison 1: $W_C(s)$ vs. $W_{SSD}(s)$

Figure 4.1 shows that the subjective quality with the $W_{SSD}(s)$ perceptual weights was lower than with the $W_C(s)$ perceptual weights. For all image types (high, medium and low frequency) at all compression ratios, the average test scores are always below zero—indicating that compression using $W_{SSD}(s)$ was perceived as worse, on average, than compression using $W_C(s)$.

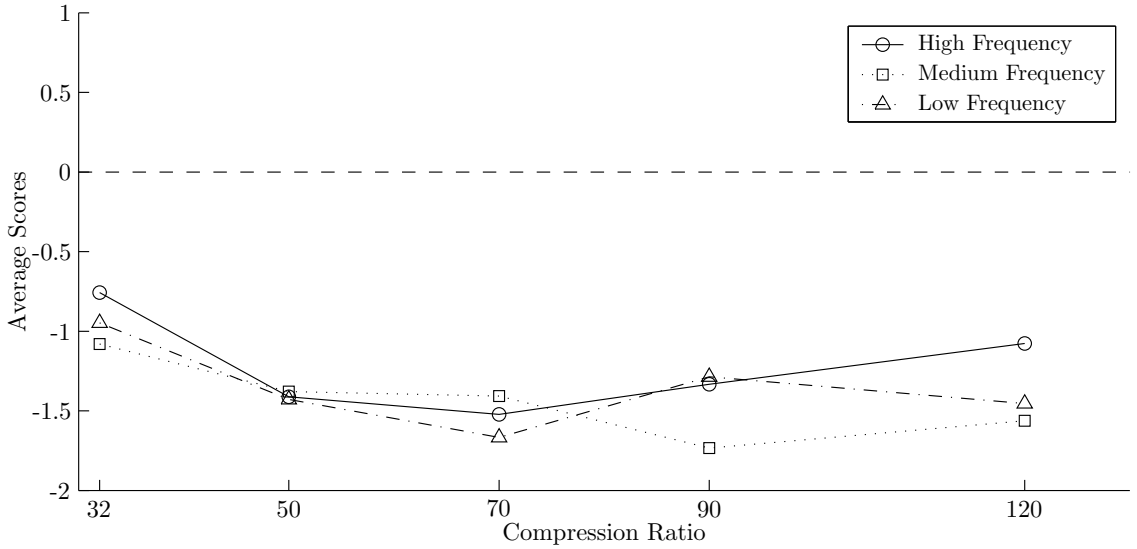


Figure 4.1: *Average test scores for the $W_C(s)$ vs. $W_{SSD}(s)$ comparison. $W_C(s)$ outperformed $W_{SSD}(s)$ for all image types at all compression ratios.*

4.3.2 Comparison 2: $W_C(s)$ vs. $W_{C-SSD}(s)$

Figure 4.2 depicts varying subjective quality results for the different types of images when the $W_{C-SSD}(s)$ perceptual weights are compared with the $W_C(s)$ perceptual weights. For high frequency images, the average test score was positive (except at 120:1 where images compressed with any method are significantly degraded); this indicates that compression using $W_{C-SSD}(s)$ was perceived, on average, as better than compression using $W_C(s)$. For low frequency images, the average test score was negative which means that compression using $W_{C-SSD}(s)$ was consistently perceived as worse than compression using $W_C(s)$.

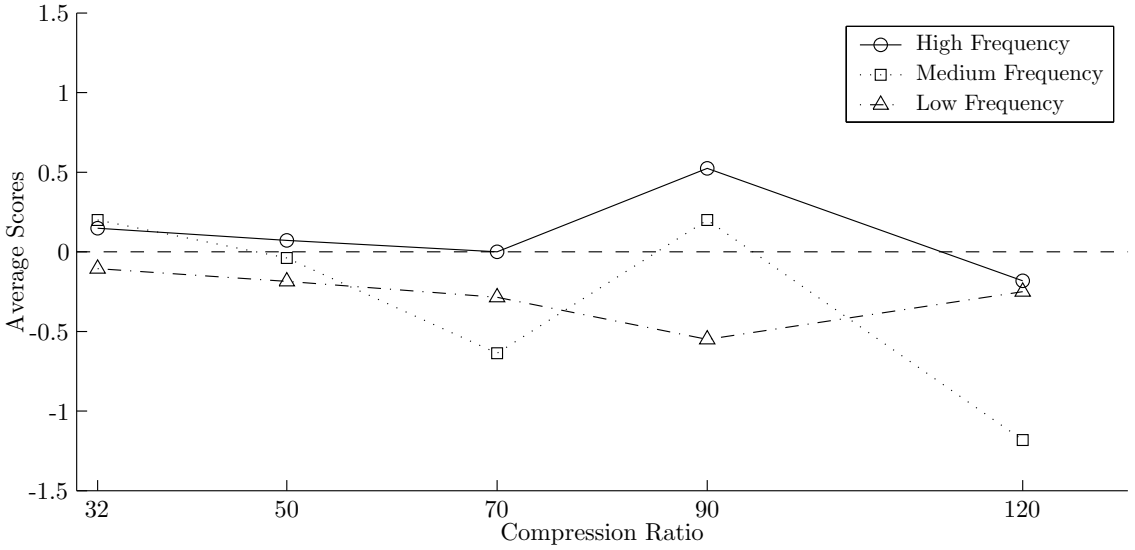


Figure 4.2: *Average test scores for the $W_C(s)$ vs. $W_{C-SSD}(s)$ comparison. $W_{C-SSD}(s)$ improved subjective quality for the high frequency images, but $W_C(s)$ provides better quality for low frequency images.*

Figure 4.2 exhibits mixed results for medium frequency images. One test sub-image was taken from the newspaper/silverware section of Bike. The subjective test scores for each participant are depicted in Figure 4.3. The participants' evaluations were positive and negative for this image; thus, there was a lack of consensus on perceived quality.

In general, for medium frequency images at low to moderate compression ratios (32:1 and 50:1), $W_{C-SSD}(s)$ improved the subjective quality over $W_C(s)$. $W_{C-SSD}(s)$ is better able to preserve the high frequencies (edges, texture) due to the SSD part of the weight that gives more emphasis to the higher frequencies than the CSF part of the weight.

At higher compression ratios ($\geq$ 70:1), $W_{C-SSD}(s)$ continues to preserve edges better than $W_C(s)$. Figure 4.4 (a) illustrates how $W_C(s)$ did not reconstruct the edges of the silverware as sharply as $W_{C-SSD}(s)$ in Figure 4.4 (b). However, $W_{C-SSD}(s)$ tends to introduce shadow artifacts at the edges of smooth areas juxtaposed against detailed areas. For example, compare the smooth top of the newspaper in Figure 4.5 (a) compressed using $W_C(s)$ with the same area in Figure 4.5 (b) compressed using $W_{C-SSD}(s)$. Figure 4.5 (a) is perceptually better.

A participant who focused more on the silverware than the newspaper would have contributed

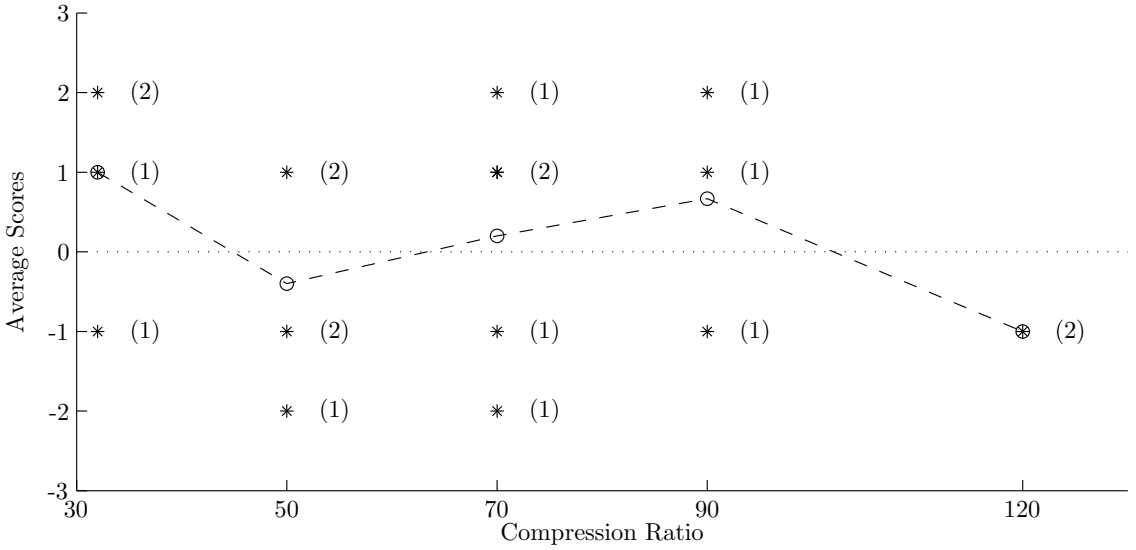


Figure 4.3: *Individual (*) and average test scores (dashed line) for the $W_C(s)$ vs. $W_{C-SSD}(s)$ comparison using the newspaper/silverware sub-image from Bike. There was a lack of consensus on perceived quality.*

to the positive scores at 70:1 in Figure 4.3. On the other hand, a participant who focused more on the newspaper than the silverware would have contributed to the negative scores at 70:1 in Figure 4.3.

4.3.3 Comparison 3: $W_C(s)$ vs. $W_{C-CSD}(s)$

The subjective quality with the $W_{C-CSD}(s)$ perceptual weights was lower than with the $W_C(s)$ perceptual weights for low and medium frequency images. This differs slightly from the $W_{C-SSD}(s)$ performance which showed improved perceptual quality for medium frequency images at 32:1 and 50:1. Also unlike the $W_{C-SSD}(s)$ results, $W_{C-CSD}(s)$ generated similar perceptual quality as $W_C(s)$ for high frequency images. Although the average test score for all images at all compression ratios was negative for $W_{C-CSD}(s)$—indicating that compression using $W_{C-CSD}(s)$ was consistently perceived as worse than compression using $W_C(s)$—it did not perform as poorly as $W_{SSD}(s)$.

(a) $W_C(s)$. (b) $W_{C-SSD}(s)$.

Figure 4.4: *The silverware edges compressed at 70:1 using (a) $W_C(s)$ and (b) $W_{C-SSD}(s)$. The edges in (a) were not as sharp as those in (b).*

(a) $W_C(s)$. (b) $W_{C-SSD}(s)$.

Figure 4.5: *The newspaper compressed at 70:1 using (a) $W_C(s)$ and (b) $W_{C-SSD}(s)$. The smooth areas had a shadow artifact in (b) which was not present in (a).*

4.3.4 Additional Metrics: PSNR and WVDP

Although the focus of our study was on subjective quality evaluated by test participants, we also examined peak signal-to-noise ratio (PSNR); the results are depicted in Figure 4.6. Interestingly, the PSNR values were higher with $W_{C-SSD}(s)$ than with $W_C(s)$ for the high frequency images

(and lower for the low frequency images). For medium frequency images, PSNR was higher with $W_{C-SSD}(s)$ at 32:1 and 50:1, but lower for 70:1 and higher. Our perceptual evaluations were in line with PSNR—an uncommon occurrence.

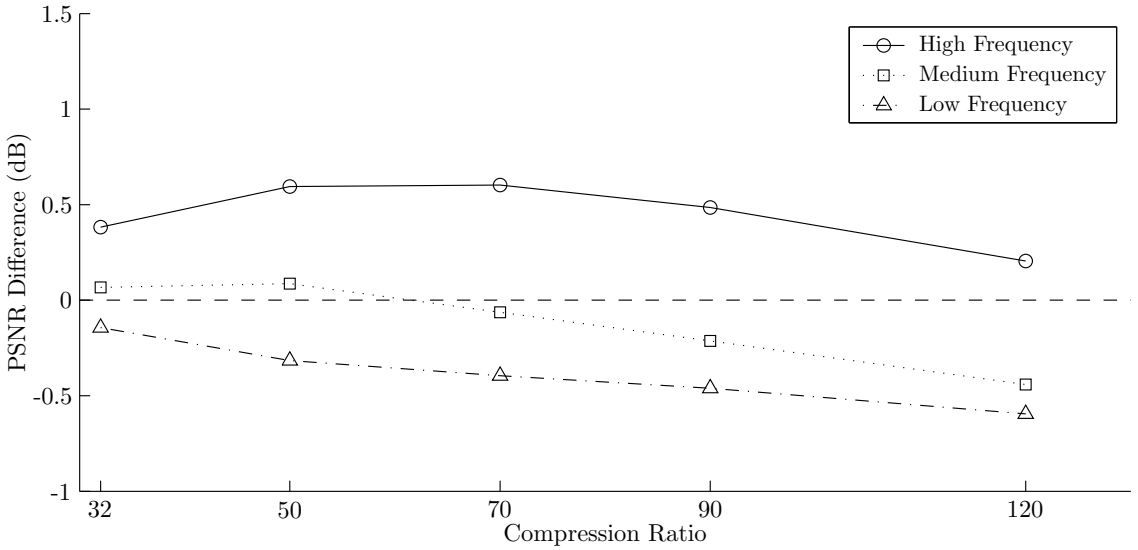


Figure 4.6: *Average difference in PSNR between $W_{C-SSD}(s)$ and $W_C(s)$; positive values represent a higher PSNR for $W_{C-SSD}(s)$.*

We also calculated the wavelet visible difference predictor (WVDP) metric [20]. This performance measure is supposed to be able to capture subjective quality by computing a single number. Our results indicated that the WVDP for $W_{C-SSD}(s)$ was worse than the WVDP for $W_C(s)$ for all images at all compression ratios. This does not correlate with the results of our subjective tests.

Chapter 5

Challenges in Hardware Implementation of JPEG2000

The main steps involved in the JPEG2000 encoding algorithm are (1) color transform, (2) wavelet transform, (3) EBCOT tier-1, (4) EBCOT tier-2, and (5) bitstream packing. EBCOT tier-1 converts the wavelet coefficients into an arithmetic coded bitstream using context adaptive bitplane coding. As described in Section 2.3.1 EBCOT tier-1 can be thought of, conceptually, as having a context formation (CF) task and an arithmetic encoder (AE) task. The CF task traverses the code-block in stripe order and generates symbols to code along with coding contexts. The AE task accepts the symbols and contexts and generates an arithmetically encoded bitstream. The particular instance of arithmetic encoder used in EBCOT is called the MQ coder. The CF task consists of three coding passes for each bitplane: significance propagation (pass 0), magnitude refinement (pass 1), and cleanup (pass 2). The MQ coder, which is the particular instance of the AE task, is a binary multiplier less arithmetic coder that uses several coding contexts. A finite state machine is used to compute adaptive probabilities of each of the coding contexts.

An understanding of the steps involved in EBCOT tier-1 is necessary to identify the hardware implementation challenges posed by the algorithm. This chapter first discusses the details of the CF task followed by a detailed discussion of the AE task. This is followed by a review of the literature so far on accelerating hardware implementations of the algorithm. The chapter closes with a discussion of some of the issues that have not been looked at in detail in JPEG2000 hardware research which provides a motivation for the new architecture presented in Chapter 6.

5.1 The Context Formation Task

After the scaled and shifted image pixels are passed through wavelet transform module, the DWT coefficients need to be converted into the sign-magnitude representation before the CF task can start. The sign bits and the magnitude bits are then stored and accessed separately during the coding process. In the discussion that follows the sign bits are represented as $\chi[\mathbf{j}]$ and the magnitudes are represented as $v[\mathbf{j}]$. Here the subscript $\mathbf{j}$ is a two-element vector (j_1, j_2) that represents the row and column of the coefficient in the code-block. The total number of rows in the code-block is represented as J_1 and the total number of columns is represented as J_2. All traversal through the coefficients within a code-block follows the strict stripe traversal order as laid down in Section 2.3.1.

5.1.1 Binary State Variables

The CF task maintains three sets of binary state variables. These variables are used to determine the coding pass membership of each bit encountered during the stripe traversal of the code-block. As will be presented later, each bit encountered in the stripe traversal is coded using any one of three coding passes – *significance propagation* (pass 0), *magnitude refinement* (pass 1), or *cleanup* (pass 2). Each coefficient in the code-block has associated with it, the three binary variables. The identity of the pass that actually encodes the bit of a particular coefficient is determined by the values of the three binary state variables for that coefficient. The values of these state variables are updated as the coding progresses. More information on the three state variables is presented below.

- *Significance State*, $\sigma[\mathbf{j}]$: At the begining of the encoding process for a code-block, the value of $\sigma[\mathbf{j}]$ is initialized to 0 for all coefficients indicating that they are all initially insignificant. As the coding progresses from the top bitplane to the bottom, when the magnitude bit of a coefficient becomes 1 for the first time, it means that the coefficient has become significant at that bitplane. $\sigma[\mathbf{j}]$ for that coefficient is then set to 1 and does not change after that.

- *Delayed Significance State*, $\overleftarrow{\sigma}\,[\mathbf{j}]$: $\overleftarrow{\sigma}\,[\mathbf{j}]$ is a delayed version of $\sigma[\mathbf{j}]$. $\overleftarrow{\sigma}\,[\mathbf{j}]$ is initialized to 0 for all coefficients and is set to 1 only after the first magnitude refinement pass for the coefficient at $\mathbf{j}$. The magnitude coding context (to be introduced later) for the first magnitude refinement

pass is determined using the delayed significance state.

- *Pass Membership State*, $\pi[\mathbf{j}]$: This state variable determines whether the coefficient at $\mathbf{j}$ has already been coded by pass 0 or not. It is only if the coefficient has not been coded by pass 0, that it needs to be considered for coding by either pass1 or pass 2. If the coefficient at $\mathbf{j}$ is coded by pass 0, then $\pi[\mathbf{j}]$ is set to 1, otherwise it is set to 0. Passes 1 and 2 only consider the coefficient for coding if $\pi[\mathbf{j}]$ is 0.

5.1.2 The Main Encoding Routine

The pseudo code for the main encoding routine is given in Algorithm 1. The algorithm uses the signs and magnitudes of the code-block coefficients as input and generates the encoded bytes. The first two lines initialize the variables used by the AE task (to be described later). Lines 3 to 7 perform two things: (1) initialize the three sets of binary state variables for all the code-block coefficients and (2) find out the highest non-zero bitplane for the code-block. At the end of the *for* loop, the register v^{tmp} contains 1 in all bits up to the highest non-zero bitplane. This register is used in lines 8 to 11 to set the starting bitplane to the highest non-zero bitplane. Finally lines 12 to 18 perform the actual encoding. The three coding passes are performed sequentially for each bitplane starting from the highest to the lowest. Note that only the cleanup pass is performed for the highest bitplane.

5.1.3 Significance Propagation Pass

The significance propagation pass (pass 0) is the first pass performed for each bitplane (except the highest one). This routine uses signs, magnitudes and significance states as input. The routine generates symbol-context pairs (to be consumed by the AE task) and also updates the significance states and the pass-membership states. The pass traverses all the bits in the current bitplane in the stripe order shown in Figure 2.4 and for each bit, performs the following processing. The line numbers here refer to Algorithm 2. Line 3 computes the neighborhood significance context, $k^{\text{sig}}[\mathbf{j}]$, for that bit. $k^{\text{sig}}[\mathbf{j}]$ is one of 9 values between 0 and 8. A non-zero value of $k^{\text{sig}}[\mathbf{j}]$ means that at least one of the neighbors in the context window is significant. Lines 4 to 11 shows the processing that needs to be done if the coefficient at $\mathbf{j}$ has been insignificant till now and if it has significant

Algorithm 1: *Main Bitplane Coding Algorithm.*

In : Signs: $\chi[\mathbf{j}]$, Magnitudes: $v[\mathbf{j}]$
Out: Encoded bytes

Initialize the MQ coder ;
Initialize the Context State tables to be used by the MQ coder ;
$v^{\text{tmp}} \leftarrow 0$ (*A register wide enough for the largest coefficient.*);
for each $\mathbf{j} \in [0, \mathrm{J}_1) \times [0 \times \mathrm{J}_2)$ **do**
    (*For each coefficient belonging to the code-block.*) ;
    Initialize $\sigma[\mathbf{j}] \leftarrow 0$, $\overleftarrow{\sigma}[\mathbf{j}] \leftarrow 0$ and $\pi[\mathbf{j}] \leftarrow 0$;
    $v^{\text{tmp}} \leftarrow v^{\text{tmp}} \vee v[\mathbf{j}]$ ($\vee$ *is bitwise inclusive OR.*);
$K \leftarrow K_b^{\max}$;
while $(K > 0)$ AND $\left(2^{K-1} > v^{\text{tmp}}\right)$ **do**
    (*Flush out all the upper bitplanes that are all zeros.*) ;
    $K \leftarrow K - 1$;
for $p = K-1, \ldots, 1, 0$ **do**
    (*For all bitplanes that are not all zero.*) ;
    if $p < K - 1$ **then**
        (*Not for highest bitplane.*) ;
        Perform **Encoder-Pass0(**χ**,** $\mathbf{v}$**,** σ**)** ;
        Perform **Encoder-Pass1(**$\mathbf{v}$**,** σ**,** $\overleftarrow{\sigma}$**,** π**)** ;
    Perform **Encoder-Pass2(**χ**,** $\mathbf{v}$**,** σ**,** π**)** ;

neighbors. The current bit of the coefficient and the neighborhood context are stored as the symbol-context pair. Also the pass-membership state for the coefficient is set to 1 indicating that it has been coded in this pass. If the current bit is 1 (the coefficient has just become significant), it's sign is encoded and the significance state for the coefficient is updated to 1.

