OPTIMIZING THE JPEG2000 BINARY ARITHMETIC ENCODER FOR VLIW ARCHITECTURES

Brian Valentine, Oliver Sohm

Texas Instruments, Inc.

ABSTRACT

This paper proposes several techniques for optimizing the JPEG2000 binary arithmetic encoder on Very Long Instruction Word (VLIW) architectures. Binary arithmetic coding (BAC) contains a large amount of conditional and sequential processing steps that make parallelism on VLIW devices difficult to realize. The purpose of this paper is to illustrate an optimized software implementation that can software pipeline on a VLIW device. The Texas Instruments (TI) TMS320C64x Digital Signal Processor (DSP) was chosen as the implementation platform. Results of our optimized code show a 2.4x performance speed-up over a straightforward implementation of the arithmetic encoder as defined in the JPEG2000 standard.

1. INTRODUCTION

The JPEG2000 image coding standard [2] embodies the latest advances in still image compression technology. Among its benefits, JPEG2000 provides superior compression over JPEG at low bit-rates, progressive transmission, and resilience to transmission errors [3]. Embedded Block Coding with Optimized Truncation (EBCOT) and binary arithmetic coding are the most complex and computationally intensive modules of the JPEG2000 Standard [4]. These modules, in their inherent form, pose significant challenges to exploiting the inherent instruction-level parallelism of software-programmable VLIW devices. This is due to the fact that arithmetic coding algorithms contain sequential processing steps, nested conditional operations, and inner while loops that prevent efficient software pipelined schedules. To overcome this bottleneck, system designers typically tend toward custom hardware solutions [1]. Figure 1 shows the C64x VLIW-based architecture. The C64x's eight functional units allow it to process up to eight instructions in parallel. An efficient software pipelined loop will maximize usage of these resources. This paper focuses on techniques that will allow for an efficient software implementation of the arithmetic encoder on commercial-off-the-shelf DSPs such as the C64x.

The JPEG2000 binary arithmetic encoder is characterized by four functions, Code MPS, Code LPS, RENORME, and BYTEOUT [2]. These functions are executed based on the context state of the arithmetic encoder, its interval width (A), and codeword value (C). The encoder must decide if a Most Probable Symbol (MPS) or Least Probable Symbol (LPS) is encoded, whether to renormalize (RENORME) the interval width and codeword, and determine if a compressed byte needs to be sent to the bitstream (BYTEOUT). Adding to the complexity of the arithmetic encoder, the BYTEOUT procedure is embedded within the RENORME procedure, which in turn, is embedded in both the Code LPS and Code MPS procedures.

The following sections discuss several optimization steps which enable a fast software implementation of the arithmetic encoder. Section 2 describes the principles of software pipelining and how it applies to this implementation. Section 3 describes decoupling coefficient bit modeling (CBM) and arithmetic encoding. Section 4 describes the elimination of the inner while loop associated with renormalization. Section 5 discusses how to separate the BYTEOUT procedure from encoding. Section 6 discusses how to software pipeline the arithmetic encoder for the C64x DSP. Section 7 compares the execution speed of the optimized encoder with the straightforward implementation.

2. SOFTWARE PIPELINING

To obtain a fast implementation, the arithmetic encoder has to be efficiently software pipelined. Software pipelining is a scheduling technique that allows the parallel execution of multiple iterations of a loop [8,9]. This leverages the parallel architecture of VLIWs. The idea is to start executing a subsequent iteration of a loop before the previous one has completed. By finding the minimum initiation interval, the number of cycles that must execute between successive iterations, performance

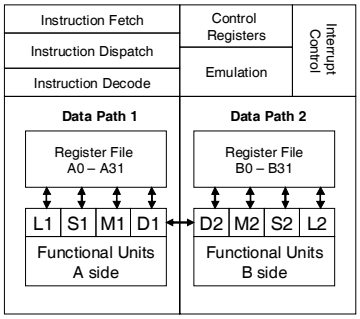

Figure 1: C64x VLIW-based DSP Architecture

can be maximized. An efficient pipelined schedule however, is prevented by the overall structure of the coefficient bit modeler, which contains nested loops, nested conditional execution paths, and long dependency paths. The optimizations described in the following sections eliminate these obstacles by restructuring the algorithm.

