Analysis and Architecture Design of EBCOT for JPEG-2000

Kuan-Fu Chen, Chung-Jr Lian and Hong-Hui Chen, Liang-Gee Chen

Department of Electrical Engineering, National Taiwan University
1, Sec.4, Roosevelt Road, Taipei 106, Taiwan

ABSTRACT

This paper presents detailed analysis and efficient architecture design of Embedded Block Coding with Optimized Truncation (EBCOT) for JPEG-2000. Detailed profile of the context formation process in EBCOT is analyzed to get an insight into the characteristics of the operation. Column-based operation is adopted to enable higher process speed. Two speed-up methods, referred to as Pixel Skipping (PS), and Group-Of-Column Skipping (GOCS), are proposed. It is shown that over 60% of processing time can be reduced by exploiting the two methods. A column-based architecture using these combined speed-up ideas is then proposed.

1. INTRODUCTION

JPEG-2000 [1][2][3] is an emerging standard for still image compression. It not only has better compression performance than the existing JPEG [7] standard but also provides new features not available in JPEG. JPEG-2000 is composed of two parts: Discrete Wavelet Transform (DWT) and EBCOT, as shown in Fig. 1. Wavelet transform is a subband transform, and it transfers images from spatial domain to frequency domain. The generated coefficients are then scalar quantized and entropy coded by EBCOT.

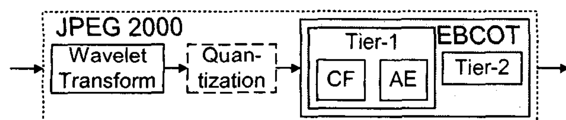

Fig. 1. JPEG-2000 block diagram

EBCOT coding algorithm is proposed by David Taubman [4][5][6]. It contains two parts, tier-1 and tier-2, as shown in Fig. 1. Tier-1 is a complete context-based arithmetic encoder. It divides quantized subbands composed of wavelet coefficients into code blocks, and encodes code blocks into independent embedded bit-streams. Tier-2 orders the bit-streams into final JPEG-2000 format bit-stream according to rate-distortion values calculated by tier-1 and features specified by user. As we can see in the run time profile of JPEG-2000 in Table I, the great part of computation load is on tier-1 coder.

In this paper, we aim our focus and analysis on Tier-1 coder and propose architecture for this part. Our analysis of EBCOT is presented in Section 2, which details profiles and analysis of the EBCOT Tier-1 coder. Two speed-up methods are proposed according to the analysis results. The proposed architecture exploiting these speed-up ideas is described in Section 3. Experiment results on different images are given in Section 4. A short conclusion is given in Section 5.

Table I. Run time profile for JPEG-2000 (Image 1792x1200, 5 level wavelet decomposition, 1 layer, profile at PIII-733 128M RAM, Visual C++ 6.0 and Windows ME)

Operation	Single Component		3 Components(RGB)	
	Lossless	Lossy	Lossless	Lossy
Intercomponent transform			**0.91**	**14.12**
Intracomponent transform	**10.81**	**26.38**	**11.9**	**23.97**
quantization		**6.42**		**5.04**
EBCOT Tier 1	**71.63**	**52.26**	**69.29**	**43.85**
pass 1	*14.89*	*14.82*	*13.9*	*12.39*
pass 2	*10.85*	*7*	*10.94*	*5.63*
pass 3	*26.14*	*16.09*	*25.12*	*13.77*
arithmetic encoder	*19.75*	*14.35*	*19.33*	*12.06*
EBCOT Tier 2	**17.56**	**14.95**	**17.9**	**13.01**
layer formation	*10*	*9.52*	*9.94*	*7.95*
marker insertion	*7.56*	*5.43*	*7.96*	*5.06*

2. EBCOT ALGORITHM AND ANALYSIS

Tier-1 of EBCOT utilizes context-based arithmetic coding method to encode each code block into independent embedded bit-stream. Tier-1 coder can be viewed as two parts: Context Formation (CF) and Arithmetic Encoder (AE). CF scans all pixels in code block in a specific order, and generates corresponding contexts for each bit. AE encodes the code block data according to their contexts. EBCOT encodes the quantized wavelet coefficients bitplane by bitplane from MSB to LSB. Every 4 rows in a bitplane are called a "stripe," and each pass in every bitplane scans in order stripe by stripe. Then in every stripe, data are scanned column by column. Every column is composed of 4 bits. So the scanning hierarchy of a code block is bitplane, pass, stripe, column, bit, as shown in Fig. 2.