Algorithm 2: *Encoder-Pass0 (Significance Propagation).*

In : Signs: χ, Magnitudes: $\mathbf{v}$, Significance States: σ
Out: Encoded bytes, Significance States: σ, Pass Membership States: π

for each $\mathbf{j}$ **do**
    (*Following the stripe based traversal.*);
    Compute $k^{\text{sig}}[\mathbf{j}]$ (*Neighborhood significance context.*);
    if $\sigma[\mathbf{j}] = 0$ AND $k^{\text{sig}}[\mathbf{j}] > 0$ **then**
        (*If previously insignificant and if any neighbor is significant.*);
        Store: Symbol $= v^p[\mathbf{j}]$, Context $= k^{\text{sig}}[\mathbf{j}]$;
        if $v^p[\mathbf{j}] = 1$ **then**
            (*If significant.*);
            $\sigma[\mathbf{j}] \leftarrow 1$;
            Perform **Encode-Sign**(χ, σ) ;
        $\pi[\mathbf{j}] \leftarrow 1$;
    else
        (*If previously significant.*);
        $\pi[\mathbf{j}] \leftarrow 0$;

On the other hand, if the current coefficient is already significant (from a previous bitplane) or if it has insignificant neighbors, then it is not coded in this pass and the pass-membership state is

set to 0 indicating that one of the following two passes needs to pick up this coefficient.

Algorithm 3 shows the pseudo code for the sign encoding routine. The routine uses the signs and significance states as input and generates symbol-context pairs. First, in lines 1 and 2, it computes horizontal and vertical sign bias functions (χ^{h} and χ^{v}). These values indicate the predominant signs of the horizontal and vertical neighborhoods. These values can take on any integer value between –2 and +2. Next, in lines 3 and 4, the ranges of the sign bias functions are restricted to be between –1 and +1 by truncation. Next in line 5, the neighborhood sign context (k^{sign} and χ^{flip}) is computed. Finally if the product of $\chi[\mathbf{j}]$ and χ^{flip} is positive, 0 and k^{sign} are stored as the symbol-context pair. If the product is negative 1 and k^{sign} are stored as the symbol-context pair.

Algorithm 3: *Encode-Sign.*

In : Signs: χ, Significance States: σ
Out: Encoded bytes

1 Compute horizontal sign bias function : $\chi^{\mathrm{h}} \triangleq \chi[j_1, j_2-1]\sigma[j_1, j_2-1] + \chi[j_1, j_2+1]\sigma[j_1, j_2+1]$ ($\mathbf{j} = (j_1, j_2)$) ;
2 Compute vertical sign bias function : $\chi^{\mathrm{v}} \triangleq \chi[j_1-1, j_2]\sigma[j_1-1, j_2] + \chi[j_1+1, j_2]\sigma[j_1+1, j_2]$;
3 Compute truncated horizontal sign bias function : $\chi^{-\mathrm{h}} \triangleq \mathrm{sign}\left(\chi^{\mathrm{h}}[\mathbf{j}]\right) \min\left\{1, \left|\chi^{\mathrm{h}}[\mathbf{j}]\right|\right\}$;
4 Compute truncated vertical sign bias function : $\chi^{-\mathrm{v}} \triangleq \mathrm{sign}\left(\chi^{\mathrm{v}}[\mathbf{j}]\right) \min\left\{1, |\chi^{\mathrm{v}}[\mathbf{j}]|\right\}$;
5 Compute k^{sign} and χ^{flip} from $\chi^{-\mathrm{h}}$ and $\chi^{-\mathrm{v}}$. (*Neighborhood sign context.*) ;
6 **if** $\chi[\mathbf{j}].\chi^{\mathrm{flip}} = 1$ **then**
7 **Store:** Symbol = 0, Context = k^{sign} ;
8 **else**
9 $(\chi[\mathbf{j}].\chi^{\mathrm{flip}} = -1)$;
10 **Store:** Symbol = 1, Context = k^{sign} ;

Neighborhood Significance Context

The neighborhood significance is one of 9 values between 0 and 8. It indicates a coding context for the AE task to use. The value of $k^{\mathrm{sig}}[\mathbf{j}]$ is computed using the significance states of the 8 neighboring coefficients of $\mathbf{j}$. This set of neighbors is called a context window and is shown by the red square in Figure 2.4. First three intermediate quantities are computed as shown below:

$$k^{\mathrm{h}}[\mathbf{j}] = \sigma[j_1, j_2-1] + \sigma[j_1, j_2+1] \tag{5.1}$$

$$k^{\mathrm{v}}[\mathbf{j}] = \sigma[j_1-1, j_2] + \sigma[j_1+1, j_2] \tag{5.2}$$

$$k^{\mathrm{d}}[\mathbf{j}] = \sum_{k_1=\pm 1} \sum_{k_2=\pm 1} \sigma[j_1+k_1, j_2+k_2] \tag{5.3}$$

As is evident from the equations, these intermediate quantities give a measure of the significance of the horizontal, vertical and diagonal neighborhoods of $\mathbf{j}$. Finally the actual value of $k^{\mathrm{sig}}[\mathbf{j}]$ can

be obtained from a mapping of these intermediate values to the context using Table 5.1.

Table 5.1: *Look-up table for computing neighborhood significance contexts.*

$k^{\mathrm{sig}}[\mathbf{j}]$	LL and LH blocks			HL blocks			HH blocks	
	$k^{\mathrm{h}}[\mathbf{j}]$	$k^{\mathrm{v}}[\mathbf{j}]$	$k^{\mathrm{d}}[\mathbf{j}]$	$k^{\mathrm{h}}[\mathbf{j}]$	$k^{\mathrm{v}}[\mathbf{j}]$	$k^{\mathrm{d}}[\mathbf{j}]$	$k^{\mathrm{d}}[\mathbf{j}]$	$k^{\mathrm{h}}[\mathbf{j}]+k^{\mathrm{v}}[\mathbf{j}]$
8	2	x*	x	x	2	x	≥ 3	x
7	1	≥ 1	x	≥ 1	1	x	2	≥ 1
6	1	0	≥ 1	0	1	≥ 1	2	0
5	1	0	0	0	1	0	1	≥ 2
4	0	2	x	2	0	x	1	1
3	0	1	x	1	0	x	1	0
2	0	0	≥ 2	0	0	≥ 2	0	≥ 2
1	0	0	1	0	0	1	0	1
0	0	0	0	0	0	0	0	0

* Don't care

Neighborhood Sign Context

The sign context is one of five values between 10 and 14 that indicates five different neighborhood sign conditions. The look-up table shown in Table 5.2 is used to compute the neighborhood sign context given the truncated horizontal and vertical sign bias functions.

Table 5.2: *Look-up table for computing neighborhood sign contexts.*

$\chi^{-\mathrm{h}}[\mathbf{j}]$	$\chi^{-\mathrm{v}}[\mathbf{j}]$	k^{sign}	χ^{flip}
1	1	14	1
1	0	13	1
1	-1	12	1
0	1	11	1
0	0	10	1
0	-1	11	-1
-1	1	12	-1
-1	0	13	-1
-1	-1	14	-1

5.1.4 Magnitude Refinement Pass

The pseudo code for the magnitude refinement pass (pass 1) is given in Algorithm 4. This pass uses information from the coefficient magnitudes, significance states, delayed significance states

and pass-membership states. The algorithm generates symbol-context pairs and also updates the delayed significance states.

Algorithm 4: *Encoder-Pass1.*

In : Magnitudes: $\mathbf{v}$, Significance States: σ, Delayed Significance States: $\overleftarrow{\sigma}$, Pass Membership States: π
Out: Encoded bytes, Delayed Significance States: $\overleftarrow{\sigma}$

```
1 for each j do
2     (Following the stripe based traversal.) ;
3     if σ[j] = 1 AND π[j] = 0 then
4         (If already significant from a previous bitplane.) ;
5         Compute k^mag from σ←[j] and k^sig[j] (Magnitude context.) ;
6         Store: Symbol = v^p[j], Context = k^mag ;
7         σ←[j] ← σ[j] ;
```

For each coefficient in the code-block this pass performs the following processing. First, the coefficient is considered for coding only if it is significant ($\sigma[\mathbf{j}] = 1$) and if it was not encoded in the significance propagation pass ($\pi[\mathbf{j}] = 0$). These two conditions ensure that magnitude refinement takes place only if the coefficient became significant in a higher bitplane than the current one. If this condition is satisfied, it next generates the magnitude refinement coding context using the delayed significance states and the neighborhood significance context as shown in Table 5.3.

Table 5.3: *Look-up table for computing magnitude refinement contexts.*

$\overleftarrow{\sigma}[\mathbf{j}]$	$k^{\mathrm{sig}}[\mathbf{j}]$	k^{mag}
0	0	15
0	> 0	16
1	x	17

Then the current coefficient bit and the magnitude context are stored as the symbol-context pair. Finally the delayed significance context is updated with the significance context. The use of the delayed significance state warrants some discussion. First time magnitude refinement has been assigned two special coding contexts in JPEG2000. During the first magnitude refinement (at the bitplane below the one where it became significant), the delayed significance state is still 0 and so the context is either 15 or 16 depending on whether the coefficient has a significant neighbor or not. All succeeding magnitude refinements for that coefficient have the delayed significance state at 1 and so the context generated in always 17.

5.1.5 Cleanup Pass

The cleanup pass (Pass 2) is the final pass for every bitplane except the topmost one. As was presented above, the significance propagation pass picks up those coefficients that had been insignificant till then, but which had significant neighbors. The magnitude refinement pass picks up those coefficients that already became significant in a previous bitplane. The cleanup pass then picks up all the coefficients that are left over from the previous two passes, i.e. those coefficients that are insignificant and also have insignificant neighbors. It performs this in two steps – (1) run length encoding followed by (2) explicit significance coding of coefficients that caused a run interruption (if at all). Algorithm 5 shows the pseudo code of the cleanup pass. It consists of mainly two conditional statements. The first *if* statement (lines 3 to 19) performs the run length encoding and the second *if* statement (lines 20 to 29) performs the significance coding for the run interruption.

The run length encoding is performed only if all four coefficients in a stripe column have all insignificant neighbors. If this is not the case each of the four coefficients are individually handled by the second *if* statement. If all four have all insignificant neighbors, the algorithm computes the number of coefficients (from the top of the column) that are themselves insignificant in the current bitplane (lines 9 to 11). If all 4 are themselves insignificant in the current bitplane ($r = 4$), then the algorithm enters the full run mode and stores 0 and $k^{\text{run}}(= 9)$ as the symbol-context pair. This results in the most efficient encoding since 4 bits have been encoded at one shot. None of these coefficients will now be processed by the second *if* statement. If not all 4 of the coefficients are themselves insignificant in the current bitplane, the algorithm enters the run-interruption mode. The run-length, r, now contains the number of coefficients from the top of the stripe column that are themselves insignificant. First 1 and k^{run} are stored as the symbol-context pair indicating a run interruption. Then two more symbols, the MSB and LSB of the run-length, r are stored along with the run interruption context, $k^{\text{uni}}(= 18)$.

The second *if* statement (lines 20 to 29) is the part that performs explicit significance coding of those coefficients that have all insignificant neighbors and which come after the breaking of the run-length sequence. The condition in the *if* statement ensures that only those coefficients that have been insignificant till now and which were not coded by the significance propagation pass are considered for coding here. Lines 22 to 23 ensure that those coefficients that came before the

break of the run-length sequence are not encoded here. The significance of the run-interrupting coefficient ($r = 0$) is encoded by the three symbol-context pairs stored during the run interruption. Therefore this coefficient only needs a sign coding. All coefficients below it in the stripe column undergo explicit significance and sign coding.

Algorithm 5: *Encoder-Pass2.*

In : Signs: χ, Magnitudes: $\mathbf{v}$, Significance States: σ, Pass Membership States: π
Out: Encoded bytes, Significance States: σ

for each **j** **do**
  (*Following the stripe based traversal.*) ;
  if (j_1%4 = 0) AND ($j_1 \leq J_1 - 4$) **then**
    (*Entering a full stripe column.*) ;
    $r \leftarrow -1$ (*Signifies not using run mode.*) ;
    if $k^{\text{sig}}[j_1 + i, j_2] = 0$ for all $i \in \{0, 1, 2, 3\}$ **then**
      (*Enter run mode since all coefficients in the column have all insignificant neighborhoods.*) ;
      $r \leftarrow 0$ (*Signifies using run mode.*) ;
      while ($r < 4$) AND ($v^p[j_1 + r, j_2] = 0$) **do**
        (*While coefficient itself is insignificant.*);
        $r \leftarrow r + 1$ (*Count the run length.*) ;
      if $r = 4$ **then**
        (*Full run mode.*) ;
        Store: Symbol = 0, Context = k^{run} ;
      else
        (*Run interruption.*) ;
        Store: Symbol = 1, Context = k^{run} ;
        Store: Symbol = $\left\lfloor \frac{r}{2} \right\rfloor$, Context = k^{uni} (*Encode MSB of run length*);
        Store: Symbol = r%2, Context = k^{uni} (*Encode LSB of run length*);
  if $\sigma[\mathbf{j}] = 0$ AND $\pi[\mathbf{j}] = 0$ **then**
    (*If insignificant and not coded in Pass0.*) ;
    if $r \geq 0$ **then**
      $r \leftarrow r - 1$ (*No need to code significance; already performed by run mode encoding.*) ;
    else
      Store: Symbol = $v^p[\mathbf{j}]$, Context = $k^{\text{sig}}[\mathbf{j}]$;
    if $v^p[\mathbf{j}] = 1$ **then**
      (*Significant; need to encode sign.*);
      $\sigma[\mathbf{j}] \leftarrow 1$;
      Perform **Encode-Sign**(χ, σ) ;

5.2 The Arithmetic Encoder Task

In Section 5.1 we saw how the CF task processes the quantized wavelet coefficients in bitplanes using three coding passes and generates symbol-context pairs. These symbol-context pairs are converted to an encoded bitstream by the AE task. The AE task is a multiplierless version of the arithmetic encoder that uses a finite state machine to perform the more probable symbol (MPS)

estimation as well and the less probable symbol (LPS) probability estimation. A high-level block diagram of the AE task is shown in Figure 5.1.

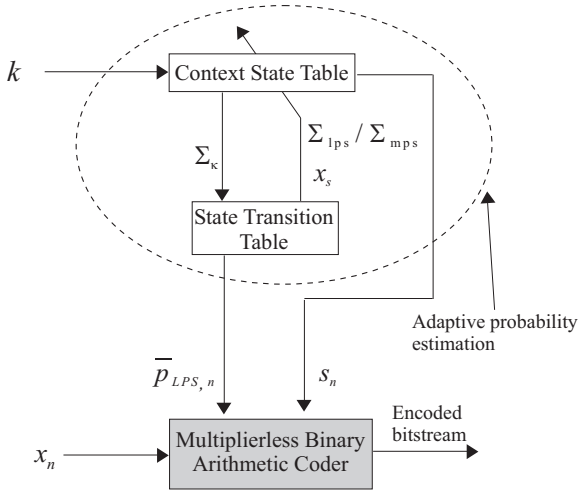


Figure 5.1: *Block Diagram of the AE task.*

The AE task accepts as input a sequence of symbols to be encoded ('0' or '1') denoted as x_n as well as a corresponding sequence of contexts, κ. The context can be one of 19 different ones denoted by numbers between 0 and 18. These contexts designate the type of symbol that is being coded. For example the context could be one of 9 significance contexts, one of 5 sign contexts, one of 3 magnitude refinement contexts or one of 2 run-mode contexts.

The AE task maintains the state, Σ_κ of each context κ in a Context State Table. The state of the context that was passed is used to look up the MPS symbol and LPS probability corresponding to that state from the state-transition table which is then sent to the multiplier less arithmetic coder. Also depending on whether the encoded symbol is an MPS or LPS, the corresponding next state is also looked up from the state transition table and is used to update the state of the context. Detailed descriptions of these blocks are now given below.

5.2.1 Context State Table

The context state table consists of 3 columns that are read from and written to by the AE task. The table with its initial values is shown in Table 5.4. The first column of the table is the context that is passed into the AE task and serves as an index that is used to access the table. The

Table 5.4: *Context State Table with initial values.*

Context, κ	**Type**	**State, Σ_κ**	**MPS, s_κ**
0	Significance	4	0
1		0	0
2		0	0
3		0	0
4		0	0
5		0	0
6		0	0
7		0	0
8		0	0
9	Run-mode	3	0
10	Sign	0	0
11		0	0
12		0	0
13		0	0
14		0	0
15	Magnitude	0	0
16		0	0
17		0	0
18	Run Interruption	46	0

second column of the table is merely for information that identifies the type of context. The third column of the table stores the state of the context. This column needs to store one of 47 states and therefore needs 6 bits to store each context. The fourth column stores the MPS corresponding to each context. The value is either 0 or 1 and therefore it needs 1 bit for each context.

In summary, the context state table needs to store and update two values for each of the 19 contexts. The total storage space needed is a total of 7 bits for each of the contexts.

5.2.2 State Transition Table

The state transition table used in the AE task is shown in Table 5.5. This is a fixed table and values only need to be read from this table. The table contains 5 entries for each of the 47 states that the contexts could be in. The first column of the table serves as a state-index used to read off the other entries and therefore this column need not be explicitly stored. Columns 2 through 4 are used for state transition. The state of the current context in the context state table is updated

with Σ_{lps} if the currently coded symbol is the LPS. If the currently coded symbol is the MPS it is updated with Σ_{mps}. Also under certain conditions the MPS of the current context in the context state file is flipped depending on whether X_s is 1 or 0. Finally column 5 is an estimate of the LPS symbol probability. This value is used to perform updates on the interval length and lower bound values of the arithmetic encoder.

Table 5.5: *State transition table for the AE task.*

State, Σ	Transition			Probability $\bar{p}_{\text{LPS}}$
	Σ_{mps}	Σ_{lps}	X_s	
0	1	1	1	0x5601
1	2	6	0	0x3401
2	3	9	0	0x1801
3	4	12	0	0x0AC1
4	5	29	0	0x0521
5	38	33	0	0x0221
6	7	6	1	0x5601
7	8	14	0	0x5401
8	9	14	0	0x4801
9	10	14	0	0x3801
10	11	17	0	0x3001
11	12	18	0	0x2401
12	13	20	0	0x1C01
13	29	21	0	0x1601
14	15	14	1	0x5601
15	16	14	0	0x5401
16	17	15	0	0x5101
17	18	16	0	0x4801
18	19	17	0	0x3801
19	20	18	0	0x3401
20	21	19	0	0x3001
21	22	19	0	0x2801
22	23	20	0	0x2401
23	24	21	0	0x2201
24	25	22	0	0x1C01
25	26	23	0	0x1801
26	27	24	0	0x1601
27	28	25	0	0x1401
28	29	26	0	0x1201
29	30	27	0	0x1101
30	31	28	0	0x0AC1
31	32	29	0	0x09C1
32	33	30	0	0x08A1
33	34	31	0	0x0521
34	35	32	0	0x0441
35	36	33	0	0x02A1
36	37	34	0	0x0221
37	38	35	0	0x0141
38	39	36	0	0x0111
39	40	37	0	0x0085
40	41	38	0	0x0049
41	42	39	0	0x0025
42	43	40	0	0x0015
43	44	41	0	0x0009
44	45	42	0	0x0005
45	45	43	0	0x0001
46	46	46	0	0x5601

5.2.3 Internal State Registers

The context state file and the state transition table are maintained to perform adaptive probability estimation. Apart from these states, the AE task stores a set of internal state registers that are used to perform the actual arithmetic coding. This section describes the structure of these registers. The AE task stores 5 internal registers named A, C, $\bar{T}$, $\bar{t}$ and L. The register A stores the interval length and the register C stores the lower bound. $\bar{T}$ is a temporary byte buffer and $\bar{t}$ maintains the exponent of A and C in an indirect manner. It is a down-counter which identifies the time at which partially generated code bits should be shifted out of the coder. Finally L is a register that stores the number of encoded bytes that have been shifted out. This register is used to maintain the lengths of the bitstreams after each coding pass. Figure 5.2 shows the structure of the two main registers, A and C. Register A is 16 bits in width and its MSB is always maintained to be

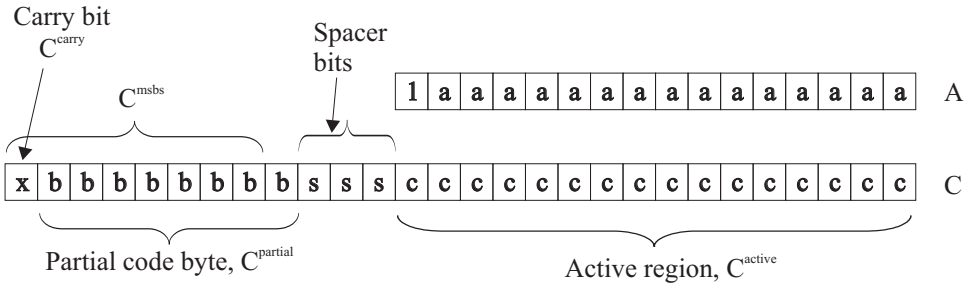


Figure 5.2: *Structure of the main internal registers of the AE task.*

'1' since it represents the mantissa of the interval length. The active region of register C is also 16 bits in width and is known as C^{active}. This part of register C holds the mantissa of the lower bound. The exponent of A and C are not explicitly maintained, but are maintained on a modulo-8 basis. This is because the MQ coder is a byte-based coder and coded bytes need to be transferred out of the coder only when the partial code-byte, C^{partial} is full, which happens once every 8 re-normalizations. Register $\bar{t}$ maintains a count of the number of bits that have been shifted out of the left end of C^{active} and thus indirectly acts as the exponent. Register $\bar{T}$ is a temporary byte buffer that stores the value from C^{partial} before it is sent out of the coder. The detailed pseudo-codes for the encoding algorithm are given in the next section.