3. DECOUPLING THE COEFFICIENT BIT MODELER FROM THE ARITHMETIC ENCODER

In a straightforward implementation, the coefficient bit modeler would generate a single decision bit (D) and context number (CX) that are then passed to the arithmetic encoder. The arithmetic encoder would process one CX/D pair at a time. However, JPEG2000 does not require coupling of the coefficient bit modeler and arithmetic encoder [6]. To bring the arithmetic encoder into an efficient loop form, bit modeling and arithmetic encoding are decoupled. CX/D pairs are queued up in a buffer as they are generated, and later sent to the arithmetic encoder for processing. This allows the arithmetic encoder to operate on multiple CX/D pairs at once, which reduces the function call overhead, and opens up the possibility of software pipelining the loop in a later optimization step.

4. ELIMINATING THE RENORME WHILE LOOP

To qualify the arithmetic encoder for pipelining, inner loops have to be eliminated. The arithmetic encoder contains a renormalization (RENORME) while loop which is used to keep the interval width A above 0x8000. In the loop, the value of A is left-shifted by one and tested to see if (A < 0x8000) during each iteration. Figure 2 shows a reproduction of the RENORME flowchart found in Annex C of the JPEG2000 Standard [2]. If the processor implements an instruction that can determine the number of left-most zeros present in A, this while loop can be eliminated. Renormalization can then be realized by perfoming the appropriate number of bit-shifts

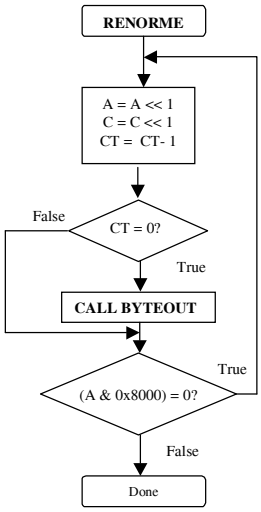

Figure 2: RENORME Procedure

in a single operation. In the hardware based arithmetic encoder proposed in [5], a "leading zero detecter" was created to determine the amount of left-most zeros in the A register. In our implementation, the C64x instruction LMBD is used [7]. Tests are included to prevent overshifting of A and C in the event that a byte from codeword C needs to be written to the bitstream. Figure 3 shows the new structure of RENORME.

5. DECOUPLING ENCODING AND BYTEOUT

It can be observed that the BYTEOUT procedure is executed at a much lower rate than actual symbol encoding. Only after a certain number of bits have been encoded will the BYTEOUT procedure append encoded bits to the bitstream. In our tests, it was found that the number of times BYTEOUT needed to be called was on average 5% of the total number of symbols encoded. Therefore, further optimization should be targeted at encoding rather than the BYTEOUT function. This is made possible by removing BYTEOUT from the encoding procedure and merging it with renormalization into a separate loop. These two loops are created in a way such that renormalization is performed in both. The first of the two loops will encode a binary decision based on the Code LPS and Code MPS procedures. Renormalization will occur if necessary. This loop will be refered to as the "encoding loop". If a BYTEOUT needs to occur, the encoding loop is exited. The second loop, which will be referred to as the "output loop", is entered to perform a BYTEOUT and complete renormalization (RENORME). The RENORME in the output loop will execute any left-shifts of A and C that were not completed prior to calling BYTEOUT. The encoding loop can now be iterated as often as possible until it is detected that a

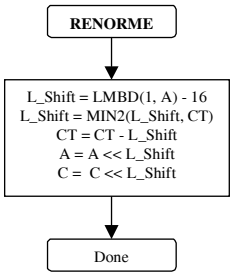

Figure 3: Modified RENORME

byte has to be output. The encoding loop is then terminated and the output loop is entered. After that, encoding resumes.