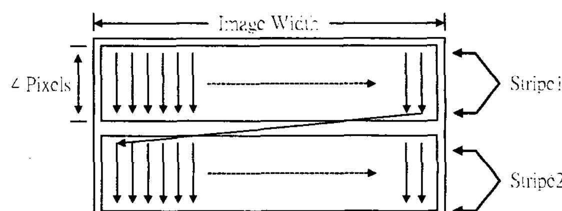

Fig. 2. Scanning order of context formation in every pass

Contexts for all bits are generated according to their neighbors using four coding methods. Before CF, the quantized wavelet coefficients are separated into sign and magnitude (in 1's compliment). A pixel is called "significant" after the first '1' bit is met while encoding magnitude part from MSB to LSB, and "insignificant" before the first '1' bit appears. The context of each bit is determined by significant situations of its neighbors. There are four coding methods to generate context for each bit in a code block: Zero Coding, Run-Length Coding, Sign Coding, and Magnitude Refinement.

0-7803-6685-9/01/$10.00©2001 IEEE

Every bitplane is encoded using 3 passes in turn. Each pixel in a bitplane is encoded in one of 3 passes. Pass 1 is "Significant Propagation Pass." Pixels having at least one significant neighbor are coded in this pass. Pass 2 is "Magnitude-Refinement Pass." All significant pixels are coded in this pass. Pass 3 is "Clean-Up Pass." Pixels not coded in first two passes are coded in this pass. While coding a bitplane, every pixel is checked once in all 3 passes to determine if this pixel should be coded. In Taubman's architecture, a straightforward method is used. Every single bit is checked and (or) coded in all 3 passes, which cost total 3 clocks. Coding a 64x64 code block with 8-bit precision will take 64x64x8x3 clocks. That makes tier-1 coder become bottleneck of JPEG-2000 system design.

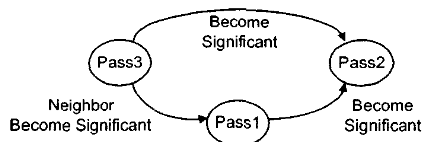

Fig. 3. Evolution Map: higher bitplanes to lower bitplanes

According to the characteristic of the context formation process, great improvement on process time can be achieved. Every pixel is insignificant at the beginning of coding first bitplane (Most Significant Bitplane), so all pixels are coded in pass 3. As we coding toward lower bitplanes, some pixels become significant. They will be coded in pass 2, and their insignificant neighbors will be coded in pass 1, as shown in Fig. 3.

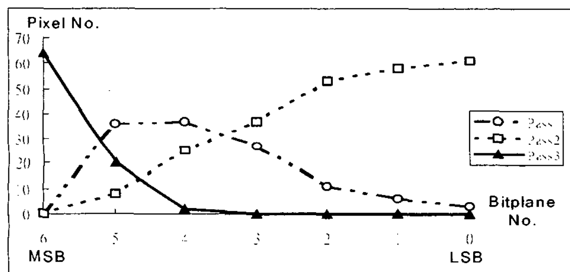

Fig. 4. Pixel distribution in 3 passes

The characteristic of the context formation process makes the distribution of pixels in 3 passes vary greatly from bitplane to bitplane. A real case is analyzed and the result is shown in Fig. 4. It is an 8x8 LL subband (lowest frequency subband) from 256x256 Lena image with 5-level wavelet decomposition. In highest bitplane (MSB), all pixels are coded in pass 3. However, in lower bitplane (near LSB), most pixels are coded in pass 2, only few pixels are coded in pass 1, and non in pass 3. The number of pixels coded in pass 1 increases at the beginning, and then decreases because more and more pixels coded in pass 1 become significant. Every pass spends the same time in the architecture in [3], but some passes do not even encode a single pixel (such as pass 3 in bitplane 0~3). We make a great improvement on process time utilize distribution variation feature, and will be discussed later.