5.2.4 Pseudocodes

The context state table is initialized with the values shown in Table 5.4. The initialization of the internal state registers is shown in Algorithm 6 and the main AE encoding algorithm is shown in Algorithm 7.

Algorithm 6: *MQ Coder initialization.*

In : —
Out: —

$A \leftarrow$ `0x8000` (*represents initial interval length 1*);
$C \leftarrow 0$ (*represents initial lower bound 0*);
$\bar{t} \leftarrow 12$ (*initially $\bar{t}$ is not 8 because the three spacer bits have to be filled up*);
$\bar{T} \leftarrow 0$;
$L \leftarrow -1$ (*avoids transferring $\bar{T}$ to the outside before it is filled for the first time*) ;

Algorithm 7: *Main MQ encoding algorithm.*

In : Symbol: x, Context: κ
Out: Encoded Bytes: $\bar{T}$

Use the context κ to obtain s_κ and Σ_κ from the context state table;
Use the state Σ_κ to read off the entries $\Sigma_{\mathrm{mps}}(\Sigma_\kappa)$, $\Sigma_{\mathrm{lps}}(\Sigma_\kappa)$, $X_s(\Sigma_\kappa)$ and $\bar{p}_{\mathrm{LPS}}(\Sigma_\kappa)$ from the state transition table;
$s \leftarrow s_\kappa$, $\bar{p} \leftarrow \bar{p}_{\mathrm{LPS}}(\Sigma_\kappa)$;
$A \leftarrow A - \bar{p}$;
if $A < \bar{p}$ **then**
    (*conditional exchange of MPS and LPS*);
    $s \leftarrow 1 - s$;
if $x = s$ **then**
    (*assign MPS the upper sub-interval*);
    $C \leftarrow C + \bar{p}$;
else
    (*assign LPS the lower sub-interval*);
    $A \leftarrow \bar{p}$;
if $A < 2^{15}$ **then**
    if $x = s_\kappa$ **then**
        (*the symbol was a real MPS*);
        $\Sigma_\kappa \leftarrow \Sigma_{\mathrm{mps}}(\Sigma_\kappa)$;
    else
        (*the symbol was a real LPS*);
        $s_\kappa \leftarrow s_\kappa \oplus X_s(\Sigma_\kappa)$;
        $\Sigma_\kappa \leftarrow \Sigma_{\mathrm{lps}}(\Sigma_\kappa)$;
(*update the context state table*);
while $A < 2^{15}$ **do**
    (*perform renormalization shift*) ;
    $A \leftarrow 2A$;
    $C \leftarrow 2C$;
    $\bar{t} \leftarrow \bar{t} - 1$;
    if $\bar{t} = 0$ **then**
        TransferByte($\bar{T}$, C, L, $\bar{t}$) ;

Since the mantissa is of size 16 bits, the initial value of interval-length, 1, should have been represented with 2^{16} and the exponent down-counter register, $\bar{t}$ should have been initialized with

8 + 3 spacer bits = 11. But representing 2^{16} needs 17 bits and the value of A is less than unity always after the initialization. This is a waste of a bit. Therefore we initialize A with 2^{15} and compensate for this by forcing the exponent to downcount one extra time at the beginning. This is the reason why $\bar{t}$ is initialized with 12 and not 11. All other initializations should be self-evident.

In the main encoding routine, the identity of the more probable symbol (MPS) and the state of the coding context that was passed are first obtained from the context state table. In lines 8–13, the actual update of the interval's lower bound and interval length happens. These are the core arithmetic encoding steps. In lines 14–21 the state of the current coding context is updated in the context state table using the values read off from the state transition table. Finally lines 23–29 perform the renormalization operation that is central to the implementation of finite precision arithmetic encoders. During the renormalization operation bytes may be transmitted out of the encoder depending on whether enough number of shifts have been made to force a byte out.

The pseudo code for the **TransferByte** routine is given in Algorithm 8. The **TransferByte** routine basically emits the byte in the temporary byte buffer, transfers a byte from the more significant side of C to the temporary byte buffer and finally resets those bits transferred from C to zero. In the process it performs some special processing for the cases where there is a carry from C. Special processing also needs to be performed for the cases where the byte buffer contains 0xFF.

5.3 Review of Hardware Implementation Research

This section presents the most important contributions to implementing EBCOT tier-1 that is available in literature. The general consensus among all researchers is that EBCOT tier-1 is the most computationally intensive module. Software profiling experiments have revealed that on the average approximately 60% of the processing time for JPEG2000 is spent on the tier-1 module [7,8]. There is also consensus that the conventional serial software architecture presented in the JPEG2000 book by Taubman et al. [2] is inefficient and that better architectures can be designed in hardware.

Andra et al. proposed a system level architecture for all three tasks involved in JPEG2000 encoding—DWT, CF, and AE [21]. Most of the subsequent work on high speed EBCOT architectures focused solely on improving the clock cycle efficiency of the CF task. Lian et al. [7] did some pioneering work on improving the efficiency of a hardware implementation of EBCOT tier-1.

Algorithm 8: *The **TransferByte** routine.*

```
   In  : —
   Out: Encoded Bytes: T̄
 1 if T̄ = 0xFF then
 2     (cannot propagate any carry past T̄; need bit stuffing) ;
 3     if L ≥ 0 then
 4         EmitByte(T̄) ;
 5     L ← L + 1 ;
 6     T̄ ← C^msbs ;
 7     C^msbs ← 0 ;
 8     t̄ ← 7 (transfer 7 bits plus carry) ;
 9 else
10     T̄ ← T̄ + C^carry (propagate any carry bit from C into T̄) ;
11     C^carry ← 0 (reset carry bit) ;
12     if L ≥ 0 then
13         EmitByte(T̄) ;
14     L ← L + 1 ;
15     if T̄ = 0xFF then
16         (decoder will see this as bit-stuffing; need to act accordingly) ;
17         T̄ ← C^msbs ;
18         C^msbs ← 0 ;
19         t̄ ← 7 (transfer 7 bits plus carry) ;
20     else
21         T̄ ← C^partial ;
22         C^partial ← 0 ;
23         t̄ ← 8 (transfer full byte) ;
```

This research performed a fairly comprehensive analysis of EBCOT tier-1. At the outset they claimed that the reason why EBCOT is so computationally intensive was because it performed bit-level processing. Also the fact that each bitplane had to be traversed thrice (once for each coding pass) highly complicated a hardware implementation. They designed an accelerator module that performed these operations more efficiently. They were also able to realize this architecture in a prototyping chip.

They broadly classified EBCOT tier-1 into two modules – (1) context formation module, CF and (2) arithmetic encoder module, AE. The CF scans through the coefficient bits and generates contexts. The AE uses these contexts to compute adaptive probabilities using a finite state automaton. Lian et al. tried to optimize the CF module. Their analysis revealed that generating the contexts of the 4 bits of a column simultaneously was not different from generating them sequentially. By doing this, two methods of speedup were obtained.

The first method was called *Sample Skipping* (SS). Here during a pass, the relevant context for the 4 bits in each column were computed simultaneously, and only those bits that actually needed to be coded in that pass are coded, one after the other. The second method was called

Group-Of-Column Skipping (GOCS). There were several cases when none of the 4 bits in a column had to be coded in pass 0. Thus, if a group of columns (say N columns) all had no bits that needed to be coded in pass 0, then they could all be directly skipped in pass 1, and directly run-length encoded in Pass 2. This also provided substantial savings in clock cycles.

The proposed architecture was modeled on Verilog to generate a prototype chip which could process a 64×64 code-block. It was found that with only GOCS in place the processing time (measured in clock cycles) of the CF module was reduced by 22% from the conventional serial architecture. If only SS is in place, the processing time reduces by 55% from the conventional architecture. With both SS and GOCS in place, they demonstrated that the processing time could be reduced by 60% from the conventional architecture.

Gupta et al. corrected a minor error in Lian's architecture to properly account for significance propagation within a stripe column [22]. However, their primary contribution optimized the CF module further by introducing logic that computed symbol-context pairs for an entire stripe column in one clock cycle. This architecture achieved nearly 50% savings in clock cycles compared to Lian's SS; with the corresponding higher clock rate, this translated to a speed-up of 2.6. Lian and Gupta's work achieved throughput improvements by speeding up the CF task; both maintained JPEG2000's original coding efficiency.

The JPEG2000 standard includes three mode switches (RESET, RESTART, and CAUSAL) that, when turned on, allow the three coding passes to be executed in parallel. These mode switches may be turned on or off in any combination. In the default mode, ([7, 21, 22]), all of the mode switches are off. The following pass-parallel architectures turn on these mode switches to achieve parallelism at the expense of coding efficiency.

Chiang et al. [23] provided new architectural suggestions that improved the speed of the EBCOT tier-1 engine. They suggested the use of two separate context-window logic units (CWLU). A context window is the set of pixels that form the immediate neighbors of the column being processed. A CWLU computes the contexts based on the values of the state variables of the column under consideration. The first CWLU performs either pass 0 or pass 1 depending on the significance of the individual pixels. Again the SS method of Lian et al. is used to efficiently code only those pixels that have to be coded. The second CWLU lags behind the first CWLU by one column and performs pass 2 only on those pixels that have not been coded by the first CWLU. Thus all three

passes are run in parallel using some degree of pipelining to ensure the causality between passes.

Note that in the original algorithm, each coding pass traversed through all the stripes before moving on to the next pass. Thus in this architecture the RESET, RESTART and CAUSAL mode-switches need to be turned on to preserve the causality between the coding passes. More details about the meanings of these switches and their effects are presented in the next section. Note that because of these mode-switches independent encoded bitstreams are being generated in parallel for each coding pass. Another problem is the need to deal with three separate AE modules. To eliminate this problem, the authors suggest a pass switching arithmetic encoder (PSAE). The PSAE coder has one hardware unit that performs the core arithmetic functions, but has three sets of stored states, one of which is used depending on which of the passes is being encoded. The processing speed of the proposed architecture was compared with that of the SS method suggested by Lian et al. for three images. The results show that processing time (in clock cycles) of the CF module reduced by approximately 25% on the average.

Gangadhar et al. [24] performed a detailed analysis of the context generation module of EBCOT tier-1 and proposed an improved architecture that was pass-parallel. The original serial architecture required the bits in each bitplane to be traversed 3 times, once for each coding pass. This is a waste of clock cycles since each bit is encoded in only one of the three passes. This paper rectified this problem by running the passes in parallel, similar to the work done by Chiang et al. Two separate Processing Elements (PE) were introduced, with PE1 performing Pass 0 and PE2 performing either Pass 1 or Pass 2. PE1 and PE2 run simultaneously (thus giving rise to pass-parallelism) and the causality between the passes was maintained by turning on the RESET, RESTART and CAUSAL mode-switches and by making PE2 process the bits that are 2 columns behind PE1. Within PE1, the SS method introduced in Section 1 was used to efficiently spend clock cycles for coding only those bits that need to be coded. From the updated (and causal) state variables available to PE2 it could determine which coding pass the rest of the bits belong to and perform the coding as required.

This architecture was modeled using VHDL and synthesized targeting a Xilinx Virtex II system. The PEs were designed as Finite State Machines (FSM). When implemented on XC2V1000, the package performed at 56 MHz. This implementation was also designed for a single 64×64 sized code-block. During actual implementation they demonstrated that this new architecture reduced

the processing time (in clock cycles) of the CF module by 75% when compared to the sequential architecture and by more than 34% when compared to the Chiang's architecture [23].

Chiang et al. improved on their earlier pass-parallel architecture and proposed a parallel context modeling architecture with a matching pipelined arithmetic encoder [25]. At a clock speed of 185 MHz, this architecture reduced processing time by 25% compared to their earlier work. Fang et al. proposed a method that concurrently encodes multiple bitplanes [26]. They computed theoretical clock cycle savings compared to Lian's and Chiang's architectures. Xing et al. proposed a sample-parallel architecture that reduced the clock cycle count by 50% compared to Gangadhar and by 60% compared to Chaing [27].

Pastuszak also achieved throughput improvement, but in a different way than the other approaches. Instead of focusing on the CF task, Pastuszak improved the throughput of the AE task by designing a hardware AE module that encodes two symbol context pairs in each clock cycle [28]. The new pipelined AE module ran on faster clocks than previous designs. The combined effect was a 75% improvement in processing speed for the AE module; however, the implication for system level throughput improvement was not presented. This approach, like Lian's and Gupta's, maintains JPEG2000's original coding efficiency.

5.4 Mode Switches for Parallelism

The JPEG2000 standard allows for certain mode switches that can be turned on, which allows the coding passes to be executed in parallel and still generate a valid JPEG2000 bitstream. The values of the mode switches can be read by the decoder from the JPEG2000 header, which allows the decoder to perform the decoding in a mode identical to the encoder. Here we discuss these switches and the effect that they have on the nature of the bitstream generated. The standard allows for any of these mode switches to be turned on in any combination. In the default mode presented in the pseudo codes, none of the mode switches given here have been turned on. The same is true in the case of the research performed by Lian et. at. We call this default mode the *serial* version of EBCOT tier-1.

- **RESTART** : This mode switch forces the encoder and decoder to re-initialize the MQ coder of the AE task at the start of each coding pass. This results in each coding pass having an

independent encoded bitstream. Thus if we have 5 bitplanes, with RESTART turned on, we get $1 + (3 \times 4) = 13$ independent encoded bitstreams. Although this mode disrupts the probability estimation process of the AE task, which results in some loss of coding efficiency, the RESTART mode switch is necessary for any pass parallel architecture.

- **RESET** : This mode switch forces the encoder and decoder to re-initialize the context state table at the start of each coding pass. In other words, by turning on the RESET mode switch we no longer have context-state adaptation causality. This results in the estimated symbol probabilities being less accurately modelled and therefore a further loss in coding efficiency. With just this mode switch turned on we get no particular advantage unless some kind of encoder or decoder design constraint forces one to reset the context states for each coding pass. But when used along with RESTART, this provides a powerful tool to implement simple pass parallel architectures. With both RESTART and RESET turned on, the causal relationship between the coding passes within a bitplane no longer exists and therefore they can be implemented in parallel.

- **CAUSAL** : This mode switch forces the encoder and decoder to consider all coefficients from future stripes (the ones below) to be insignificant. This allows us to place the parallel coding pass processors closer to each other. For example, if the CAUSAL switch is turned off, then in order to start a particular coding pass for a stripe, we have to wait till the previous pass has finished the current stripe and then has processed the first two columns of the succeeding stripe before the pass can start. This is because the context window that is used to compute coding contexts contains the coefficients that are present below the current one. With the CAUSAL switch turned on, the context window is limited to the bottom edge of the stripe and therefore it does not matter whether the coefficients below have been processed by the preceding pass or not.

5.5 Effect of Mode Switches

When the RESTART mode switch is turned on the coding efficiency of the AE task is affected because the internal registers are reset at the beginning of each coding pass. What should have been one long continuous and efficiently encoded bitstream is now being split into several indepen-

dent ones. When the RESET mode switch is turned on, the probability adaptation engine is reset at the beginning of each coding pass. This again results in inefficient arithmetic coding. Where, earlier, each coding context had used highly adapted probabilities, now they use partially adapted probabilities. The effect of turning on the CAUSAL mode switch is that this may result in the computation of improper coding contexts. The finite state machine that is the probability adaptation engine was designed with the nature of the coding contexts in mind. If the contexts that are computed are not the 'real' contexts (according to design), the wrong context may be adapted, thus resulting in some loss of efficiency in the arithmetic coding. Thus the effect of turning on any of these mode switches is, in general, a loss of coding efficiency. This means that in order to achieve the same amount of distortion in the compressed file as before, the compressed file size would have to be larger. Conversely in order to achieve a given compressed file size, the compressed image would contain more distortion.

Figures 5.3 to 5.5 show the average reduction in PSNR from the conventional serial mode due to the turning on of each of the individual mode switches. This average was computed over 23 images from the standard image set [18] for six different compression ratios. The figures also show the range between the maximum and minimum reduction in PSNR from the serial mode.

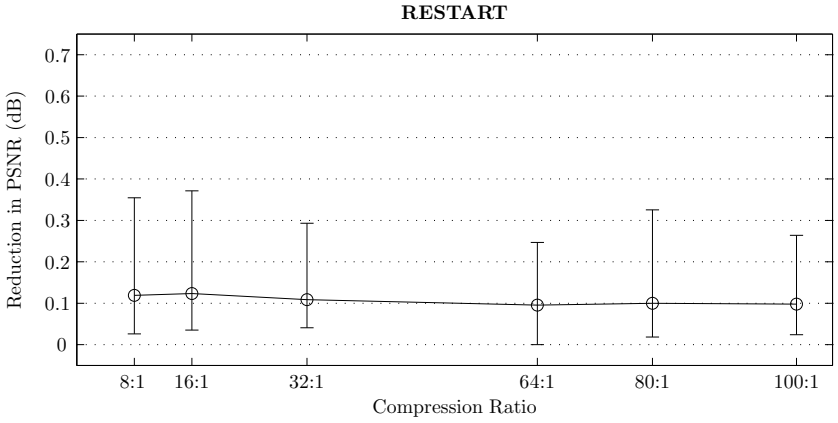


Figure 5.3: *Reduction in PSNR due to RESTART mode switch.*

Notice that the RESTART mode causes the maximum degradation in performance with approximately 0.1 dB reduction in PSNR across all compression ratios and a maximum of 0.37 dB reduction. This is followed by the RESET mode that causes approximately 0.05 dB reduction in PSNR with a maximum of close to 0.25 dB reduction. Finally the CAUSAL mode results in least

degradation with an average of approximately 0.02 dB reduction in PSNR and a maximum of just over 0.1 dB reduction. Figure 5.6 shows the reduction in PSNR when all three mode switches are turned on. Under this condition the average PSNR reduction is between 0.1 to 0.2 dB with a maximum reduction of 0.68 dB. Most of the research that has been done to implement pass parallel architectures have turned on all three mode switches. It would not be possible to exploit parallelism between passes without turning on the RESTART mode switch. But observe that coding efficiency can be preserved to a certain extent if only the RESTART mode switch is turned on. In the best case scenario this gives an improvement of 0.3 dB in PSNR improvement over the case where all three switches are turned on. Therefore there is a need to explore the possibility of improved throughput architectures that also preserve some of EBCOT's coding efficiency.

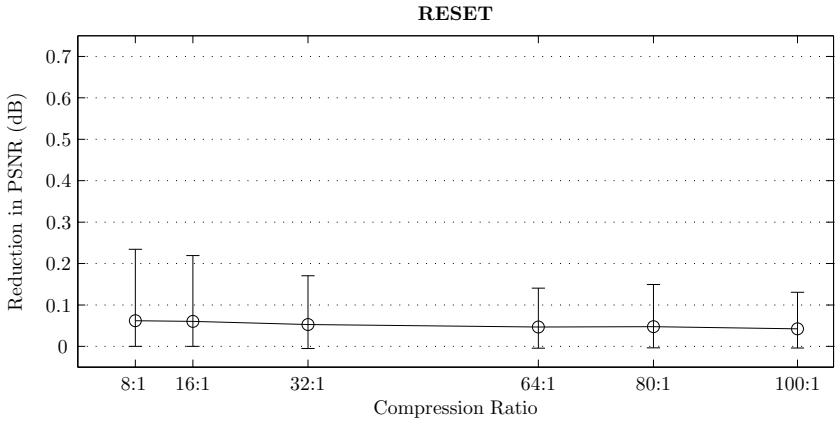


Figure 5.4: *Reduction in PSNR due to RESET mode switch.*

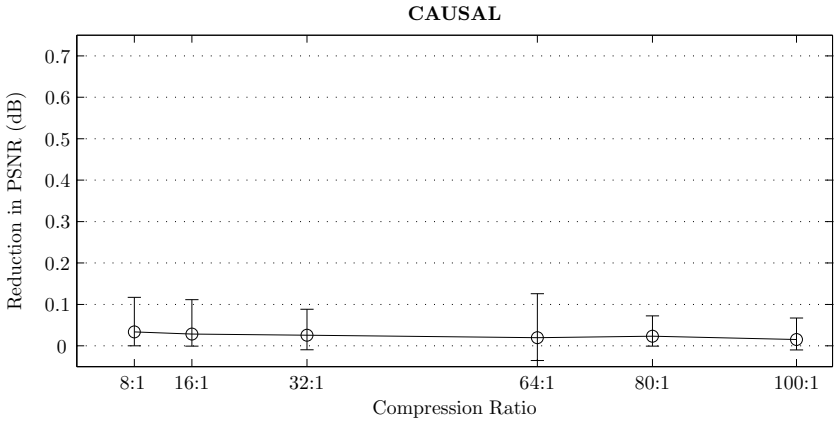


Figure 5.5: *Reduction in PSNR due to CAUSAL mode switch.*

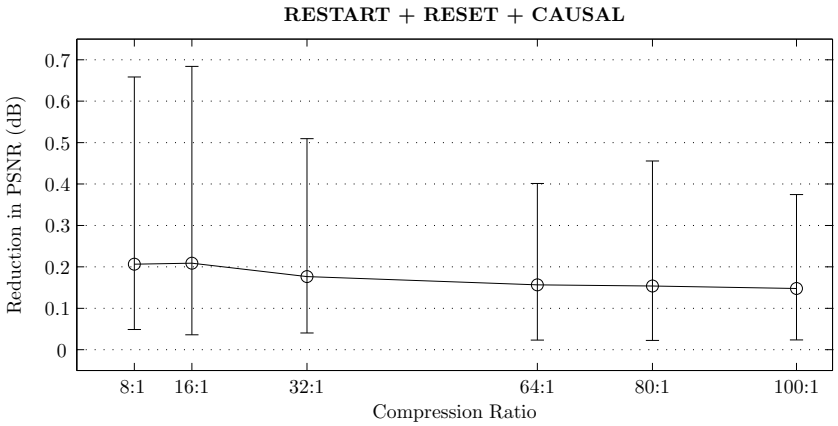


Figure 5.6: *Reduction in PSNR due to RESTART+RESET+CAUSAL mode switch.*

Chapter 6

New EBCOT Tier-1 Process Analysis

The conventional, serial process model for EBCOT tier-1 is shown in Figure 6.1. Assume that the code-block coefficient magnitudes are represented using $K_{\mathrm{b}}^{\max}$ bits. The bitplanes are numbered from $K_{\mathrm{b}}^{\max} - 1$ down to 0. In each of the bitplanes the coding passes are performed sequentially starting with pass 0 (significance propagation) followed by pass 1 (magnitude refinement) followed by pass 2 (cleanup). This is true for all bitplanes, except the topmost ($K_{\mathrm{b}}^{\max} - 1$), for which only pass 2 is performed. The serial process results in a single continuous encoded bitstream for the whole code-block. This bitstream may be truncated at the end of any of the passes by EBCOT tier-2 using optimal truncation by taking into account rate-distortion statistics from all code-blocks.