Removing the operations for BYTEOUT from the encoding procedure has the additional benefit of making the encoding loop more efficient. Cycle penalties for exiting the encoding loop to enter the output loop are negligible, since this occurs very infrequently. Figure 3 shows the modified RENORME function. The test (CT == 0?) from Figure 2 now becomes an exit condition for the encoding loop instead of a test embedded in RENORME.

6. ENCODING LOOP OPTIMIZATION

Software pipelining of the encoding loop will allow for the loads of CX/D pairs, lookup table values, and context states for subsequent iterations to occur in parallel with encoding operations. This will reduce the number of cycles needed to encode a single CX/D pair because the latencies associated with loads from memory can be hidden within encoder computations.

Since computations on the arithmetic encoder interval width A and codeword C for a future iteration cannot occur until the current iteration has completed its update of A and C, a limit is placed on the minimum initiation interval that can be achieved in a pipelined loop. The remaining optimization steps shorten existing recurrence paths as to allow for a small initiation interval and thus, a more efficient software pipelined schedule.

6.1. Exploiting Parallelism Across Conditional Execution Paths

The arithmetic encoder contains many nested conditional statements that limit parallelism. Algorithm restructuring was performed to minimize the number of different conditional execution paths and introduce more parallelism. This was achieved by determining which instructions could be executed speculatively, and minimizing the number of predication flags [9] and conditional expressions required. In this regard, several observations were made that allowed for simplification of the control flow.

All conditions that direct the code flow are computed at the beginning and then used to predicate the encoding operations [9]. The majority of the code flow is directed by conditions which are based on whether a LPS or MPS is coded, and the value of A with respect to Qe and the constant 0x8000. Some of these conditions can be computed in parallel. The two computations of the arithmetic encoder, (A = Qe) and (C = C + Qe), are common to Code MPS and Code LPS.

The C64x provides predicate registers which will enable the conditional execution of operations based on the Boolean value of associated conditions [7]. Using predicated execution effectively collapses the Code MPS and Code LPS procedures into a single process, shortening the recurrence path and thus, allowing for more efficient software pipelining of the encoding loop.

6.2. Optimized Data Packing

An optimum storage format in memory was devised for the context state data which minimizes the number of operations required to load, store, and extract the individual elements. The probability look-up table index and MPS switch values are packed into one byte. To also make context state updates more efficient, the probability look-up table is structured in such a way that an updated MPS switch can be easily packed into the registers containing the next index (NMPS/NLPS).

6.3. Eliminating Memory Dependencies

In the special case that a context number CX from iteration i is equal to the CX from iteration (i-1), and the CX state was updated in iteration (i-1), a memory dependency will exist. This is because iteration i must use the CX state updated in iteration (i-1). This memory dependency is eliminated by obtaining the updated state data for the next iteration directly from the register it was written to in the previous iteration, rather than from memory. This effectively replaces the load operation (and all associated delay slots) in the recurrence path with a simple register move operation. Note that there is no dependency if the CX from iteration i is different from the CX of iteration (i-1). In this case, the context state for the next iteration can be read before the context state of the previous iteration is updated in memory.

6.4. Software Pipelined Encoder Loop

Figure 4 is a graphical representation of the loop pipeline that shows how iterations overlap. The software pipelined

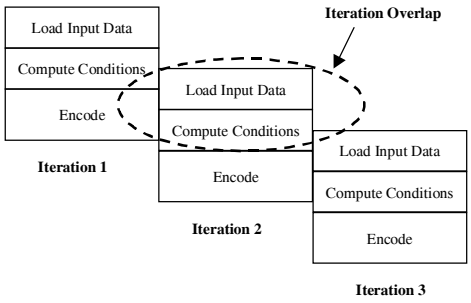

Figure 4: Encoding Loop Pipeline

encoder loop will encode a CX/D pair with loads for subsequent CX/D pairs, context states, and probabilities occurring in parallel. In the event that a BYTEOUT becomes necessary, an early exit of the encoding loop is forced and the output loop (containing BYTEOUT and RENORME) is entered. The pipelined loop contains instructions to preserve live-out variables in the event that the encoding loop is exited. This will allow for the reversal of any loads, stores, and computations for future iterations that should not have occurred. In the event that a byte needs to be written to the bitstream, the output loop is entered. If there are still remaining CX/D pairs to be encoded after a BYTEOUT, the encoding loop is re-entered.