2.1 Column-Based Operation

Higher speed and data reuse can be achieved via column-based operation. In Taubman's architecture [3], data are supplied to context formation element one pixel at a time. We speed-up context formation by processing more than one pixel every clock. In the proposed architecture, column-based operation is adopted instead of pixel-based operation. Data are supplied to context formation element one column (four pixels) at a time. There are two advantages of column-based operation: 1) pixels in a column can be checked simultaneously, so speed-up methods proposed can be applied. 2) Higher data reuse in significant and sign state variables. Memory access frequency of these state variables can be reduced.

2.2 Two Speed-Up Methods

Table II. Columns classified by number of NBC pixels contained (Baboon, 512x512, 5-level wavelet, code block 64x64)

NBC pixel no	number of column in each pass				
	Pass1	Pass2	Pass3	Sum	
0	181076	159223	258328	598627	47.85%
1	72650	47663	14437	134750	10.77%
2	60921	49313	10532	120766	9.65%
3	51098	63132	6568	120798	9.66%
4	29391	75805	170807	276003	22.06%

In every bitplane, each pixel is coded in one of 3 passes, so one column in every pass may contain 0~4 Need-to-Be-Coded (NBC) pixels. Table II shows the analysis results of column-based operation. Columns are classified in every pass according to number of NBC pixels in them. For example, there are 181076 columns contain zero NBC pixels in pass 1, 159223 columns in pass 2, and 258328 columns in pass 3. There are in total 598627 (47%) columns contain 0 NBC pixels. The percentage of columns having four NBC pixels (which means no processing time is wasted in Taubman's architecture [3]) is only 22.06%. According to Taubman's architecture, coding a column costs 4 clocks no matter how many NBC pixels are in it. The key ideas to improve the process speed are: (1)skip no-operation pixels, and (2)skip no-operation columns (columns with zero NBC pixel). These ideas are described below:

1) Pixel Skipping (PS): PS is designed for all 3 passes, which skips no-operation pixels in a column. By column-based coding, pixels in a column can be parallel checked to see if they are NBC pixels. Only NBC pixels are processed, other no-operation pixels are skipped. If there are n NBC pixels in a column($0 < n < 4$), only n clocks will be spent on coding these NBC pixels, and 4-n clocks are saved. Since most columns have less than 4 NBC pixels, this method can improve system process speed greatly, as the experimental results shown in Sec. 4.

2) Group-Of-Column Skipping (GOCS) : GOCS skips a group of no-operation columns together in pass 2 and pass 3. By using PS speed-up methods, every no-operation column will cost only 1 clock for checking. To further improve process speed, these no-operation columns should be skipped. Due to the property of wavelet transform, spatial correlation between wavelet

coefficients is strong, so no-operation columns usually group together. We choose every 8 consecutive columns as a GOC. If there is no any column in this GOC to be a NBC column, this GOC is directly skipped, which can save 31 clocks compare to Taubman's design. GOCS can only applied on pass 2 and pass 3, because of the significant propagation in pass 1. So regular coding is applied to pass 1, that is, columns are checked one by one. After coding pass 1, all NBC columns of pass 2 and pass 3 are decided. The GOCs that contain NBC columns are recorded. While encoding pass 2 and pass 3, only those GOCs contain NBC columns are processed and all no-operation GOCs are skipped.

Every single method can improve processing speed, and each can be used together or separately. The percentage of improvement is given in Section 4.

3. ARCHITECTURE

An efficient column-based context formation architecture for tier-1 coder is described in this Section. The key ideas are based on the column-based operation and two speed-up methods described above.