Each of the passes has been divided into the two EBCOT tier-1 tasks: (1) context formation (CF), and (2) arithmetic encoding (AE). In Figure 6.1 these have been subscripted using the pass number. The three passes for each bitplane are grouped using an overbrace. The double arrows represent causal dependency between the tasks and the single arrows represent data flow. Consider the tasks for bitplane $K_{\mathrm{b}}^{\max} - 2$. Task CF_0 represents the CF processing for significance propagation for bitplane $K_{\mathrm{b}}^{\max} - 2$ and task AE_0 represents the corresponding AE processing. Similarly the pair (CF_1, AE_1) represents CF and AE processing respectively for magnitude refinement for bitplane $K_{\mathrm{b}}^{\max} - 2$. Finally the pair ($\mathrm{CF}_2$, AE_2) represents CF and AE processing respectively for cleanup for bitplane $K_{\mathrm{b}}^{\max} - 2$.

Symbol-context pairs are generated by each CF task and sent to the corresponding AE task. This is represented by the single arrows going from each CF task to the corresponding AE task. In bitplane $K_{\mathrm{b}}^{\max} - 2$, CF_1 cannot start until CF_0 has finished. This is because magnitude re-

finement requires the most updated significance states in order to determine the coefficients whose magnitudes need to be refined. Similarly CF_2 cannot start until CF_1 has finished. These causal dependencies are represented by the double arrows going from CF_0 to CF_1 and from CF_1 to CF_2. Similarly AE_1 for bitplane $K_b^{max} - 2$ cannot start before AE_0 finishes. This is because AE_1 needs access to the final context state table and MQ registers at the end of AE_0 before it can start. The same is true for the causal dependency of AE_2 on AE_1. These causal dependencies are represented by the double arrows going from each AE task to the next.

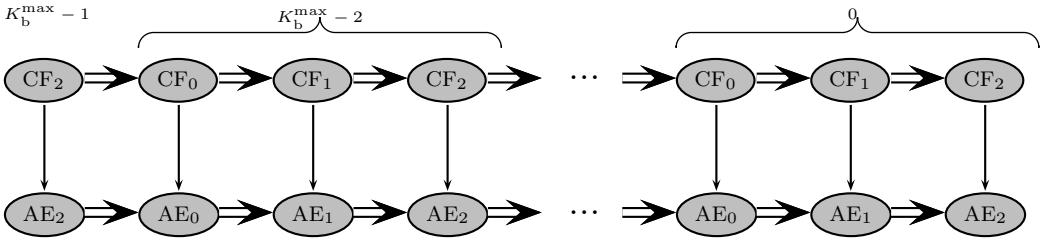


Figure 6.1: *The conventional serial EBCOT tier-1 process.*

In this conventional serial process, the algorithm does not allow for much parallelism because of the causal relationship of the context-state table and the internal MQ registers from one coding pass to the next. This results in slow implementations in both hardware and software.

6.1 The Split Arithmetic Encoder Process for EBCOT Tier-1

Now consider a new process model for the EBCOT tier-1 algorithm with only the RESTART mode switch turned on. This means that each pass generates an independent bitstream, thus allowing them to be executed in parallel. Also since RESET and CAUSAL flags are not set, this process preserves some of the coding efficiency of the serial process. A schematic of the new split arithmetic encoder (SAE) process is shown in Figure 6.2.

The key idea behind the SAE process is to exploit the parallelism of the AE tasks. However, unlike the pass-parallel architectures, we preserve coding efficiency and instead trade off redundancy for parallelism (i.e. speed). In SAE the tasks are reformulated; the new process introduces redundancy between the tasks, but allows for parallelism. A new task called the context state adaption (CSA) task is created. CSA is identical to that part of the AE task that calculates the context state table values. The SAE process retains the CF and AE tasks in the serial process.

Here, again we assume that the code-block has $K_{\mathrm{b}}^{\max}$ bits which are numbered from $K_{\mathrm{b}}^{\max} - 1$ down to 0. Each pass now consists of three tasks: CF, CSA, and AE. The CF tasks can be run in parallel as long as they are separated by at least one stripe. This dependency is represented by the double arrows between neighboring CF tasks in Figure 6.2. Now the CF task passes its symbol-context pairs to both the CSA and AE tasks. The CSA task computes the initial context state table values required by the AE task in the next pass. Because the CSA task needs the context state table values from the previous pass, CSA tasks must be performed in sequence. However, the SAE process eliminates the dependencies between AE tasks in the serial process. An AE task need not wait for the AE task in the previous pass to finish before it can begin work; it need only wait for the CSA task in the previous pass to finish. The CSA task is significantly simpler than the AE task; consequently, CSA runs faster and finishes earlier. Thus it is now possible to run several AE tasks simultaneously.

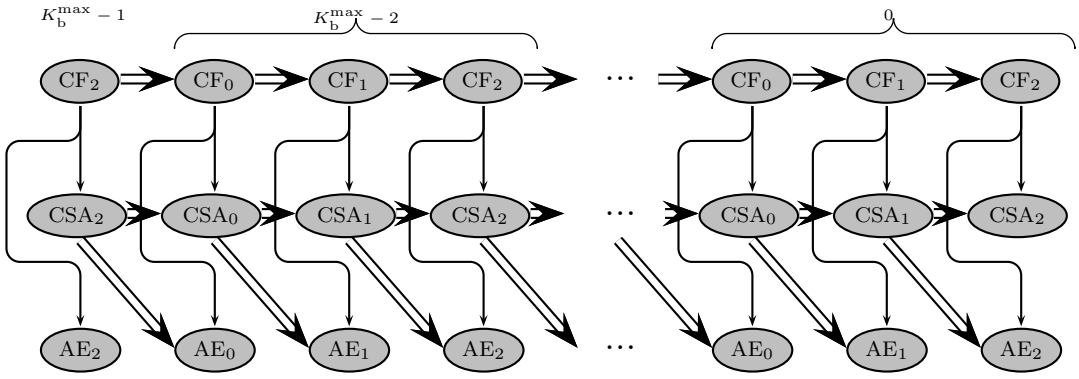


Figure 6.2: *The proposed split arithmetic encoder EBCOT tier-1 process. CSA is a new task; the causal dependency between AE tasks has been eliminated.*

For software implementations, the pseudocode for a CSA task looks like the AE task without the bitstream creation. Since bitstream creation consumes time, several clock cycles are saved if only context state adaptation is performed. For example CSA_0 and AE_0 start at the same time, but CSA_0 finishes several clock cycles before AE_0. This allows AE_1 to start before AE_0 finishes. With sufficient CSA savings, several AE tasks can be run at the same time. This results in higher system throughput.

For hardware implementations, the CSA task allows for a hardware module with smaller delay. Therefore a hardware CSA module can run on faster clocks and finish before the corresponding

hardware AE module. A second hardware AE module can be started before the first hardware AE module finishes. This idea can be extended to more copies of the hardware AE module. The result is higher system throughput.

Figures 6.3 to 6.5 shows the total number of clock cycles saved after each coding pass is completed for three sample code-blocks of sizes 16×16, 32×32, and 64×64. These savings were measured for the *Parrots* image compressed using 5 levels of the biorthogonal 9/7 wavelet transform. This estimate is based on the assumption that we save 1 clock cycle per renormalization shift. We see that the larger the code-block, the larger is the number of clock cycles saved. Figure 6.6 shows the total number of clock cycles that were saved in each code-block of the *Parrots* image. We see substantial savings of the order of approximately 20,000 clock cycles for the larger code-blocks in the luminance plane.

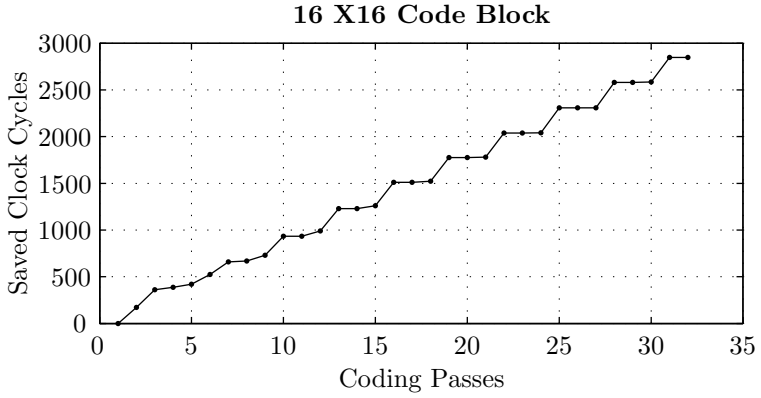


Figure 6.3: *Savings in clock cycles for a sample* 16×16 *code-block.*

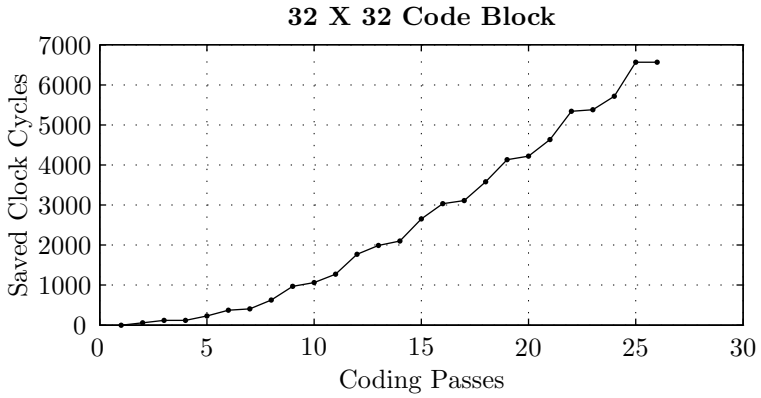


Figure 6.4: *Savings in clock cycles for a sample* 32×32 *code-block.*

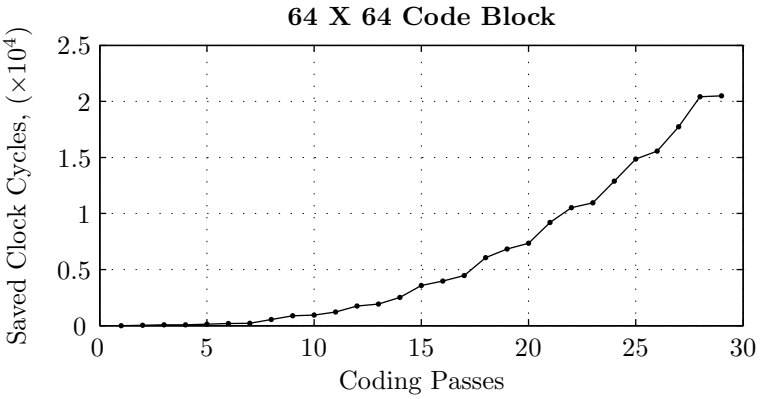


Figure 6.5: *Savings in clock cycles for a sample* 64×64 *code-block.*

6.2 Cumulative Clock Savings Analysis

Using the SAE process that was presented in the previous section, a cumulative clock savings estimation was performed. In this analysis the number of clock cycles that were saved by the CSA module for each coding pass was estimated and these savings passed on to the next coding pass. This work is based on estimation of the approximate number of clock cycles consumed to perform the various sections of the AE task. In order to perform a more accurate analysis, information regarding the actual number of clock cycles that is consumed for each section of the AE task is needed. Also this analysis assumes that the CF tasks are able to process the coefficients independent of the CSA and AE tasks. The CF tasks generate coding contexts and symbols and write them into a stack. These coding contexts and symbols are read from the stack in a first in first out (FIFO) order by the CSA and AE tasks.

6.2.1 Estimation of Clock Consumption

The pseudocode for the AE task is shown in Algorithm 9. Consider a division of the algorithm into four blocks: *INITIALIZE*, *UPDATE-INT*, *UPDATE-CS*, and *RENORMALIZE*. This listing also shows the estimated number of clock cycles that each step of the algorithm takes. These estimates assume that operations using unrelated data are run on the same clock cycle, while operations using related data run serially on separate clock cycles. The pseudocode for the **TransferByte** routine is shown in Algorithm 10. It has been divided into three blocks; clock cycle estimates are indicated.

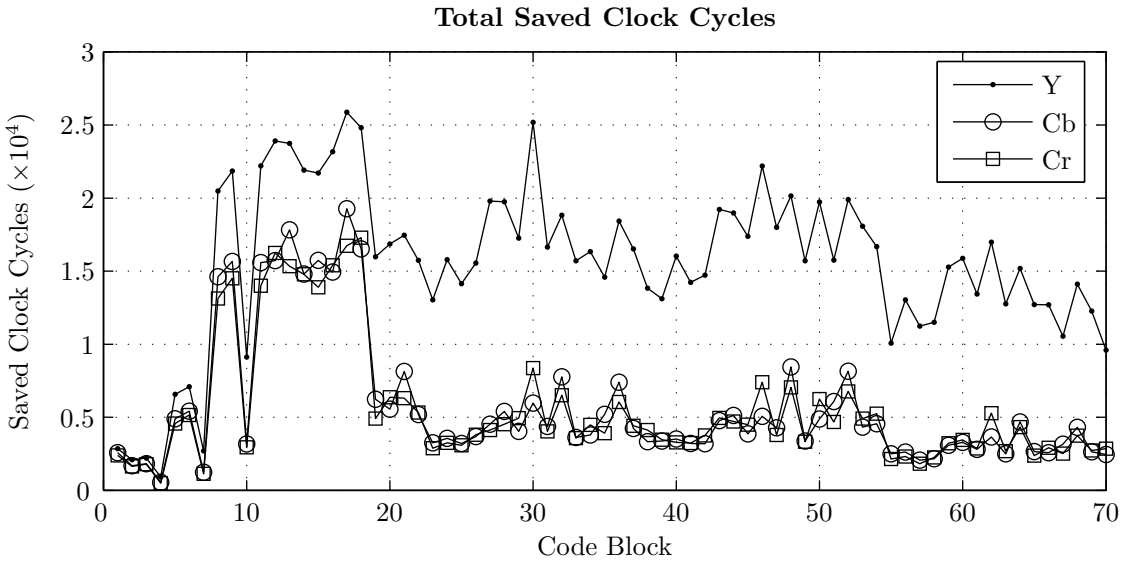


Figure 6.6: *Total savings in clock cycles for all code-blocks of the Parrots image.*

The *INITIALIZE* block consumes a total of 2 clock-cycles with line 4 consuming the first and lines 5 to 7 consuming the second clock-cycle. The *UPDATE-INT* block performs the interval updates and consumes a total of 1 clock-cycle since only one of lines 10 or 13 is run at any time. The *UPDATE-CS* performs the context-state updates and also consumes a total of 1 clock-cycle. This is because at any time, it either runs line 18 or runs lines 21 and 22. Since lines 21 and 22 are not causally related, they can be run in the same clock-cycle. Finally the *RENORMALIZE* block performs the interval renormalization as many times as required. Each time a renormalization is performed it either has to transfer a byte or not. If it does not have to transfer a byte, then this block does not consume any clock cycles since the C and A register updates can be performed in the same clock cycle as *UPDATE-CS*. On the other hand, if it has to transfer a byte, then it consumes either 5 or 4 clock cycles depending on whether or not bit-stuffing was encountered in the **TransferByte** routine. The estimation of the clock-cycle consumption for the **TransferByte** routine follows similar principles. To simplify the analysis, here we assume the worse case scenario (quicker **TransferByte**) that the **TransferByte** routine always takes 3 clock-cycles.

Algorithm 9: *Main MQ encoding algorithm.*

In : Symbol: x, Context: κ
Out: Encoded Bytes: $\bar{T}$

Use the context κ to obtain s_κ and Σ_κ from the context state table;
Use the state Σ_κ to read off the entries $\Sigma_{\text{mps}}(\Sigma_\kappa)$, $\Sigma_{\text{lps}}(\Sigma_\kappa)$, $X_s(\Sigma_\kappa)$ and $\bar{p}_{\text{LPS}}(\Sigma_\kappa)$ from the state transition table;
$s \leftarrow s_\kappa$, $\bar{p} \leftarrow \bar{p}_{\text{LPS}}(\Sigma_\kappa)$;
$A \leftarrow A - \bar{p}$ (1 CLK);
if $A < \bar{p}$ **then**
    (*conditional exchange of MPS and LPS*);
    $s \leftarrow 1 - s$ (1 CLK);
INITIALIZE, 2 CLKs

if $x = s$ **then**
    (*assign MPS the upper sub-interval*);
    $C \leftarrow C + \bar{p}$ (1 CLK) ;
else
    (*assign LPS the lower sub-interval*);
    $A \leftarrow \bar{p}$ (1 CLK);
UPDATE-INT, 1 CLK

if $A < 2^{15}$ **then**
    (*update the context state table*);
    if $x = s_\kappa$ **then**
        (*the symbol was a real MPS*);
        $\Sigma_\kappa \leftarrow \Sigma_{\text{mps}}(\Sigma_\kappa)$ (1 CLK);
    else
        (*the symbol was a real LPS*);
        $s_\kappa \leftarrow s_\kappa \oplus X_s(\Sigma_\kappa)$;
        $\Sigma_\kappa \leftarrow \Sigma_{\text{lps}}(\Sigma_\kappa)$ (1 CLK);
UPDATE-CS, 1 CLK

while $A < 2^{15}$ **do**
    (*perform renormalization shift*) ;
    $A \leftarrow 2A$;
    $C \leftarrow 2C$;
    $\bar{t} \leftarrow \bar{t} - 1$ (1 CLK);
    if $\bar{t} = 0$ **then**
        TransferByte($\bar{T}$**,** C**,** L**,** $\bar{t}$**)** (4 OR 3 CLKs);
RENORMALIZE, 1 OR (5 OR 4) CLKs

6.2.2 The CSA Task

The CSA task is comprised of *INITIALIZE*, *UPDATE-INT*, *UPDATE-CS*, and the C and A register updates in *RENORMALIZE*. Thus the CSA task consumes 4 clock cycles for each symbol-context pair generated by the CF task. It may seem that the C and A register updates in *RENORMALIZE* are being overlooked in this estimate; however, these updates are easily performed in the same clock cycle as the *UPDATE-CS* operations. Let S represent the number of symbol-context pairs generated in each coding pass. The estimated number of clock cycles consumed by the CSA task for each coding pass is:

$$N_{\text{CSA}} = 4S. \tag{6.1}$$

Algorithm 10: *The **TransferByte** routine.*

In : —
Out: Encoded Bytes: $\bar{T}$

```
1  if T̄ = 0xFF then                                              ┐
2      (cannot propagate any carry past T̄; need bit stuffing) ;  │
3      if L ≥ 0 then                                             │
4          EmitByte(T̄) (1 CLK);                                  │
5      L ← L + 1 ;                                               │ NO-PROP-CARRY, 3 CLKs
6      T̄ ← C^msbs (1 CLK);                                       │
7      C^msbs ← 0 ;                                              │
8      t̄ ← 7 (transfer 7 bits plus carry) (1 CLK);               ┘
9  else
10     T̄ ← T̄ + C^carry (propagate any carry bit from C into T̄) (1 CLK); ┐
11     C^carry ← 0 (reset carry bit) ;                                   │
12     if L ≥ 0 then                                                     │ PROP-CARRY, 2 CLKs
13         EmitByte(T̄) ;                                                 │
14     L ← L + 1 (1 CLK);                                                ┘
15     if T̄ = 0xFF then                                                  ┐
16         (decoder will see this as bit-stuffing; need to act accordingly) ; │
17         T̄ ← C^msbs (1 CLK);                                            │
18         C^msbs ← 0 ;                                                   │
19         t̄ ← 7 (transfer 7 bits plus carry) (1 CLK);                    │ BIT-STUFF, 2 CLKs
20     else                                                               │
21         T̄ ← C^partial (1 CLK);                                         │
22         C^partial ← 0 ;                                                │
23         t̄ ← 8 (transfer full byte) (1 CLK);                            ┘
24
```