7. ARITHMETIC ENCODER PERFORMANCE

A straightforward implementation of the arithmetic encoder as defined in Annex C of the JPEG2000 Standard text [2], and our optimized version, were benchmarked on the TMS320C6416 DSP. The software pipelined encoding loop operates at a rate of 13 cycles per context/decision pair. The number of cycles spent on the output loop and pipeline re-initialization is approximately 71. Since about 95% of the processing occurs in the encoding loop, the 71 cycles spent outside of the pipelined kernel has a minimal effect on overall performance. Our results show an average 2.4x speed-up when comparing our optimized kernel to the straightforward implementation. The tests were performed on several photographic images using a JPEG2000 encoder with a 9/7 lifting-based irreversible wavelet filter, and YUV 4:4:4 color input images. The arithmetic encoder was called once per coding pass (significance, refinement, cleanup), and terminated after encoding one codeblock. Results are shown in Table 1.

8. CONCLUSION

Techniques for software pipelining the JPEG2000 binary arithmetic encoder were discussed. The arithmetic encoder was implemented on a C64x DSP to show the benefits of the described optimizations. A performance boost of 2.4x over a straightforward implementation was shown. Some of the described optimization techniques may also be applied to the JPEG2000 binary arithmetic decoder and to some extent, other arithmetic coders such as those used in H.264.

Image 512x512	Original (Cycles)	Optimized (Cycles)	Com-pression	Speed -up
Lena	35,129,408	14,538,352	15.1	2.4x
Peppers	44,042,024	18,216,192	12.6	2.4x
Baboon	92,802,832	38,775,032	4.6	2.4x

Table 1. Arithmetic Encoder Comparison

9. REFERENCES

[1] K. Andra, C. Chakrabarti, and T. Acharya, "A High-Performance JPEG2000 Architecture," *IEEE Trans. on Circuits and Systems for Video Technology*, vol 13, No. 3, pp. 209-218, March 2003.

[2] ISO/IEC 15444-1:2000, "Information technology - JPEG 2000 image coding system - Part 1: Core coding system", July 31, 2002

[3] C. Christopoulos, A. Skodras and T. Ebrahimi, "The JPEG2000 Still Image Coding System: An Overview," *IEEE Trans. on Consumer Electronics*, vol 46, No. 4, pp. 1103-1127, Nov 2000.

[4] C. Lian, K. Chen, H. Chen, and L. Chen, "Analysis and Architecture Design of Block-Coding Engine for EBCOT in JPEG 2000," *IEEE Trans. on Circuits and Systems for Video Technology*, vol 13, No. 3, pp. 219-230, March 2003.

[5] K. Ong, W. Chang, Y. Tseng, Y. Lee, and C. Lee, "A High Throughput Context-based Adaptive Arithmetic Codec for JPEG2000," *IEEE International Symposium on Circuits and Systems*, vol 4, pp. 133-136, May 2002.

[6] D.S. Taubman and M.W. Marcellin, *JPEG2000 Image Compression Fundamentals, Standards and Practice*, Kluwer Academic Publishers, Boston, 2002.

[7] Texas Instruments, *TMS320C6000 CPU and Instruction Set Reference Guide*, SPRU 189, Oct 2000.

[8] Texas Instruments, *TMS320C6000 Programmer's Guide*, SPRU 198, Aug 2002.

[9] N.J. Warter, D.M. Lavery, and W.W. Hwu, "The Benefit of Predicated Execution for Software Pipelining," *Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences*, vol 1, pp. 497-506, Jan 1993.