3.1 System

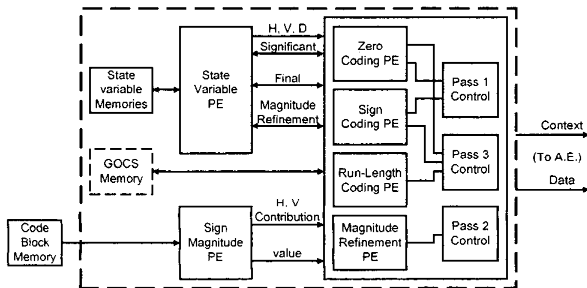

Fig. 5. Tier-1 context formation system block diagram

In Fig.5, *code block memory* stores the whole block data in a sign and magnitude separation manner. *State Variable Memories* are used for storing three different kinds of state variables (significant, magnitude refinement, final) necessary for context formation. These state variables corresponding to current coding column are then sent to *State Variable PE*. After some operation in *State Variable PE*, variables needed for 4 coding PE are generated. *Sign Magnitude PE* works similar to *State Variable PE*. Four *coding PEs* can generate contexts according to the state variables, sign and magnitude. Three *Pass Controllers* control four *coding PEs* and select the output contexts.

3.2 Column-Based Operation

The key point to applying column-based operation is to solve memory arrangement and access for significant and sign variables. For coding a single bit, we need 9 significant state variables and 9 sign variables (variables of self pixel and 8 neighboring pixels). In a column-based design, 18 significant and sign variables are needed at the same time, including current and two neighboring columns, 6 bits each, as shown in Fig. 6. Since variables from neighboring stripes are needed, it is quite a challenge to arrange and access memory. For example, if significant state variables are stored in a memory, 1 column within a stripe (4 bits) as a word, then every time a new column is processed, 3 words must be loaded from memory, which will take 3 clocks. Besides, more than needed data are loaded. Only 6 more variables are needed for coding a new column, but 12 variables are accessed.

This problem can be solved by using three smaller memories, and set 2 bits as a word. As shown in Fig. 6, variables of different row are interleaving placed in 3 memories (A, B, C, B, A, B....). By this arrangement, variables needed for processing a new column can be accessed at the same time. In a situation of continuous coding, after finish coding one column, we just shift the significant state variables in register array to left, and load one new column into right side of this array.

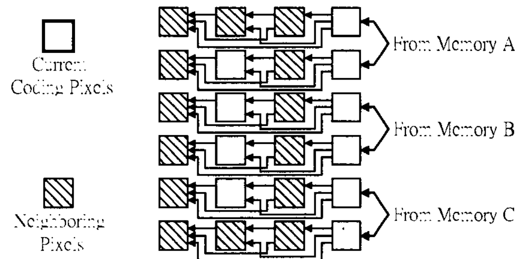

Fig. 6. Significant and sign variable registers

Fig. 7 is the architecture for *significant state variables PE*. It can provide not only the significant state variables needed, but also the sum of significant neighbors of each pixel in current coding column simultaneously. The architecture for *sign variables PE* are similar to Fig. 7, except one small converting circuit is used instead of these adders. The converting circuit converts sign variables of neighboring pixels into H,V contributions needed by *Sign Coding PE*.

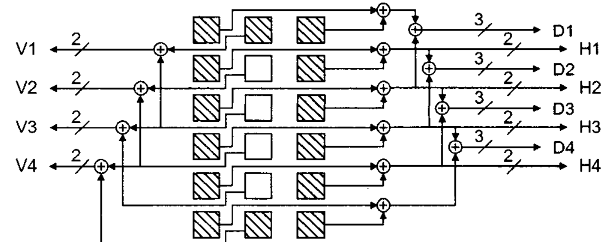

Fig. 7. First 3 columns of *significant state variable PE*

3.3 Pixel Skipping

To implement pixel skipping method, indexes for NBC pixels in this column must be generated one by one. The architecture for pixel skipping is shown in Fig. 8. A 4-bit bus indicates if pixels in

current coding column are NBC pixels or not. The pixel accumulator counts the number of pixels (0~3) already coded in this column. The index of current coding pixel is generated, then the pixel accumulator is compared with the number of NBC pixels. If all NBC pixels in this column are coded, a change column signal is generated. By this architecture, all no-operation pixels can be skipped with only a little hardware cost.