6.2.3 The AE Task

The AE task consists of *INITIALIZE*, *UPDATE-INT*, *UPDATE-CS*, and *RENORMALIZE*. Let S be the number of symbol-context pairs generated by the CF task and let B represent the number of encoded bytes generated in a coding pass (i.e. the number of calls to **TransferByte**). As in the CSA task, $4S$ clock cycles are consumed through line 26 of Algorithm 9. The remaining clock cycles consumed by the AE task are those in **RENORMALIZE**. **TransferByte** consumes either 3 or 4 clock cycles; the worst case (3) is assumed. Therefore B is the number of clock cycles consumed by the updation of $\bar{t}$ and $3B$ is the number of clock-cycles consumed to execute the **TransferByte** routine during the entire coding pass. Thus the total number of clock-cycles consumed during the coding pass can be written as:

$$N_{\mathrm{AE}} = 4S + 4B. \tag{6.2}$$

6.2.4 Analysis Results

The expressions derived in the two preceding sections were used to estimate the cumulative clock-savings that can be obtained after an entire code-block was encoded. Tests were run on the *parrots* image and the starting and ending times (clock-cycle count) were computed using the aforementioned expressions. Along with the starting and ending times of the CSA and AE tasks, the same for the serial AE task was also computed. Figure 6.7 shows the results from this computation for code-block number 30 from the *parrots* image. Figure 6.8 shows a closer view of the same between coding passes 18 and 22. In Figures 6.7 and 6.8 the x-axis is an index to the coding

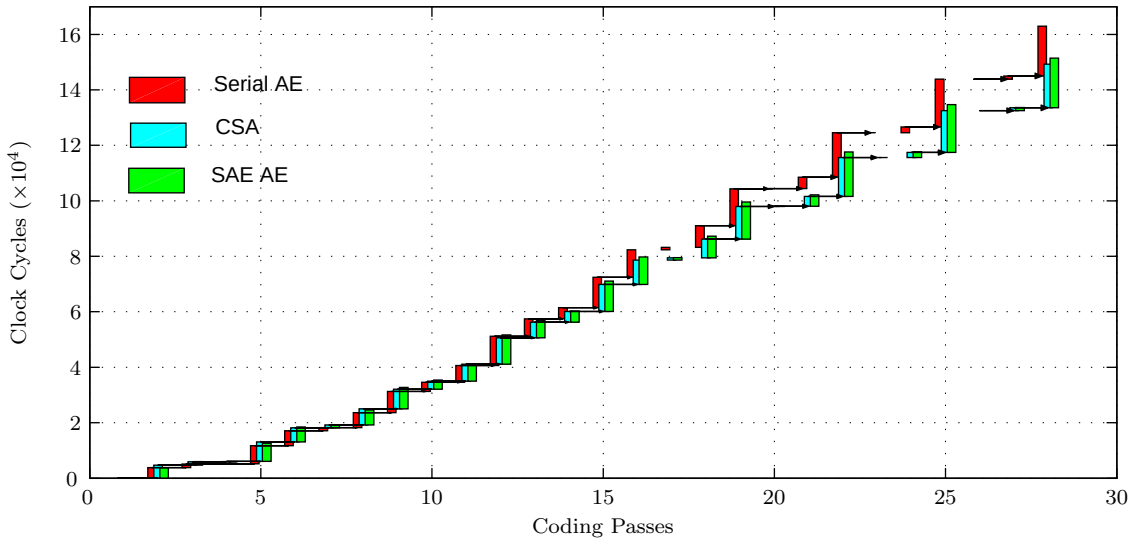


Figure 6.7: *Estimated timing of CSA and AE tasks for the SAE process and AE tasks for the serial process for code-block number 30 of the parrots image.*

passes. The first coding pass is the cleanup pass for the highest bitplane and the last coding pass is the cleanup pass for the lowest bitplane. The y-axis represents the number of clock cycles completed since the beginning of the encoding process for that code-block. Here the duration of a particular task is represented by a colored rectangle. This code-block has a total of 28 coding passes corresponding to 10 bitplanes. In the figures, each coding pass is associated with three colored timing-rectangles, red for the AE task of the serial process, cyan for the CSA task of the SAE process and green for the AE task of the SAE process. The causal relationship between modules of adjacent coding passes is represented by the arrows.

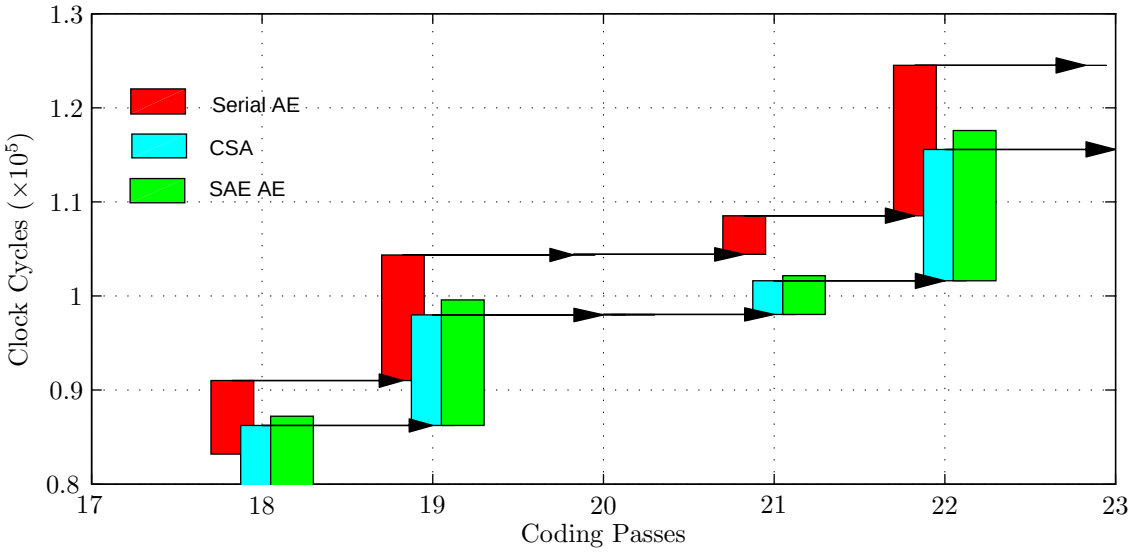


Figure 6.8: *Estimated timing of CSA and AE tasks for the SAE process and AE tasks for the serial process for code-block number 30 of the parrots image (blown-up view).*

Coding pass 20 is a significance propagation pass; coding passes 18 and 21 are magnitude refinement passes; and coding passes 19 and 22 are cleanup passes. In Figure 6.8 it is evident that CSA_{18} finishes before AE_{18}. Therefore CSA_{19} and AE_{19} are able to start before AE_{18} finishes. It is the cumulative effect of these clock cycle savings that realize the SAE speedup. Serial AE_{28} (the final pass for code-block 30) finishes 13,705 clock cycles after SAE's AE_{28}. This is equivalent to 8.4% cumulative savings in clock cycles for the code-block. Note: the execution time of pass 20 is too small to appear on this scale.

Figures 6.9, 6.10, and 6.11 show the cumulative clock savings obtained at the end of encoding of each of the 210 code-blocks in the *parrots* image. The clock-cycle savings obtained here for each code-block are the difference between the finishing times of the serial AE tasks and our new SAE AE task at the end of the last coding pass in each code-block. Figure 6.9 shows the raw number of clock-cycles saved in each of the 210 code-blocks. This computation was done for a fairly coarse quantization of the DWT coefficients (at the knee of the quantization-PSNR curve [29]). As the quantization is made finer, the encoder has to perform more and more magnitude refinement passes. As more and more magnitude refinement passes are performed we expect to see even more savings in clock cycles due to the new architecture.

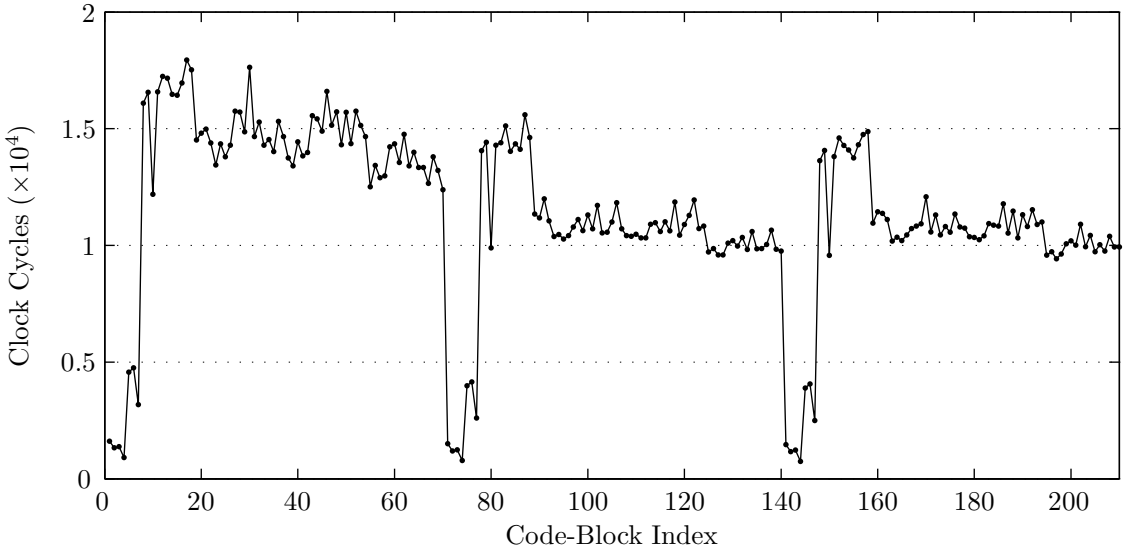


Figure 6.9: *Estimated number of clock-cycles saved at the end of encoding each code-block for the parrots image.*

Figure 6.10 shows the percentage savings in clock cycles for each of the 210 code-blocks in the image. These percentages were computed with respect to the total number of clock-cycles consumed by the last of the serial AE task. Finally Figure 6.11 shows the cumulative percentage savings in clock-cycles as each code-block is encoded. Note that at this quantization level, we obtain approximately 8% in clock-cycle savings. Figure 6.9 reveals that code-block ranges [1, 7], [71, 77], and [141, 147] result in smaller savings; this occurs because these are the small code-blocks associated with the lowest resolution subbands in each of the three color components.

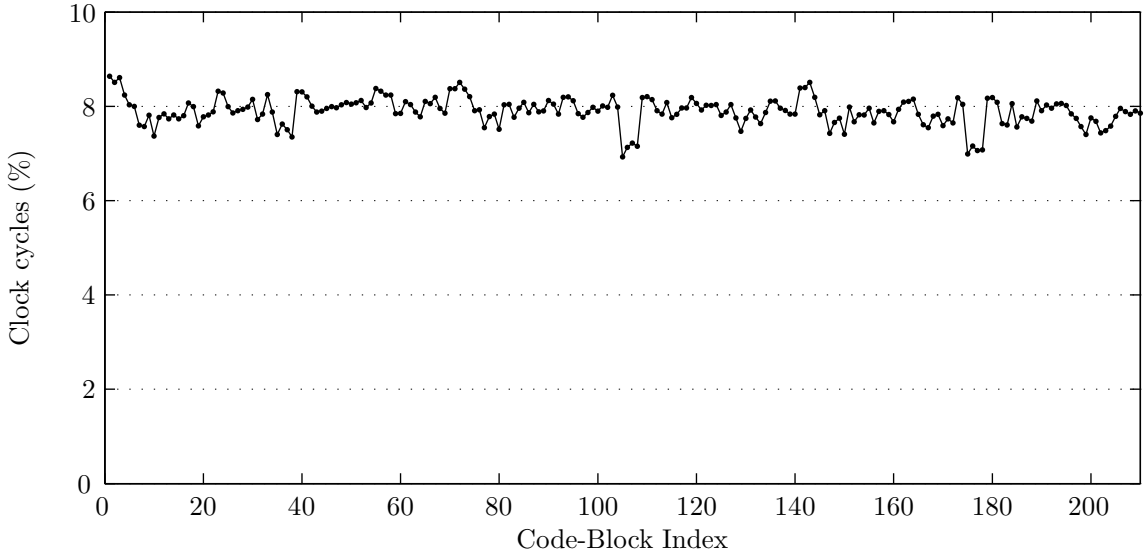


Figure 6.10: *Estimated percentage of clock-cycles saved at the end of encoding each code-block for the parrots image.*

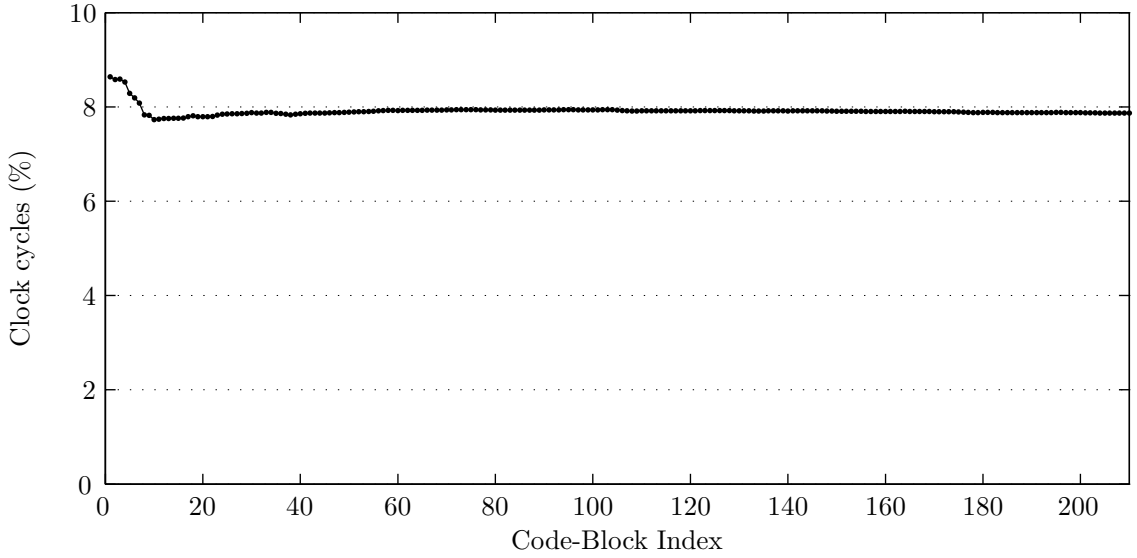


Figure 6.11: *Estimated cumulative percentage of clock-cycles saved as each code-block for the parrots image gets encoded.*

Chapter 7

Software Implementation of the SAE Process

This chapter presents a multithreaded software implementation of the SAE process that was presented in Chapter 6. The execution time and throughput results for a conventional serial implementation of the process shown in Figure 6.1 is compared with a multithreaded implementation of the SAE process shown in Figure 6.2. An estimate of the potential savings showed that the SAE process is capable of saving approximately 8% of encoding time. Here we analyze the performance of an actual software implementation of the new process.

7.1 A Note on the Computing Platforms

The multithreaded software was implemented in C on two platforms. The first is a 64-processor SGI Altix machine, called *Inferno2*, which is a non-uniform memory access (NUMA) system, running Linux. This machine uses Intel's Itanium II processors running at 1.6 GHz. The second platform is a PC with a Pentium IV, 2.8 GHz, simultaneous multithreading (SMT) processor [30,31] with two virtual processing units, also running Linux. In each case the processor cycle count register was used to measure the execution time of the implementation. In the case of *Inferno2* each thread is allocated its own real processor whereas in the case of the PC the OS multiplexes all threads onto the two virtual processors exported by the hyperthreaded processor.

7.2 The Serial Implementation

The conventional serial architecture was implemented using C on Linux for both *Inferno2* as well as the PC. The wavelet coefficients for 16 test images are quantized to 32 bits and stored in data files. The serial implementation loads all the quantized wavelet coefficients from all code-blocks to memory and converts them to bitplane form. In other words, the array of 32-bit integers representing the code-block coefficients is converted to 32 arrays of 1 bit each. Here each array represents a bitplane and each element in the array represents the bit value of the coefficient in that position in that bitplane. This conversion speeds up the CF module by arranging the data in the form that it is accessed. The same data arrangement is also used in the multithreaded implementations described later in this chapter. The *main*() function performs this conversion and calls the main encoding function multiple times, one after the other, for each code-block in the image.

The main encoding function creates the database required for book-keeping (state variables) as well as space for the output encoded bitstreams. This function calls the three coding pass functions in turn starting with pass 2 for the highest bitplane, then followed by all three passes for each of the succeeding bitplanes. Within each coding pass function, symbol-context pairs are generated depending on the significance state of the coefficient being encoded as well as that of its neighbors. The arithmetic encoder is called as soon as new symbol-context pairs are generated, so there is no need to store any of the symbol-context pairs. The arithmetic encoder function generates encoded bytes and concatenates them into the bitstream.

The time at which any event occurs (for example, ending time of the encoding process) is computed by reading the cycle count from a processor register and subtracting it from the cycle count at the beginning of the encoding process. This cycle count can be directly converted into time by using the known clock frequency of the processor.

7.3 The SAE Implementation on *Inferno2*

The SAE process was implemented using a multithreaded, task-parallel software architecture. The pre-processing of the quantized coefficient data for each code-block is very similar to that performed in the serial implementation. In other words, the quantized coefficients are read from the input

data file and loaded into memory. These coefficients are converted to bitplane form and fed to the CF module. The manner in which the CF, CSA and AE processing is performed is different from the serial implementation.

First a list of tasks to be performed is created based on the number of code-blocks as well as the number of bitplanes in each code-block. Following the pass-parallel idea in [24], the CF processing is divided into two tasks (or processing elements), PE1 and PE2. PE1 represents the CF processing for Pass 0 for all bitplanes from all code-blocks. PE2 represents the CF processing for both Pass 1 and Pass 2 for all bitplanes from all code-blocks. These two tasks can run in parallel as long they are spatially separated by at least one stripe. The third task represents the CSA processing for all passes from all code-blocks. These three tasks are followed by several tasks that represent the AE processing for each pass. For example if a code-block has $K_b^{\max}$ bitplanes, then it adds $3K_b^{\max} - 2$ AE tasks to the list.

These tasks are performed in multiple, parallel threads that implement the SAE process. Following a task-parallel model, each thread repeatedly performs the next pending task from a queue until there are no more left. Since the number of threads controls the available concurrency, it was controlled using a command line argument. Speedup is achieved in two ways: (1) the CF, CSA and AE tasks are performed in parallel in multiple threads, and (2) the introduction of the fast CSA task allows multiple AE tasks to be run in parallel.

The data dependency conditions of the SAE process are maintained by blocking a thread until its data is ready. The data dependency is summarized by two conditions: (1) the CSA processing for each pass must wait until all CF processing for that pass is finished; (2) the AE processing for each pass must wait until all CF processing for that pass has finished and all CSA processing for the previous pass has finished. The implementation uses locks and condition variables in combination with suitable flags.

As an example, consider the multithreaded software design with five threads depicted in Figure 7.1. The columns labeled T1, T2, T3, T4, and T5 each represent a thread. T1 performs PE1 processing and T2 performs PE2 processing; Figure 7.1 shows PE1 and PE2 grouped together as CF. T3 performs the CSA processing for each pass (one after the other). Threads T4 and T5 both perform AE processing. Some AE tasks are processed by T4, while others are processed by T5 (depending on which thread was waiting to pick up the next pending AE task).

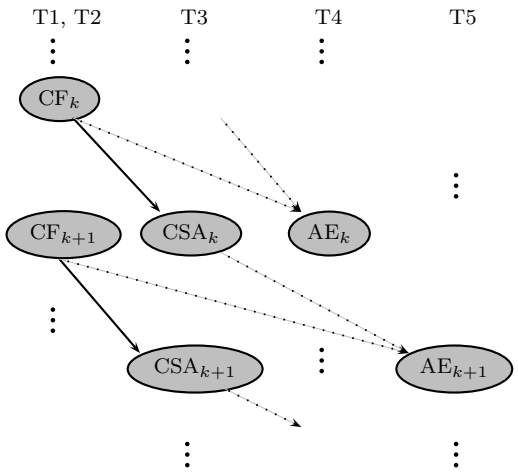


Figure 7.1: *Blocking and signaling mechanism for the multithreaded implementation of the SAE process for EBCOT tier-1.*

Thread T3 (CSA) checks before each pass that all symbol-context pairs for that pass have been generated by the CF task (PE1 and PE2). If not, it blocks and waits for the CF task to finish with that pass. This dependency is denoted by the solid arrows in Figure 7.1. The thread executing the CF task signals the CSA thread once the pass is complete. The CSA thread uses a fast double-locking idiom to avoid locking in case the CF task has already finished. This is implemented as follows. A flag called CSA_READY (one for each coding pass) is used to effect this mechanism. This flag is initialized to 0 and set to 1, once, by the CF thread processing the coding pass when it is complete. The CF thread uses appropriate locks while setting the flag and signals the CSA thread afterwards. The CSA thread first checks the value of the flag (without any locking) and only if the flag has not been set, does it need to go into a "wait on signal" state. Thus in most cases the CSA thread is able to avoid expensive locking and waiting operations.

