Analysis of Multiple Parallel Block Coding in JPEG2000

Michael Dyer, Saeid Nooshabadi, David Taubman
School of Electrical Engineering and Telecommunications
The University of New South Wales, Sydney, Australia
{m.dyer, saeid, d.taubman}@unsw.edu.au

Abstract— **We present the analysis and results for a System on a Chip (SoC) Software/Hardware Codesign platform, for parallel coding in JPEG2000 compression standard. We show that there are optimum numbers of parallel block coders and scheduling granularity per row of codeblocks. The system was implemented on an Altera NIOS II processor with flexible integrated peripheral.**

Index Terms— **Image coding, Parallel processing.**

I. Introduction

JPEG2000 [1] is a relatively recent image compression scheme. The main computational blocks in the compression scheme are Discrete Wavelet Transform (DWT) and the algorithm known as Embedded Block Coding with Optimal Truncation [2]. EBCOT is responsible for modeling and entropy coding of independent blocks of quantized DWT coefficients.

The Kakadu library [3] is a software implementation of JPEG2000. Measurements show that 61.65% of time taken by Kakadu during the image processing is spent performing the block coding, verifying the claim of [4]. This presents significant motivation for providing parallel hardware block coders as a form of acceleration. This work is based on a hardware/ software codesign platform in [5] that was produced to test system level integration of high performance BitPlane Coder (BPC) of [6] and the Arithmetic Coders (ACs) of [7].

A pure hardware implementation is necessarily limited in the features it provides. Therefore, instead of implementing a complete codec on-chip, it is desirable to split the task between a general microprocessor, dedicated DWT hardware and dedicated block coding hardware. By retaining the flexibility of the Kakadu library, we are able implement an extremely generic JPEG2000 coder. The encoder possesses all the features of the JPEG2000 standard provided by the software library.

The processor chosen was the Altera NIOS II softcore processor [8], running the uClinux embedded operating system [9]. This processor is implemented on an Altera Stratix FPGA. Because of the soft nature of this processor, the addition of new peripherals to the system is quite simple. The system is built around the Avalon Switch Fabric [10] for communication between the NIOS processor and peripherals, instead of a shared bus. The switching fabric allows one to one communication between multiple devices at the same time. This feature is utilized by providing a separate physical memory for processor code and DWT coefficients, allowing the NIOS processor to continually access its own instruction/data memory while the bit plane coder extracts DWT coefficients from the DWT memory without collision. Figure 1 shows the high level block diagram of our system. The SRAM is used to store only DWT coefficients, while the code and other data are stored in the larger SDRAM.

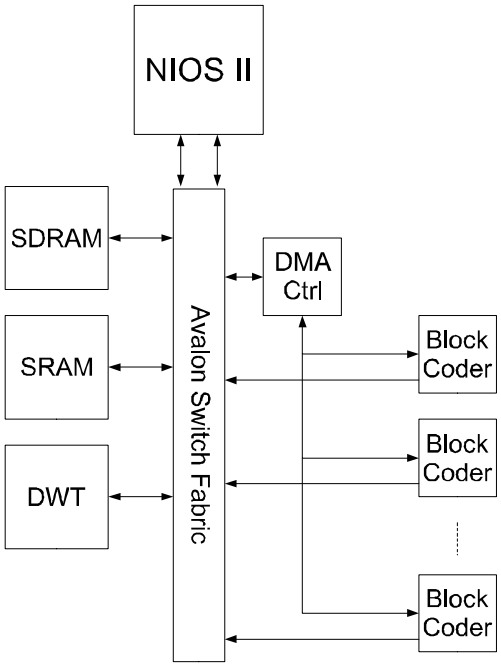

Fig. 1: NIOS II JPEG2000 System with Multiple Block Coders

II. Block Coding Peripheral

Figure 2 shows the block diagram of the block coding peripheral designed to interface to the Avalon switch. Data is written into the Codeblock RAM by a DMA controller, transferring DWT coefficients from the SRAM. A 32 bit wide data bus is used, so that two coefficients can be transferred in each clock cycle to improve transfer time. The REORG unit performs conversion of DWT coefficients from coefficient order to bitplane order, so that bitplanes are contiguous in memory. This is an aspect of design that is surprisingly absent from the literature and has the potential to bottleneck a block coding system. The BPC of [6] requires each bitplane to be supplied a column at a time. During the Most Significant Bitplane (MSB) , the sign bitplane is also loaded. The two column First In First Out (FIFO) buffers, (COL FIFOs) store columns of bitplane data produced by the REORG unit. During the first bitplane load, one FIFO holds MSB data, and the other sign data. After the first bitplane, the two FIFOs can be used to double buffer column data.