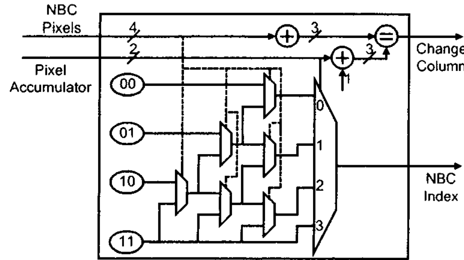

Fig. 8. Pixel skipping architecture

3.4 Group-Of-Column Skipping

Group-Of-Column skipping is applied on pass 2 and pass 3. Every 8 consecutive columns are group together as a GOC. All no-operation GOCs in these two passes can be skipped. Since the condition (NBC or not) of all GOCs in pass 2 and pass 3 must be recorded in pass 1 coding, a memory for storing the conditions of all GOCs is required. In a general case with code block size 64x64, a small memory of size 64x4 is needed (128 GOCs for each pass).

3.5 System Specifications

The design for EBCOT tier-1 coder, which includes context formation and arithmetic encoder, is finished and taped out. Tables III shows the detail specifications of this design. Note that most of the memories are necessary for EBCOT algorithm, and only 256-bit extra memory are required for proposed speed-up method.

Table III. Specifications of EBCOT tier-1 coder system

Process Technology	0.35μm CMOS 1P4M
Chip Size	3.67x3.67 μm2
Gate Count	21000 gates + 13 kbit memory
Clock Frequency	50 MHz
Supply Voltage	3.3 V
Power Consumption	115.49
Package	144 CQFP

4. EXPERIMENT RESULTS

Experiments are made on encoding various images with proposed architecture and Taubman's [3] for comparison. The processing time of context formation in tier-1 coder is shown in Table IV. It is clear that the performance improved dramatically, with more than 60% of improvement in all cases. A detail result that shows the processing time of every passes is shown in Fig. 9. In the circumstances that only PS is applied, the processing time is reduced to 43% compared with Taubman's architecture. By using PS+GOCS, processing time can be further reduced to 37%.

Image	Filter	Taubman	Proposed	Percentage
Lena	9/7	3,642,998	1,368,909	37.58%
	5/3	3,668,107	1,381,659	37.67%
Baboon	9/7	4,609,969	1,791,967	38.87%
	5/3	4,569,831	1,782,725	39.01%
Face	9/7	3,679,649	1,356,576	36.87%
	5/3	3,680,359	1,358,734	36.92%
Bike	9/7	4,120,093	1,515,449	36.78%
	5/3	4,114,038	1,518,937	36.92%

Table IV. Experimental results for processing time of proposed architecture compared with David Taubman's, on 4 different images with size 512x512, 2 kinds of filters

Fig. 9. Experimental results on detail processing time of every passes, with image 'Lena' and 9/7 filter

5. CONCLUSIONS

In this paper, analysis and architecture design of tier-1 coder of EBCOT for JPEG-2000 is presented. Column-based coding architecture with two speed-up methods, Pixel Skipping, and Group-Of-Column Skipping are proposed according to the characteristic of EBCOT algorithm. These two methods can improve the efficiency of the implementation to more than 60%.

6. REFERENCES

[1] *JPEG-2000 Part I Final Committee Draft Version 1.0*, ISO/IEC JTC1/SC29/WG1 N1646R.

[2] M. D. Adams, F. Kossentini (UBC),H. Man (SIT), T. Ebrahimi (EPFL), *JPEG-2000: The Next Generation Still Image Compression Standard*, ISO/IEC JTC 1/SC 29/WG 1N1734.

[3] C. Christopoulos, MediaLab, Ericsson Research, Sweden, *JPEG-2000 Verification Model 7.0 (Technical description)*, ISO/IEC JTC 1/SC 29/WG 1 N 1684, Apr. 2000.

[4] D. Taubman, *EBCOT: Embedded Block Coding with Optimized Truncation*, ISO/IEC JTC1/SC29/WG1 N1020R.

[5] D. Taubman, *High Performance Scalable Image Compression With EBCOT*, Proc. of IEEE International Conference on Image Processing, Kobe, Japan. 1999, vol. 3, pp. 344–348.

[6] D. Taubman, *Embedded Block Coding in JPEG-2000*, Proc. of IEEE International Conference on Image Processing, Vancouver, Canada, 2000, vol. 2, pp. 33–36.

[7] W. Pennebaker and J. Mitchell, *JPEG Still Image Compression Standard*, New York: Van Nostrand Reinhold, 1993.