Threads T4 and T5 (AE) check that two conditions are satisfied before starting AE_{k+1}: (1) all symbol-context pairs for the bitplane have been generated by CF_{k+1}; and, (2) CSA_k has finished so that the latest context state table is available. These two conditions are represented by the dotted arrows from the CF and CSA threads to the AE threads in Figure 7.1. The AE threads will block until those conditions are satisfied. Both the CF and the CSA threads increment a flag called AE_READY when their individual conditions are met. The thread that changes the flag to 2

signals the waiting AE thread. The AE thread checks whether the flag has value 2; if not, it waits for a signal from either the CF thread or the CSA thread.

A similar method is employed for the synchronization between the PE1 and PE2 threads. The separation between PE1 and PE2 must be maintained within the range $[2, N-2]$, where N is the number of stripes in the code-block. The maximum separation arises when the PE1 thread is blocked until the PE2 thread has processed the succeeding stripe for the previous bitplane. In other words, significance propagation for bitplane j cannot commence until cleanup for bitplane $j-1$ has completed. The minimum separation arises when the PE2 thread is blocked until the PE1 thread has processed the succeeding stripe of the current bitplane. In other words, within a bitplane, magnitude refinement and cleanup for stripe s cannot commence until significance propagation for stripe $s+1$ has completed. This synchronization preserves causality between passes and produces the correct symbol-context pairs.

The PE1 and PE2 threads again implement this synchronization efficiently using a double locking idiom. The synchronization is implemented at the stripe level to avoid excessively fine (and therefore expensive) synchronization. A flag called PE_Flag is created for each stripe of each bitplane for each code-block and initialized to 0. Whenever PE1 and PE2 threads finish processing the data for a stripe, the corresponding flag is incremented. Before processing the data for a stripe, PE1 checks to ensure that PE2 has already processed the succeeding stripe from the previous bitplane (by checking to see that PE_FLAG for that stripe is 2). This ensures that any significance updates that occurred in the cleanup pass for the previous bitplane are reflected in the state variables of the neighboring stripes. Also before processing the data for a stripe, PE2 checks to ensure that PE1 has already processed the succeeding stripe from the current bitplane (by checking to see that PE-FLAG for that stripe is 1). This ensures that magnitude refinement and cleanup for the stripe are performed using the up-to-date significance state variables for neighboring stripes.

The AE tasks are picked up by waiting threads using two different strategies. In one implementation, each waiting thread would pick up the next AE task that was ready to be processed and completes it. Thus each AE task may be performed in a separate working thread. In a second implementation, the working thread waits until all three AE tasks from a bitplane are ready to be processed, and then performs these three AE tasks in sequence one after the other. Thus at any given time, a working thread always picks up three AE tasks at a time. This second implementa-

tion was designed to minimize the excessive processing overhead (if any) of checking for appropriate conditions for the AE tasks to go ahead. This implementation also ensures that the load on each of the AE threads is approximately equal. It was found from simulations that usually, within a bitplane, one of the coding passes dominates in terms of number of symbol-context pairs produced. This means that one of the AE tasks is much longer than the other two. This kind of staggered load does not allow the different AE tasks to run with much overlap in time. The implementation that runs three AEs per thread equalizes the load in the different AE threads since the total number of symbol-context pairs that is generated from each bitplane is approximately the same.

7.4 The SAE Implementation on the PC

Since the PC runs an SMT processor with two virtual processing units, it had more limited parallel processing capability than the *Inferno2* machine. However, the SAE implementation on the PC had one important simplification. In the *Inferno2* implementation the CF task was split into two: PE1 and PE2 and run in parallel in two separate threads so that symbol-context pairs from the three coding passes may be generated in parallel so that subsequent AE tasks have more chances of running in parallel. In the case of the PC implementation, the CF task was maintained as a monolithic task and one thread performed this task for all coding passes of all code-blocks. The three coding passes were performed in sequence one after the other within the CF thread. The CSA thread performed the CSA tasks for all coding passes from all code-block one after the other. Several AE threads performed the AE processing simultaneously as soon as the CF data for the current coding pass as well as the context state table from the previous pass are made available. Each AE thread picks up the three AE tasks from a bitplane. These simplifications are necessary to minimize the scheduling overhead results from a parallel algorithm running on limited parallel processing resources. These simplifications also mean that the symbol-context pairs are no longer available to the parallel AE threads at the same fast rate as the *Inferno2* implementation.

7.5 Results

In order to estimate the maximum possible gain from parallelism, the SAE implementation was profiled to determine the amount of time spent in the serial portion of the code vs. the amount of

time spent in the parallel portion of the code. Linux's *gprof* utility was used; *gprof* uses statistical profiling to estimate the time spent in each function. A comparison of the profiling results of the serial and the multithreaded implementations on *Inferno2* is shown in Table 7.1. The results for the multithreaded implementation were generated by letting it run in a single thread.

Table 7.1: *Software profiling results for Inferno2.*

Task	**Serial**			**Multithreaded SAE**		
	Function	**Actual**	**%**	**Function**	**Actual**	**%**
CF	*EBCoderPass0* ( )	17.92	27.96			
	EBCoderPass1 ( )	13.16	20.53	*PE1* ( )	18.44	26.24
	EBCoderPass2 ( )	21.27	33.19	*PE2* ( )	32.40	46.11
	Overall	**52.35**	**81.68**	*Overall*	**50.84**	**72.36**
CSA	–	–	–	*allCSA* ( )	**8.61**	**12.25**
AE	*MQEncode* ( )	10.83	16.90	*passMQ* ( )	9.87	14.05
	transferByte ( )	0.51	0.80	*transferByte* ( )	0.54	0.77
	putByte ( )	0.40	0.62	*putByte* ( )	0.40	0.57
	Overall	**11.74**	**18.32**	*Overall*	**10.81**	**15.39**
Total		**64.09**	**100**		**70.26**	**100**

It was found that overall, 72.36% of the time was spent in CF processing, 12.25% in CSA processing and 15.39% in AE processing. As expected the AE processing consumes more time than the CSA processing. The implementation splits the CF task into PE1 and PE2 which are performed in parallel and they consume 26.24% and 46.11% respectively. In the ideal parallel processing environment three conditions exist: (1) there are an unlimited number of processing units, (2) there is no data dependency between the different parallel portions of the code, and (3) there is no overhead for synchronization. Under these ideal conditions the PE1, PE2, CSA and AE tasks could all run simultaneously and the total encoding time would be the time spent in the PE2 module (since that is the largest serial task).

Therefore, under these ideal conditions, according to Amdahl's law [32] the maximum savings in encoding time that could be obtained for this image is approximately $100 - 46.11 = 53.89\%$. This corresponds to a speedup factor of 2.17. In reality the number of processing units is a constraint. For example on *Inferno2* the limit on the number of processes that may be started simultaneously by a single user is 6 (on 6 different processors). Also, the implementation has its own method to limit the number of processing units by controlling the number of working threads (where each

Table 7.2: *Throughput improvement for SAE implementation using 1 AE per thread on Inferno2.*

Images	Serial	SAE(3)		SAE(4)		SAE(5)		SAE(6)		SAE(7)		SAE(8)		SAE(9)		SAE(10)	
		Thrpt	%	Thrpt	%	Thrpt	%	Thrpt	%	Thrpt	%	Thrpt	%	Thrpt	%	Thrpt	%
Aerial2	171128	236445	38.17	263119	53.76	261295	52.69	259625	51.71	255972	49.58	260047	51.96	258180	50.87	258805	51.23
Bike	60411	84191	39.36	91580	51.59	91510	51.48	89850	48.73	89614	48.34	90941	50.54	90780	50.27	89013	47.35
Bike3	57146	78501	37.37	85406	49.45	84672	48.17	83429	45.99	85889	50.30	84449	47.78	85032	48.80	82104	43.67
Cafe	58456	78982	35.11	85648	46.52	88358	51.15	88284	51.03	83727	43.23	85077	45.54	88297	51.05	87632	49.91
Cats	64566	86301	33.66	99811	54.59	95246	47.52	98077	51.90	98249	52.17	97440	50.92	96175	48.96	96537	49.52
F21_200	60283	78787	30.70	89011	47.66	89075	47.76	88288	46.46	87007	44.33	86145	42.90	81276	34.82	87252	44.74
Finger	174609	238248	36.45	264564	51.52	264118	51.26	259058	48.36	263690	51.02	262933	50.58	261154	49.57	254033	45.49
Gold	61081	80851	32.37	90770	48.61	64562	5.70	90234	47.73	88532	44.94	90413	48.02	90094	47.50	88995	45.70
Hotel	60061	79567	32.48	89958	49.78	89584	49.16	88531	47.40	87054	44.94	88653	47.60	88559	47.45	88468	47.30
Ingres1	64580	89646	38.81	98545	52.59	97604	51.14	97829	51.48	93309	44.49	97546	51.05	96434	49.32	92859	43.79
Ingres2	63581	86755	36.45	96642	52.00	95943	50.90	94677	48.91	92734	45.85	93837	47.59	93388	46.88	95425	50.08
Ingres3	65493	89608	36.82	98024	49.67	97086	48.24	99799	52.38	97690	49.16	99649	52.15	99062	51.26	95132	45.26
Ingres4	64853	87607	35.09	99062	52.75	98342	51.64	98159	51.36	97908	50.97	94240	45.31	98130	51.31	97563	50.44
Parrots	63019	85046	34.95	94725	50.31	93949	49.08	92805	47.27	93887	48.98	93650	48.61	93692	48.67	90940	44.31
Tools	57727	78840	36.57	87132	50.94	85127	47.46	82572	43.04	84161	45.79	83930	45.39	84077	45.65	81498	41.18
Woman	62570	84611	35.23	92333	47.57	94654	51.28	93936	50.13	93893	50.06	93988	50.21	93218	48.98	92864	48.42
Average	66529	90310	35.75	100143	50.53	99577	49.67	99716	49.88	98183	47.58	98713	48.38	98420	47.94	98118	47.48

working thread runs on a separate processor). On the PC, only two virtual processing units are available. In addition the data dependency between the various parallel portions of the code means that they cannot all start simultaneously, they have to wait for certain conditions to be met before they start. Thus the various parallel portions of the code execute in a staggered fashion in time.

Sixteen images from the standard image set were used to obtain processing time statistics and measure throughput and speedup [18]. The improvement in throughput of the multithreaded SAE implementation with 1 AE per thread is shown in Table 7.2. Here throughput is expressed in pixels/second and was computed by dividing the number of pixels in the image by the time taken to encode it. The time taken to encode the images was computed by averaging the results from three independent runs. For the cases of 4, 5, and 6 working threads, the improvements in throughput are 50.53%, 49.67%, and 49.88% respectively. These translate to time savings of approximately 33% or speedup factors of approximately 1.5. Table 7.3 provides the throughput improvements for the multithreaded SAE implementation with 3 AEs per thread which gives 41-42% improvement using 4, 5, or 6 threads.

Table 7.3: *Throughput improvement for SAE implementation using 3 AEs per thread on Inferno2.*

Images	Serial	SAE(3)		SAE(4)		SAE(5)		SAE(6)		SAE(7)		SAE(8)		SAE(9)		SAE(10)	
		Thrpt	%	Thrpt	%	Thrpt	%	Thrpt	%	Thrpt	%	Thrpt	%	Thrpt	%	Thrpt	%
Aerial2	171128	238841	39.57	262077	53.15	258348	50.97	255387	49.24	255086	49.06	256332	49.79	257909	50.71	257441	50.44
Bike	60411	82516	36.59	89963	48.92	88941	47.23	89328	47.87	86029	42.41	88018	45.70	87939	45.57	83988	39.03
Bike3	57146	77070	34.87	80829	41.44	78715	37.74	77589	35.77	78977	38.20	78383	37.16	78352	37.11	77204	35.10
Cafe	58456	75904	29.85	83038	42.05	82421	41.00	82062	40.38	80824	38.26	79379	35.79	79623	36.21	81499	39.42
Cats	64566	83463	29.27	92483	43.24	88993	37.83	91734	42.08	91757	42.11	91452	41.64	90920	40.82	87301	35.21
F21_200	60283	76649	27.15	84243	39.75	83505	38.52	83074	37.81	81969	35.97	82742	37.26	82691	37.17	81071	34.48
Finger	174609	204782	17.28	202232	15.82	220365	26.20	213595	22.33	232509	33.16	207393	18.78	220105	26.06	217461	24.54
Gold	61081	75372	23.40	82759	35.49	82201	34.58	75251	23.20	81436	33.32	79490	30.14	81909	34.10	79261	29.76
Hotel	60061	75381	25.51	81140	35.10	79255	31.96	79290	32.02	82425	37.24	78309	30.38	82473	37.32	79595	32.52
Ingres1	64580	85217	31.96	92330	42.97	91259	41.31	91816	42.17	91381	41.50	91419	41.56	90634	40.34	88621	37.23
Ingres2	63581	83643	31.55	91071	43.24	89817	41.26	88353	38.96	89511	40.78	85502	34.48	87444	37.53	82907	30.40
Ingres3	65493	85547	30.62	88340	34.88	90518	38.21	92252	40.86	90791	38.63	91764	40.11	91967	40.42	91066	39.05
Ingres4	64853	84818	30.79	90220	39.11	88365	36.25	88255	36.08	90984	40.29	90257	39.17	90297	39.23	90624	39.74
Parrots	63019	80116	27.13	82725	31.27	86098	36.62	82150	30.36	82340	30.66	85333	35.41	84922	34.76	84146	33.52
Tools	57727	75134	30.15	81179	40.63	80241	39.00	70954	22.91	79980	38.55	79168	37.14	78716	36.36	80146	38.84
Woman	62570	79619	27.25	89596	43.19	88621	41.63	87998	40.64	87415	39.71	87012	39.06	87228	39.41	85787	37.11
Average	66529	87007	30.78	94467	41.99	93438	40.45	93157	40.02	93138	40.00	92712	39.36	92921	39.67	91233	37.13

Thus we are able to realize substantial speedup using the multithreaded SAE implementation, despite its not achieving the ideal prediction from Amdahl's law. Apart from the inability to perform all the parallel parts of the simultaneously (due to lack of resources as well as data dependency), the reduction in speedup from the ideal is a result of the nature of the profiling operation. Because profiling was performed by rolling the multiple threads out into a single thread, it did not account for synchronization overhead between parallel threads. In real multithreaded implementations, these synchronization overheads can contribute substantially to the loss of efficiency. Real multithreaded applications run slower than profiling predictions because the cost of synchronization is highest when there is real blocking for conditions. This is particularly true for the finer synchronization between PE1 and PE2 for each stripe.

A second factor that affects the speed of a multithreaded application on a NUMA architecture is that it is more expensive to allocate memory in one thread and access it in another. Since each thread runs on a separate CPU with its own local memory resources, the remote memory accesses required in the multithreaded SAE software add to the processing time of the implementation.

Table 7.4: *Performance comparison of serial and SAE implementations on the PC.*

Images	Time (sec)				Throughput (pixels/sec)				% Improvement						Speedup		
									Time			Thrpt					
	Ser	(3)	(4)	(5)	Ser	(3)	(4)	(5)	(3)	(4)	(5)	(3)	(4)	(5)	(3)	(4)	(5)
Aerial2	19.23	16.23	16.26	16.26	218072	258429	257993	257905	15.62	15.47	15.45	18.51	18.31	18.27	1.19	1.18	1.18
Bike	54.52	45.89	45.95	45.95	76933	91392	91283	91283	15.82	15.72	15.72	18.79	18.65	18.65	1.19	1.19	1.19
Bike3	14.48	12.20	12.22	12.22	72393	85925	85826	85794	15.75	15.65	15.62	18.69	18.56	18.51	1.19	1.19	1.19
Cafe	56.58	47.74	47.79	47.81	74125	87856	87762	87731	15.63	15.54	15.51	18.52	18.40	18.36	1.19	1.18	1.18
Cats	51.52	43.18	43.24	43.26	81407	97130	96997	96964	16.19	16.07	16.04	19.31	19.15	19.11	1.19	1.19	1.19
F21_200	54.19	45.95	46.01	46.04	77399	91278	91153	91097	15.21	15.09	15.04	17.93	17.77	17.70	1.18	1.18	1.18
Finger	1.20	1.02	1.02	1.02	217972	257865	258143	258039	15.47	15.56	15.53	18.30	18.43	18.38	1.18	1.18	1.18
Gold	3.44	2.90	2.90	2.91	76228	90325	90258	90067	15.61	15.54	15.36	18.49	18.41	18.15	1.18	1.18	1.18
Hotel	3.49	2.95	2.96	2.96	75132	88882	88528	88659	15.47	15.13	15.26	18.30	17.83	18.00	1.18	1.18	1.18
Ingres1	51.28	43.25	43.29	43.39	81786	96988	96895	96666	15.67	15.59	15.39	18.59	18.47	18.19	1.19	1.18	1.18
Ingres2	52.22	44.01	44.08	44.08	80319	95301	95160	95148	15.72	15.60	15.59	18.65	18.48	18.46	1.19	1.18	1.18
Ingres3	50.76	42.77	42.83	42.84	82634	98056	97929	97911	15.73	15.62	15.60	18.66	18.51	18.49	1.19	1.19	1.18
Ingres4	51.35	43.26	43.32	43.32	81676	96958	96826	96826	15.76	15.65	15.65	18.71	18.55	18.55	1.19	1.19	1.19
Parrots	3.35	2.84	2.84	2.84	78329	92402	92454	92254	15.23	15.28	15.09	17.97	18.03	17.78	1.18	1.18	1.18
Tools	14.48	12.18	12.20	12.21	72396	86058	85921	85874	15.87	15.74	15.69	18.87	18.68	18.62	1.19	1.19	1.19
Woman	53.32	44.85	44.91	44.92	78656	93520	93386	93372	15.89	15.77	15.76	18.90	18.73	18.71	1.19	1.19	1.19
Average	33.46	28.20	28.24	28.25	84209	99924	99795	99749	15.73	15.62	15.58	18.66	18.51	18.45	1.19	1.19	1.18

The same implementation running on the hyperthreaded PC resulted in only a maximum of 13% savings over the serial implementation. This is attributable to all threads' competition for the PC's two virtual processing units. To avoid the overhead of multiplexing 4 or more threads onto 2 virtual CPUs, the implementation was simplified as described in Section 7.4. This simplified implementation obviates the need for stripe-level synchronization between the PE1 and PE2 threads and also reduces the need for pass-level synchronization between the CF, CSA and AE tasks. Synchronization between the CF, CSA, and AE threads is now needed only at the bitplane-level. Profiling on the PC suggests a theoretical maximum improvement in execution time of about 30% for the monolithic CF case, assuming an infinite number of CPUs. Table 7.4 lists the savings obtained from this simplified implementation on the PC.

On average, the savings in throughput are 18.66%, 18.51%, and 18.45% for 3, 4, 5 threads respectively. These translate to savings in time of 15.73%, 15.62%, and 15.58% respectively. These savings are about half of that predicted by Amdahl's law.

Chapter 8

FPGA Implementation of the SAE Process

A new EBCOT tier-1 process called the split arithmetic encoder process was presented in Chapter 6. This process improves the throughput of an EBCOT tier-1 implementation by simultaneously performing multiple AE tasks while preserving a substantial portion of the coding efficiency of the encoder. An estimate of the potential savings of the new process showed that approximately 8% of encoding time can be saved. A multithreaded software implementation on a multiprocessor computing platform showed that the new process can improve the throughput by 50%. In this chapter an FPGA implementation of the SAE process offers even greater throughput improvement.

8.1 A Note on FPGA Hardware Implementations

Field programmable gate arrays (FPGA) are general purpose, reconfigurable arrays of repeated logic elements. These logic elements may be configured and connected in a variety of ways to achieve a desired functionality. The reconfigurable nature of the FPGA also means that they encounter larger combinatorial delays within circuits. This means that circuits implemented on an FPGA can only run on slower clocks when compared to the same circuits implemented in VLSI. Therefore FPGAs tradeoff flexibility for speed. However, hardware processing may be duplicated at reasonable cost; consequently, algorithms that are inherently parallel in nature can be effectively implemented on an FPGA to achieve substantially higher throughputs. In addition FPGAs consume less power than VLSI implementations making them more suitable for low power mobile applications.

The serial and the proposed SAE architectures were modeled in VHDL. Two versions of the SAE architecture were realized: the first, denoted as single symbol CF SAE, has a context formation

unit that processes one bit at a time; the second, denoted as multiple symbol CF SAE, uses the idea in [22] to process an entire stripe column at once. Architectures' functionality was verified by simulating the systems in Modelsim 6.0 for sixteen test images from [18] and comparing the resulting streams of symbol-context pairs and embedded bitstreams to expected outputs. The hardware properties of the systems were obtained by synthesizing the VHDL in Altera's Quartus II version 5.0 FPGA synthesis hardware for an APEX20KE EP20K1000EFC672-1X device.