As the BPC of [6] can produce anywhere between 0 and 10 Context Data (CxD) pairs, and these pairs may be present on random CxD signal lines, a CxD arrange (CXD ARG) unit needed to be designed. This unit uses reticulation to ensure that any CxD pairs always occupy the lowest CxD lines, while maintaining order. This is required so

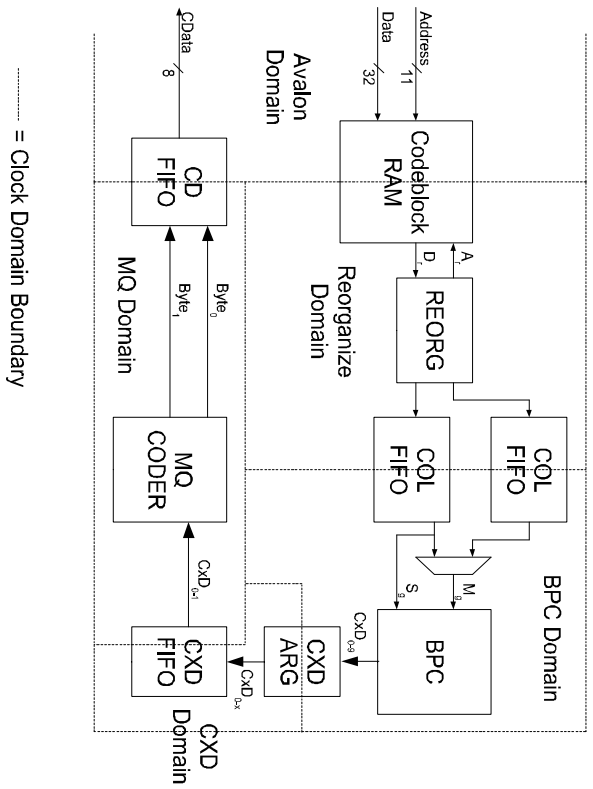

Fig. 2: Block Diagram of Avalon Block Coder Peripheral

that the CxD FIFO load system will always know where to find CxD pairs, without searching. The CXD FIFO provides one or two CxD pairs to the MQ Coder (depending on which MQ Coder has been chosen from [7]). The CXD FIFO can also be designed such that it can be concurrently loaded with more than one CxD pair per clock cycle. Concurrent loading reduces the clock frequency required in the CxD domain, but requires specially designed FIFOs.

III. Parallel Coding On the Codesign Platform

The Kakadu library natively supports multithreading, so it is more than capable of distributing codeblocks to the peripheral block coding hardware. Kakadu performs simple scheduling via queues of jobs. As the DWT progresses, more and more lines become available and are added to the job queues. It should be remembered that three rows are produced at once, one in each of the LH, HL and HH subbands. Once enough lines have been produced to form a complete row of codeblocks, the scheduler begins to distribute work to the block coder threads. There is a level of granularity to the scheduling system. In its simplest form, the scheduler can hand over a complete row of codeblocks to a block coding thread, and this thread will process the entire row of blocks until completion. Alternatively, the row can be segmented into smaller groups of codeblocks, each of which forms a separate job that can be scheduled to a block coder thread. Intuitively, it would be expected that the greatest improvement would be seen when three block coder threads are available, one for each subband. However, it must be noted that the DWT is performed in parallel with block coding. Thus, more than three rows of codeblocks may become available as the DWT progresses in the lower levels of decomposition. In this case, it is expected that coding time would benefit from the addition of more then three block coders.

To improve the block coder utilization, segmentation of the row of codeblocks is essential. For rows that have little data, block coding may finish quite quickly. If a complete row is assigned to a block coder, then the coder that has become newly idle cannot help with the processing of another row of blocks. Segmentation of the job would allow the newly idle coder to switch to another row and hence improve coder utilization and the processing time. It is tempting to segment the rows to a level where each codeblock becomes a single job, however, finer granularity adversely impacts the scheduling as it happens more often because each job is small.

Performance analysis took place in two parts. Firstly, a test image is run through Kakadu to determine how many CxD pairs per column would be produced by the bitplane coder. These results were then post processed to determine how many clock cycles would be consumed coding these pairs. This was performed for both single symbol and two symbol ACs, with varying CXD FIFO lengths and loading concurrency.