8.2 The Single Symbol CF Monolithic AE Implementation

The serial architecture for the EBCOT tier-1 encoder was derived from the serial process shown in Figure 6.1. We call this implementation "single symbol CF" because the CF module produces at most one symbol-context pair per clock cycle. We call it "monolithic AE" because the CSA processing has not been separated from the AE processing. Since all the CF tasks are performed serially, one after the other, they are assigned to the same CF hardware module. Similarly, since all the AE tasks are performed serially, they are assigned to the same AE hardware module. A schematic of the FPGA implementation of the serial architecture is shown in Figure 8.1. The CF module processes the quantized coefficients of each code-block and generates symbol-context pairs which are stored in a FIFO. The AE module reads the symbol-context pairs, one at a time from the FIFO and encodes them into a single bitstream for each code-block.

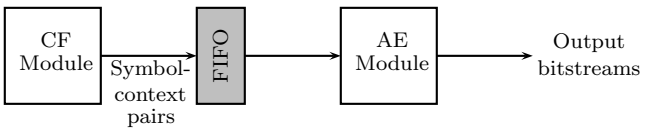


Figure 8.1: *Block diagram of a serial implementation of EBCOT tier-1 on FPGA.*

The rate of production of symbol-context pairs by the CF module is different from the rate of consumption of symbol-context pairs by the AE module. For example, during run-mode in the cleanup pass (Pass 2), when there is a run interruption, as many as four symbol-context pairs may be generated. Similarly whenever a coefficient becomes significant, two symbol-context pairs are generated. On the other hand, some clock cycles may not generate any symbol-context pairs at all. However, the AE module always consumes one symbol-context pair every clock cycle. To mitigate this mismatch in data production versus data consumption, a first in first out (FIFO) buffer is used

between the CF and AE modules. The CF module generates the symbol-context pairs and writes them to the FIFO. The AE module reads the symbol-context pairs from the FIFO and encodes them into bitstreams.

In order to generate symbol-context pairs for each coefficient bit, the CF module needs access to five sets of data: (1) the coefficient bit, (2) the sign bit of the coefficient, (3) the significance state of the coefficient and its eight neighbors, (4) the delayed significance state of the coefficient, and (5) pass membership state of the coefficient. The maximum allowable size of a code-block is 64×64. So each set of data needs a memory of size $64 \times 64 = 4\text{K}$; consequently, a single code-block needs a memory of size $5 \times 4\text{K} = 20\text{K}$. The data in this memory is accessed in a stripe-column order. This is a complex memory access requirement and the bitplane data is stored in a manner consistent with this requirement to prevent increased latency and delay for memory reads. In particular, the information bits corresponding to each stripe column are grouped together to form a data word. The FPGA system is capable of reading one data word in each clock cycle. Also the information about neighboring stripes are made available in separate memories. This allows information from as many as three stripes to be read simultaneously in one clock cycle thereby minimizing the number of required memory accesses.

The AE module converts the symbol-context pairs to an embedded bitstream through a series of subtractions, additions, shifts and look-up table functions. The memory organization for the AE module is less challenging. The only memory requirement in the AE module is the state transition table, and it is easily implemented as a 1K ROM. The context state table and internal registers of the AE module are modeled as registers.

The FIFO between the CF and AE modules allows the two modules to be run at different clock frequencies. In our implementation, the AE module has the longer critical path and must be run at the slower clock frequency (10 MHz). The shorter critical path of the CF module allows it to be run at 40 MHz (nearly the maximum operating frequency of the CF module). Thus the serial EBCOT tier-1 process implementation is a true multirate hardware system. The size of the FIFO is an important consideration when optimizing the throughput of the entire system. Optimum throughput is obtained when the rate at which the symbol-context pairs are produced by the CF module matches the rate at which they are consumed by the AE module. When the CF produces symbol-context pairs faster, the FIFO becomes full and the CF module must stall

processing until there is space in the FIFO. This is called write stalling. When the AE consumes symbol-context pairs faster, the FIFO becomes empty, and the AE module must stall processing until new symbol-context pairs become available. This is called read stalling.

In the serial architecture, a FIFO of infinite size would result in zero write stalling and thus maximum throughput for the system. But in real implementations, since the AE module is slower than the CF module, any FIFO of practical sizes will eventually fill up resulting in write stalling. The optimal size of the FIFO was determined by computing the execution time for encoding an entire image for several different FIFO sizes. Initially, as the FIFO size increased, the execution time decreased drmatically. But as the FIFO size increased beyond 4K words, we obtained negligible reduction in execution time. Thus 4K words is regarded as the optimal FIFO size for this implementation.

8.3 The Single Symbol CF Split AE Implementation

A new FPGA hardware architecture based on the SAE process for EBCOT tier-1 is shown in Figure 8.2. As before, since the CF tasks are dependent (as seen in Figure 6.2) and since the CF module is faster than the AE module, all the CF tasks have been assigned to one CF module. This CF module produces one symbol-context pair per clock cycle and hence is called "single symbol CF". Also since all the CSA tasks have to be performed one after the other, they are all assigned to one CSA module. Since the AE tasks are independent and may potentially be run in parallel we have four dedicated AE modules to perform all AE processing. There are four sets of FIFOs (called AE-FIFO) between the CF and AE modules that store the symbol-context pairs, one FIFO for each AE module. Similarly there is one FIFO (called CSA-FIFO) between the CF and CSA modules that stores symbol-context pairs. In addition there are four sets of FIFOs (called CS-FIFO) between the CSA and AE modules that stores context states at the end of coding passes as computed by the CSA module.

The CF module processes the code-block coefficients and generates symbol-context pairs. These symbol-context pairs are sent to the CSA-FIFO as well as one of the AE-FIFOs—which AE-FIFO is determined by a round-robin assignment. The symbol-context pairs from pass 1 go to AE-FIFO1, those from pass 2 go to AE-FIFO2 and so on. The symbol-context pairs from pass 5 are stored

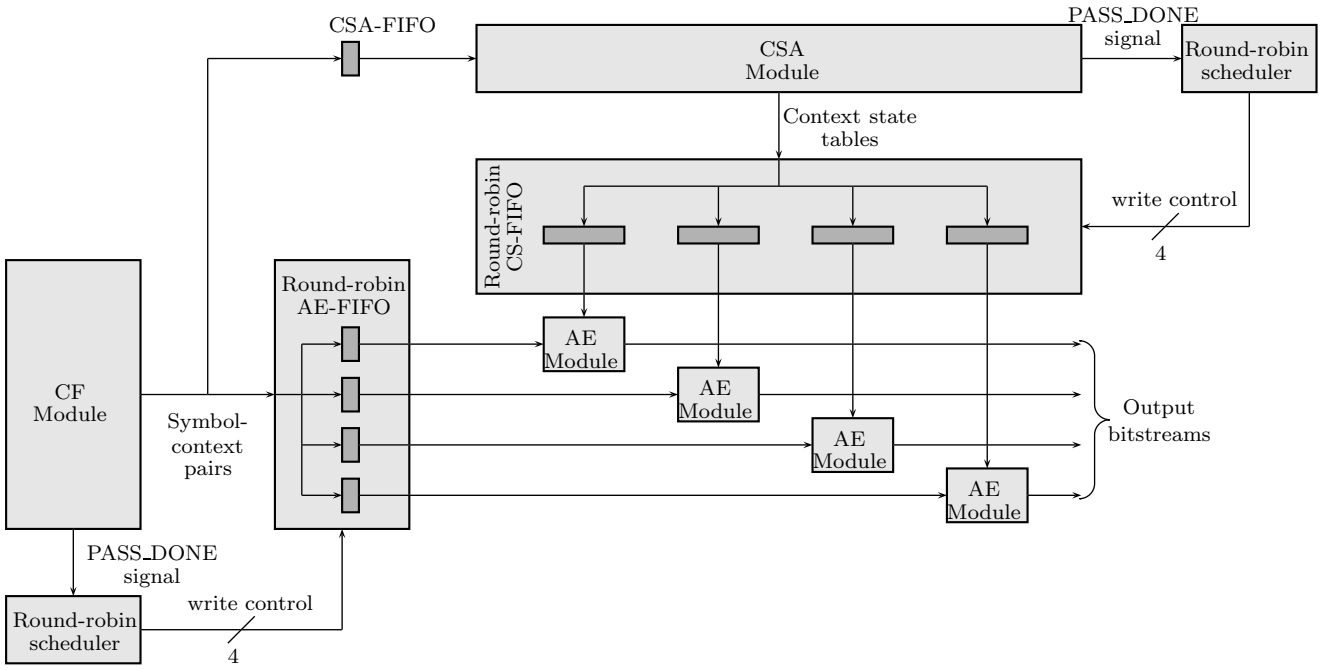


Figure 8.2: *Block diagram of the proposed SAE implementation of EBCOT tier-1 on FPGA.*

back in AE-FIFO1. This assignment ensures that the load of AE processing is equally distributed between the four AE modules. The CSA module reads the symbol-context pairs from the CSA-FIFO and computes the final context state tables for each pass. These final tables are then stored into one of the CS-FIFO (also based on a round-robin assignment). Each AE module reads the symbol-context pairs from its AE-FIFO and the latest context state table from it CS-FIFO and creates the encoded bitstream.

The natural choice in a parallel AE implementation would have been to use one dedicated AE module for each of the three coding passes. The reason for having four independent AE modules with round-robin assignment is the following. Test runs using several complete images show that within each bitplane, one of the coding passes is typically predominant and generates a large number of symbol-context pairs while the other two coding passes produce only a small number of symbol-context pairs. For example, in the uppermost bitplanes the cleanup pass (pass 2) dominates, in the middle bitplanes the significance propagation pass (pass 0) dominates and in the lower bitplanes the magnitude refinement pass (pass 1) dominates. Three dedicated AE modules would result in one of the AE modules being loaded more than the other two. Consequently the corresponding

AE-FIFO would fill up quickly and result in write stalling overall throughput degradation. Four non-dedicated AE modules with a round robin assignment avoids write stalling and improves the overall throughput.

After synthesizing the circuit and placing and routing it on the FPGA board, the AE module again had the longest critical path and had to run at the slowest clock speed (10 MHz). As before, the CF module was run at four times the clock speed of the AE module (40 MHz). The CSA module had a maximum clock speed that was intermediate between CF and AE. The CSA module was run at twice the speed (20 MHz) of the AE module. The optimal sizes of the FIFO were determined using the same empirical method that was used for the serial architecture.

8.4 The Multiple Symbol CF Split AE Implementation

The single symbol CF SAE implementation suffers from significant read stalling in the CSA module. The read stalling data from 16 of the standard test images is shown in Table 8.1. The table shows that on the average 25.22% of CSA clock cycles are wasted waiting for new symbol-context pairs to be generated by the CF module. This is in spite of the fact that the CSA is already running at twice the clock speed as the CF module. This significant read stalling in the CSA adversely affects the overall throughput of the system.

Gupta et. al's architecture [22] that processes an entire stripe column at once and generates all the symbol-context pairs for that stripe column in one clock cycle was implemented to make the CF module produce the symbol-context pairs faster. This necessitated changing the FIFOs (both CSA and AE) to allow for multiple symbol-context pairs to be written in a single clock cycle. FIFO design requires that the number of words (symbol-context pairs) to be written in each write operation must be determined before hand. An analysis was performed to determine this optimal number. The *Parrots* image was encoded using EBCOT and the number of symbol-context pairs that were generated from each stripe column for all coding passes for all bitplanes for all code-blocks was counted. The empirical probability density function and cumulative density function computed from the analysis is shown in Figure 8.3. Here the estimated probability of a stripe column generating different numbers of symbol-context pairs is depicted. The smallest number of symbol-context pairs that may be generated is 0 and the largest is 10. The largest number is

Table 8.1: *CF throughput and CSA read stalling for single symbol CF SAE implementation.*

Images	CF Throughput	Total CSA Clocks	CSA Read Stalls	% CSA Read Stalls
Aerial2	14260853	57645678	15725169	27.28
Bike	13716717	141073682	42454330	30.09
Bike3	13196415	42253221	13742187	32.52
Cafe	13591509	159039334	48814777	30.69
Cats	14838178	117322921	28967593	24.69
F21_200	12563563	145695977	2085408	1.43
Finger	14702318	3524606	894116	25.37
Gold	14027776	8955301	2550756	28.48
Hotel	13618206	9355091	2850613	30.47
Ingres1	14410961	116460690	31214922	26.80
Ingres2	14287347	123378768	33733839	27.34
Ingres3	14465377	111927711	29770029	26.60
Ingres4	14329847	115694712	31472822	27.20
Parrots	13461957	9210229	2904537	31.54
Tools	13424752	42346686	13291537	31.39
Woman	14223347	131652805	36411582	27.66
Average	**13944945**	**1335537412**	**336884217**	**25.22**

encountered during run mode in the cleanup pass—a run interruption is experienced for the first coefficient and each of the succeeding three coefficients in the stripe column are significant.

Figure 8.3 indicates that for almost all the cases (as much as 90%), the stripe columns generate 4 or fewer symbol-context pairs. So the FIFOs were designed to populate as many as four symbol-context pairs in one write cycle. For most stripe columns, one clock cycle should suffice to write all generated symbol-context pairs into the FIFO. Very few stripe columns would need two write cycles and almost no stripe columns would need three write cycles.

With these changes (after synthesis, placement and routing), the AE module was found to run at a maximum speed of 9.25 MHz. Also, the length of the critical path in the new multiple symbol CF module doubled from the previous single symbol CF implementation. Thus the CF module ran at twice the speed of the AE module, i.e. 18.5 MHz. Despite halving its clock rate, the CF module now produced symbol-context pairs faster and the CSA module was unable to keepup—the result was significant write stalling on the CF side of the CSA-FIFO. To counter this, the CSA module was pipelined to reduce the largest delay within the module, and make it run at a much higher clock rate. The pipelined CSA module ran at four times the speed of the AE module, i.e 37 MHz.

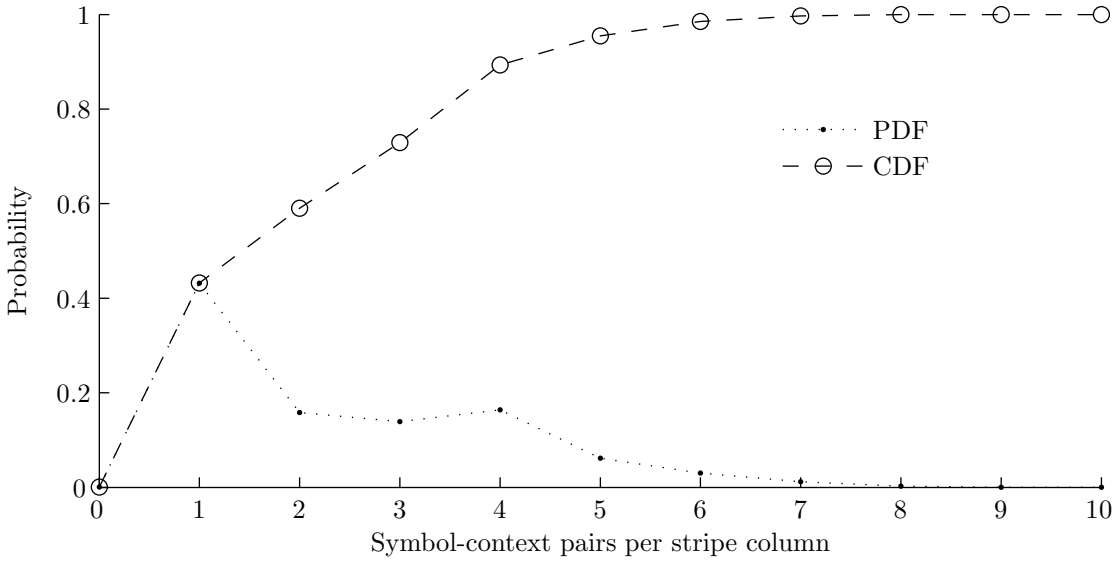


Figure 8.3: *Probability density and cumulative density of the number of symbol-context pairs generated per stripe column.*

This new implementation minimized the read and write stalling in the FIFOs and also provided the four AEs with symbol-context pairs at a much faster rate. The CSA read stalling data for the multiple symbol CF SAE implementation is shown in Table 8.2.

The table shows that the overall read stalling in the CSA has been reduced to 0.03% of the total CSA clock cycles. In addition the overall throughput of the CF module (expressed in symbol-context pairs generated per second) increased from 13944945 to 22089016, or 58.4%. This means that the four AE modules are able to acquire symbol-context pairs 1.58 times faster than with the single symbol CF module. This results in increased parallel processing by the AE modules and therefore increased overall throughput for the SAE process.

8.5 Results

A summary of the performance results from the three implementations is shown in Table 8.3. The three implementations are represented by the following acronyms: (1) "Ser" for the single symbol CF monolithic AE serial implementation, (2) "SS" for the single symbol CF SAE implementation, and (3) "MS" for the multiple symbol CF SAE implementation. The comparisons are in terms of

Table 8.2: *CF throughput and CSA read stalling for multiple symbol CF SAE implementation.*

Images	CF Throughput	Total CSA Clocks	CSA Read Stalls	% CSA Read Stalls
Aerial2	23387494	32514280	14538	0.04
Bike	21566713	82996286	8608	0.01
Bike3	20903108	24674627	19876	0.08
Cafe	21617464	92493965	11921	0.01
Cats	23537244	68415615	12280	0.02
F21_200	19342726	87536397	23207	0.03
Finger	24361691	1967589	14978	0.76
Gold	22294356	5212195	16484	0.32
Hotel	21500519	5481070	13460	0.25
Ingres1	22558862	68818082	10218	0.01
Ingres2	22487944	72508578	13078	0.02
Ingres3	22547560	66422541	14927	0.02
Ingres4	22387439	68501191	12004	0.02
Parrots	21033004	5452853	10240	0.19
Tools	21385681	24589420	17003	0.07
Woman	22512453	76940656	14206	0.02
Average	**22089016**	**784525345**	**227028**	**0.03**

three performance measures: (1) number of clock cycles consumed, (2) execution time in seconds, and (3) throughput in pixels per second. The reason for including both the clock cycles as well as the execution time is that the different implementations run on different clock frequencies; execution time normalizes this difference. Throughput normalizes the difference in image sizes.

Throughputs of grayscale and color images must be considered separately since grayscale images have one wavelet coefficient per pixel and color images have three wavelet coefficients per pixel. Consequently the throughput of grayscale images (Aerial2 and F21_200 in Table 8.3) are substantially larger than that of color images.

The improvements obtained using the two new SAE implementations are expressed in two different forms. First the percentage improvement obtained over the serial implementation is computed. In order to compute the percentage improvement in execution time for the multiple symbol CF SAE implementation, the following formula is used.

$$P_{\mathrm{MS}}^{\mathrm{Time}} = \frac{T_{\mathrm{Ser}} - T_{\mathrm{MS}}}{T_{\mathrm{Ser}}} \times 100 \tag{8.1}$$

Table 8.3: *Performance comparison of the three FPGA implementations of EBCOT tire-1.*

Images	Clock Cycles			Time (sec)			Throughput (pixels/sec)			Percentage Improvement						Speedup					
										Cycles		Time		Thrpt		Cycles		Time		Thrpt	
	Ser	SS	MS	Ser	SS	MS	Ser	SS	MS	SS	MS	SS	MS	SS	MS	SS	MS	SS	MS	SS	MS
Aerial2	41138355	28825239	16257653	4.11	2.88	1.76	1019560	1455080	2386403	29.93	60.48	29.93	57.28	42.72	134.06	1.43	2.53	1.43	2.34	1.43	2.34
Bike	97052539	70539579	41500000	9.71	7.05	4.49	432168	594603	934875	27.32	57.24	27.32	53.77	37.59	116.32	1.38	2.34	1.38	2.16	1.38	2.16
Bike3	27959215	21127791	12338000	2.80	2.11	1.33	375038	496302	786135	24.43	55.87	24.43	52.29	32.33	109.61	1.32	2.27	1.32	2.10	1.32	2.10
Cafe	108244563	79522549	46248519	10.82	7.95	5.00	387484	527436	838888	26.53	57.27	26.53	53.81	36.12	116.50	1.36	2.34	1.36	2.16	1.36	2.16
Cats	87138753	58663588	34209344	8.71	5.87	3.70	481336	714976	1134114	32.68	60.74	32.68	57.56	48.54	135.62	1.49	2.55	1.49	2.36	1.49	2.36
F21_200	92219518	72849881	43769735	9.22	7.28	4.73	454817	575746	886396	21.00	52.54	21.00	48.69	26.59	94.89	1.27	2.11	1.27	1.95	1.27	1.95
Finger	2584713	1764216	983830	0.26	0.18	0.11	1014209	1485895	2464686	31.74	61.94	31.74	58.85	46.51	143.02	1.47	2.63	1.47	2.43	1.47	2.43
Gold	6294280	4478846	2606203	0.63	0.45	0.28	416480	585294	930408	28.84	58.59	28.84	55.24	40.53	123.40	1.41	2.42	1.41	2.23	1.41	2.23
Hotel	6386438	4678633	2740600	0.64	0.47	0.30	410470	560300	884781	26.74	57.09	26.74	53.61	36.50	115.55	1.37	2.33	1.37	2.16	1.37	2.16
Ingres1	84066476	58232555	34411000	8.41	5.82	3.72	498927	720268	1127468	30.73	59.07	30.73	55.75	44.36	125.98	1.44	2.44	1.44	2.26	1.44	2.26
Ingres2	88296978	61691449	36256000	8.83	6.17	3.92	475022	679884	1070094	30.13	58.94	30.13	55.61	43.13	125.27	1.43	2.44	1.43	2.25	1.43	2.25
Ingres3	81104114	55966091	33212807	8.11	5.60	3.59	517151	749437	1168143	30.99	59.05	30.99	55.73	44.92	125.88	1.45	2.44	1.45	2.26	1.45	2.26
Ingres4	83053334	57849496	34252132	8.31	5.78	3.70	505013	725037	1132698	30.35	58.76	30.35	55.41	43.57	124.29	1.44	2.42	1.44	2.24	1.44	2.24
Parrots	6241969	4606486	2726532	0.62	0.46	0.29	419970	569076	889347	26.20	56.32	26.20	52.78	35.50	111.76	1.36	2.29	1.36	2.12	1.36	2.12
Tools	28490476	21175285	12295099	2.85	2.12	1.33	368044	495189	788878	25.68	56.84	25.68	53.35	34.55	114.34	1.35	2.32	1.35	2.14	1.35	2.14
Woman	93716604	65828973	38472000	9.37	6.58	4.16	447552	637152	1008456	29.76	58.95	29.76	55.62	42.36	125.33	1.42	2.44	1.42	2.25	1.42	2.25
Average							**482755**	**675183**	**1063199**	**28.50**	**58.00**	**28.50**	**54.59**	**39.86**	**120.24**	**1.40**	**2.38**	**1.40**	**2.20**	**1.40**	**2.20**

The improvement is also expressed in terms of the speedup factor. In order to compute the speedup in execution time for the multiple symbol CF SAE implementation, the following formula is used.

$$S_{\mathrm{MS}}^{\mathrm{Time}} = \frac{T_{\mathrm{Ser}}}{T_{\mathrm{MS}}} \tag{8.2}$$

On average, for clock cycle efficiency, there is a 28.5% improvement of the single symbol CF SAE over the serial implementation and a 58% improvement of the multiple symbol CF SAE over the serial implementation. When we look at execution time, the improvement for the single symbol CF SAE is the same 28.5%. This is because there is no difference in AE clock frequency between the serial and the single symbol SAE implementations. But the execution time improvement for multiple symbol SAE is 54.59%—slightly lower than the improvement for clock cycle efficiency since it uses a slower AE clock frequency. This illustrates why it is important to compare performance in real execution times instead of just clock cycle efficiency.

Throughput comparisons reveal that, on average, the single symbol CF SAE implementation improves throughput by 39.86% while the multiple symbol CF SAE implementation improves it by 120.24%. These are equivalent to speedup factors of 1.4 and 2.2 respectively.

Figure 8.4 compares execution times of tasks in the serial and SAE architectures as they process coding passes 8 to 14 of code-block 30 of the *Parrots* image. The figure shows start and end times for the AE tasks in the serial implementation and the CSA and AE tasks in the single symbol CF SAE implementation. The light and dark blue bars denote the time taken by the CSA module to process the CSA task in each coding pass; the light blue portion denotes time spent processing and the dark blue portion denotes time spent idle due to read stalling (waiting for the CF module to produce symbol-context pairs). The CSA module processes the CSA tasks serially and has to wait for the completion of a CSA task in one pass before it can start processing the CSA task for the successive pass. The green bars indicate the execution time for the AE tasks in the SAE architecture. Because there is no causal dependency between these tasks and the architecture includes four AE modules, up to four AE tasks may be in progress at any one time. The only constraint on the start time of an AE task is that the CSA task from the previous pass must be complete. As seen in the figure, this allows a substantial amount of AE processing to be performed in parallel, thus allowing the SAE implementation to finish faster. The figure also shows that the single symbol CF SAE implementation is hampered by significant read stalling in the CSA (dark

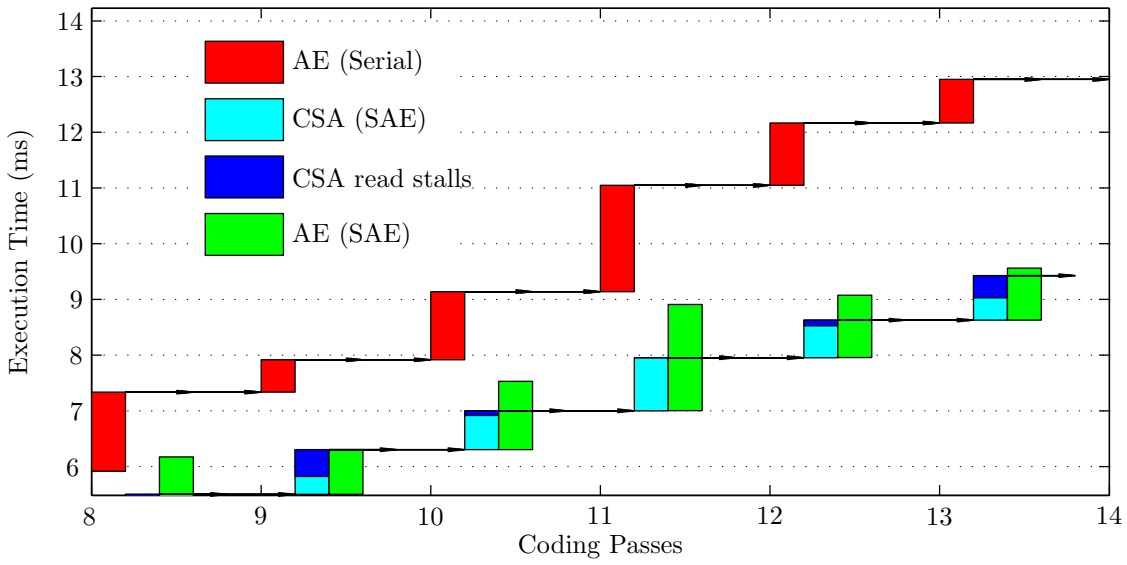


Figure 8.4: *Comparison of execution times of serial AE, CSA, and SAE for some coding passes.*

blue bars), which has been rectified in the multiple symbol CF SAE implementation. Figure 8.5 shows the same type of graphs for all passes of code-block 30; it reveals how clock cycle savings accumulate as passes are processed.

The hardware implementations of the SAE process achieve higher throughput due to a reason slightly different from the estimates or the software implementation. In the estimates as well as the software implementation, we extract the CSA task out of the AE task because the CSA task consumes less time. In the case of the hardware implementations, the simpler CSA module runs on a faster clock and is therefore able to process the symbol-context pairs much faster. Because the CSA runs at twice or four times the speed of the AE module, CSA tasks are completed well ahead of the AE tasks. This results in a substantial amount of potential parallel processing for AE tasks. This scale of parallelism is not achievable using a software implementation on reasonable computing platforms.

The serial architecture presented in [7] is widely accepted and used for comparison; however, it fails to properly account for significance propagation [22]. Other work concentrates on the percentage clock cycles saved in the CF module [23–25, 27]. However, clock cycles do not tell the whole story because they do not reflect the design's maximum operating frequency. Moreover, designs of only the CF module do not account for the mismatch between the rate of symbol-context

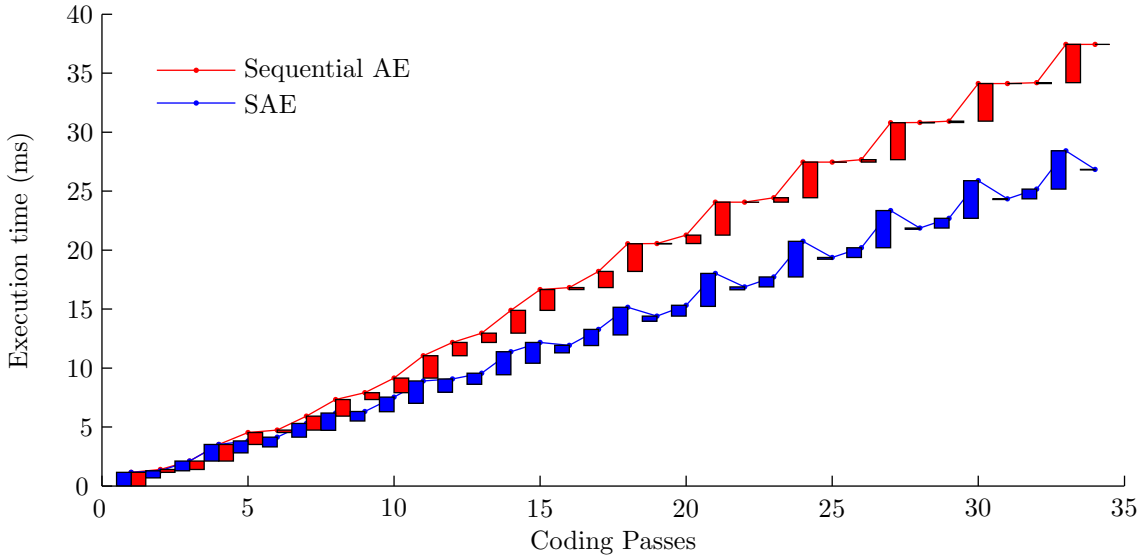


Figure 8.5: *Comparison of execution times of serial AE and SAE for all coding passes of code-block 30 of Parrots image.*

generation by the CF module and the rate of symbol-context consumption by the CSA and AE modules. The resulting read and write stalling can significantly degrade performance of a complete EBCOT tier-1 implementation. By contrast, our comprehensive approach includes the interaction between the CF and AE modules and estimates the overall throughput of the encoder.

Furthermore, previous pass-parallel and sample-parallel architectures sacrifice coding efficiency in order to achieve the speedup. Our work is unique in that we achieve improved overall throughput while preserving most of JPEG2000's coding efficiency. Our average system throughput improvement of 120.24% compares favorably with the 33% improvement estimated in [22] using a multiple symbol encoding AE module. We achieve this considerable improvement by sacrificing a small amount of coding efficiency and by using some extra hardware.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

JPEG2000 is the state of the art image coding standard that outperforms the older JPEG standard for all images across all bit rates. In addition to the improvement in compressed image quality JPEG2000 possesses unique features like embedded encoding that enables resolution and quality scalability of compressed images. These features are critical in mobile, wireless imaging systems where speed, power and size/weight constraints are challenging. Field programmable gate arrays (FPGA) are ideal candidates for such applications. This dissertation presented new perceptual weights that improve the subjective quality of coded images as well as new hardware and software architectures that achieve high throughput while maintaining coding efficiency.

New perceptual weights that improved the subjective quality of compressed images at low bit rates were designed by combining information from both the human contrast sensitivity function and the standard deviation within a subband or code-block. They were compared to the contrast sensitivity function (CSF) perceptual weights that are part of the JPEG2000 standard. The new perceptual weights based on the CSF and subband standard deviation, $W_{C-SSD}(s)$, performed better than the JPEG2000 CSF weights, $W_C(s)$, for high frequency images, but they performed worse for low frequency images. Weights based solely on subband standard deviation, $W_{SSD}(s)$, performed worse than the JPEG2000 CSF weights for all images at all compression ratios. This suggests that the underlying human contrast sensitivity function plays a more important role in perceptual quality than the amount of activity in a subband. JPEG2000's tiling feature can be used to divide an image into high, medium and low frequency tiles. Our results suggest that the

low frequency tiles be compressed using $W_C(s)$ and the high frequency tiles be compressed using $W_{C-SSD}(s)$. For the medium frequency tiles JPEG2000's visual progressive weights feature could be used to apply $W_{C-SSD}(s)$ at low to moderate compression ratios and then switch to $W_C(s)$ at higher compression ratios.

This dissertation also proposed a new parallel split arithmetic encoder (SAE) process model for embedded block coding with optimal truncation (EBCOT) tier-1 where the context state adaptation and the bitstream generation are separated into two separate tasks called context state adaptation (CSA) and arithmetic encoding (AE). The proposed SAE process is distinguished from previous research in that it achieves throughput improvement without sacrificing as much coding efficiency. Three methods to evaluate the SAE process were designed and implemented: (1) clock cycle consumption estimation, (2) multithreaded software implementation, and (3) FPGA based multirate hardware implementation. The SAE process is further distinguished from past efforts in two ways: SAE achieves most of its speedup through the AE task, not the context formation (CF) task; and, SAE is a comprehensive process that incorporates both the CF and the AE tasks in EBCOT tier-1. Consequently, SAE implementations could be further enhanced by taking advantage of faster CF and AE designs.

The resulting savings were substantiated. The "estimation" method showed that for a representative image the new process model saved as much as 8% of the total clock cycles over the serial architecture. However software and hardware implementations achieved as much as 50% and 120% throughput improvement over the conventional serial implementation. All three SAE evaluation methods (estimation, multithreading, and FPGA) preserved coding efficiency and improved throughput. Clock cycle estimation provided quick insights into the performance of the new SAE process, but significantly underestimated real performance. Real implementations of the SAE process relied on CSA executing faster than AE. In multithreaded software the SAE process mapped to less computation in the CSA thread and in hardware the SAE process mapped to smaller delay in the CSA module. Consequently, both implementations improved throughput. The hardware implementation improved throughput more than the software implementation, as expected since the hardware implementation results in greater savings in CSA processing. The substantial difference in results for the three methods used to implement and evaluate the SAE process underscores the importance of evaluating a proposed process (algorithm) from several perspectives.

9.2 Future Work

Future work on the SAE process has the potential to investigate two new opportunities to improve the overall throughput further. First the SAE process model could be applied while using a true pass-parallel CF implementation as was done for the multithreaded software implementation on *Inferno2*. In particular the CF module designed by Chiang [23] and that designed by Gangadhar [24] has the potential to increase the throughput of the CF task. Using a CF with a higher throughput, along with a larger first-in-first-out buffer, or FIFO (to mitigate write stalling on the CF side) has the potential to improve the overall throughput more than current implementations.

A second opportunity for improvement comes from pipelining the AE module. The current implementation of the AE module has minimum pipelining and therefore runs on a slow clock. By pipelining the AE module, the critical path inside the module is decreased, thus allowing it to be run on faster clocks. This has the potential to achieve equal or greater throughput compared to current implementations using a smaller number of instances of the AE module. For example two instances of the faster, pipelined AE modules (fed using a round-robin assignment) may be able to achieve the same or greater overall throughput than four instances of the slower AE modules. The hardware cost of each AE module may have increased; however this is more than compensated for by the fact that fewer instances of the AE modules need to be used.

Bibliography

[1] M. Nadenau and J. Reichel, "Opponent color, human vision and wavelets for image compression," in *Proc. of the Seventh Color Imaging Conference*, 1999, pp. 237–42.

[2] D. S. Taubman and M. W. Marcellin, *JPEG2000: Image Compression Fundamentals, Standards and Practice*. Norwell, MA: Kluwer Academic Publishers, 2002.

[3] A. Skodras, C. Christopoulos, and T. Ebrahimi, "The JPEG2000 still image compression standard," *IEEE Signal Processing Mag.*, vol. 18, no. 5, pp. 36–58, September 2001.

[4] J. M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," *IEEE Trans. Signal Processing*, vol. 41, no. 12, pp. 3445–62, 1993.

[5] A. Said and W. Pearlman, "A new, fast, and efficient image codec based on set partitioning in hierarchical trees," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 6, no. 3, pp. 243–50, June 1996.

[6] A. B. Watson, "Efficiency of a model human image code," *Journal of the Optical Society of America*, vol. 4, no. 12, pp. 2401–2417, December 1987.

[7] C. Lian, K. Chen, H. Chen, and L. Chen, "Analysis and architecture design of block-coding engine for EBCOT in JPEG2000," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 3, pp. 219–30, March 2003.

[8] Y. Hayashi, H. Tsutsui, T. Masuzaki, T. Humi, Y. Onoye, and Y. Nakamura, "Design framework for JPEG2000 encoding system architecture," in *Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 2, May 2003, pp. 740–743.

[9] M. Marcellin, M. Lepley, A. Bilgin, T. Flohr, T. Chinen, and J. Kasner, "An overview of quantization in JPEG2000," *Signal Processing: Image Communications*, vol. 17, no. 1, pp. 73–84, 2002.

[10] J. L. Mannos and D. J. Sakrison, "The effects of a visual fidelity criterion on the encoding of images," *IEEE Trans. Inform. Theory*, vol. 20, no. 4, pp. 525–36, July 1974.

[11] K. Mullen, "The contrast sensitivity of human colour vision to red-green and blue-yellow chromatic gratings," *Journal of Physiology*, vol. 359, pp. 381–400, 1985.

[12] *ITU T.800: JPEG2000 image coding system Part 1*, ITU Std., July 2002.

[13] A. B. Watson, G. Y. Yang, J. A. Solomon, and J. Villasenor, "Visibility of wavelet quantization noise," *IEEE Trans. Image Processing*, vol. 6, no. 8, pp. 1164–75, August 1997.

[14] S. S. Hemami and M. G. Ramos, "Wavelet coefficient quantization to produce equivalent visual distortions in complex stimuli," in *Proc. of SPIE Human Vision and Electronic Imaging V*, vol. 3959, January 2000, pp. 200–10.

[15] A. Beegan, L. R. Iyer, A. E. Bell, V. R. Maher, and M. A. Ross, "Design and evaluation of perceptual masks for wavelet image compression," in *Proc. of the Tenth IEEE Digital Signal Processing Workshop*, October 2002, pp. 88–93.

[16] S. S. Hemami and M. G. Ramos, "Perceptual quantization for wavelet-based image coding," in *Proc. IEEE Int. Conf. Image Proc.*, vol. 1, September 2000, pp. 645–48.

[17] K. Varma and A. Bell, "Improving JPEG2000's perceptual performance with weights based on both contrast sensitivity and standard deviation." in *Proc. IEEE International Conference Acoustics Speech and Signal Processing (ICASSP)*, May 2003.

[18] *ITU T.24: Standardized digitized image set*, ITU Std., June 1998.

[19] *ITU-R BT.500-11: Methodology for the subjective assessment of the quality of television pictures*, ITU Std., 2002.

[20] A. P. Bradley, "A wavelet visible difference predictor," *IEEE Trans. Image Processing*, vol. 8, no. 5, pp. 717–30, May 1999.

[21] K. Andra, C. Chakrabarti, and T. Acharya, "A high performance JPEG2000 architecture," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 3, pp. 209–18, March 2003.

[22] A. K. Gupta, D. Taubman, and S. Nooshabadi, "High speed VLSI architecture for bit plane encoder of JPEG2000," in *The 47th Midwest Symposium on Circuits and Systems (MWSCAS'04)*, vol. 2, July 2004, pp. 233–236.

[23] J. Chiang, Y. Lin, and C. Hsieh, "Efficient pass-parallel architecture for EBCOT in JPEG2000," in *Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 1, May 2002, pp. 773–776.

[24] M. Gangadhar and D. Bhatia, "FPGA based EBCOT architecture for JPEG2000," in *Proc. of the IEEE International Conference on Field-Programmable Technology*, December 2003, pp. 228–233.

[25] J. Chiang, C. Chang, Y. Lin, C. Hsieh, and C. Hsia, "High-speed EBCOT with dual context-modeling coding architecture for JPEG2000," in *Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 3, May 2004, pp. 865–68.

[26] H. Fang, T. Wang, C. Lian, T. Chang, and L. Chen, "High speed memory efficient EBCOT architecture for JPEG2000," in *Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 2, May 2003, pp. 736–39.

[27] Z. Xing, Y. Ye, Q. Xing, W. Tuo, and S. Hai-Bin, "A cycle-efficient sample-parallel EBCOT architecture for JPEG2000 encoder," in *Proc. of the International Symposium on Intelligent Multimedia, Video and Speech Processing*, Oct 2004, pp. 386–89.

[28] G. Pastuszak, "A high-performance architecture for embedded block coding in JPEG 2000," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 15, no. 9, pp. 1182–91, Sep 2005.

[29] K. Varma and A. E. Bell, "DSP Tips and Tricks: JPEG2000–Choices and tradeoffs for encoders." *IEEE Signal Processing Mag.*, vol. 21, no. 6, pp. 70–75, November 2004.

[30] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *22nd Annual International Symposium on Computer Architecture*, June 1995.

[31] T. Moseley, J. Kihm, D. Connors, and D. Grunwald, "Methods for modeling resource contention on simultaneous multithreading processors," in *International Conference on Computer Design*, Oct 2005.

[32] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

Vita

Krishnaraj M. Varma received his Bachelor of Technology degree in Applied Electronics and Instrumentation Engineering in October 1997 from the College of Engineering, Trivandrum, University of Kerala, India. After receiving his degree he worked for Tata Consultancy Services (TCS) as Assistant Systems Engineer. After a brief stint at the TCS center in Bombay, India, he was sent on assignment to the National Association of Securities Dealers (NASD) located in Rockville, MD, to take over the charge of production support for crucial market regulation software systems.

In August 2000, he left TCS to start his graduate study at Virginia Tech. At the Digital Signal Processing Research Laboratory (DSPRL) at Virginia Tech, he worked on developing new algorithms to determine the direction of arrival of speech signals in reverberant environments using three dimensional microphone arrays.

After completing the MS degree in Electrical Engineering in December 2002, he continued his study at Virginia Tech, pursuing a Ph.D in Electrical Engineering at the Digital Signal Processing and Communications Laboratory (DSPCL). There he worked on improving the performance of the JPEG2000 image compression standard by designing new perceptual weights to improve the subjective quality of compressed images and also designed a new process model for JPEG2000's EBCOT algorithm that was shown to improve encoding throughput in both software and hardware. He is expected to complete his doctoral degree requirements in May 2006. His research interests are in the areas of digital signal processing, image processing, coding and communications.