A second analysis is then performed on the test image. The information collected from the first analysis allows us to analyze the performance of parallel hardware block coding. By enabling the multithreaded features of the Kakadu library, multiple, parallel software block coders threads are created. During this stage of analysis, the block coder threads simply start off the hardware block coders. The software block coder threads merely sleep for the amount of time a real block coder would take to process the block. Because DWT coefficients must be transferred from SRAM to the block coder, a mutex is created. When a software block coder is told to begin processing a block, it locks the mutex for the same amount of time it would take to transfer the coefficients. This will cause other block coders requiring DWT coefficients to idle while they wait for the memory bus. The memory bandwidth between the SRAM and the block coders will determine how much time is taken to transfer coefficients, and will also affect the how often threads block each other. Statistical collection code was added to the threads, collecting information on how long each block coder spends idle, transferring coefficients or processing the codeblock. The DWT engine is assumed to be capable of processing one image sample per clock cycle (not unreasonable, considering the published work on DWT design[11]). Each analysis is run 10 times and averaged, to smooth out any variations caused by local processor load. The time taken for block coding the image is the maximum sum of the idle, memory transfer and coding times out of all the block coder threads. This is because threads run in parallel, and thus it is the thread that finishes last that determines image block coding time.

Various analysis were run, to investigate the effects of different levels of implementation complexity and efficiency. In all systems, it is assumed that the MQ AC and BPC are clocked at the same rate, and that this rate is 50MHz. The REORG unit is assumed to run at clock rates high enough to prevent stalling under these conditions. Analysis variables are MQ AC CxD concurrency, CXD FIFO load concurrency (how many CxDs can be simultaneously loaded into the CxD FIFO, after the CxD Arrange unit), CxD FIFO length, Block Coder memory bandwidth, and Scheduler Granularity (how many codeblocks are allocated at once to each block coder thread)

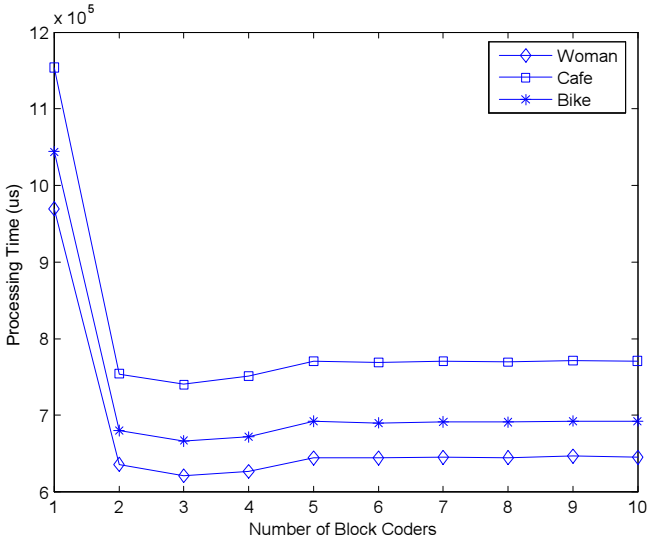

Fig. 3: Processing time, Multiple Block Coders, 50MHz Memory Bus, no FIFO, Single CxD MQ Coder

IV. Results

Our analysis shows that increasing the CXD FIFO length past 16 words (for both 1 and 2 CxD/Cycle MQ Coders, in a system with a single block coder), does not result in significant improvement in the processing time. We also analyzed the effect of load concurrency on a single block coding system with CxD FIFO length of 16 and a 2 CxD/cycle MQ AC. Exceeding a concurrent load capability of 4 CxD pairs/cycle does not exhibit much improvement. This is to be expected, as the columns of 4 samples high are unlikely to produce more than one CxD pair each.

Figure 3 shows the results for multiple block coders for the simplest of systems. Each codeblock row is a single job, there is a single CxD MQ AC, no FIFO coupling between the BPC and AC modules, and only one sample is transferred to the codeblock RAM every clock cycle at 50MHz. As predicted, the system shows the greatest improvement in the processing time when there are 3 block coders present, one to process each subband. Interestingly, there is a slight rise in the processing time as the number of block coders increases past 3. This could be due either to increased scheduling activity, or memory access being bottlenecked as coders fight for DWT coefficients. To study the influence of memory access effects, the analysis of Figure 3 were repeated with increasing memory clock speed and with a Single Sample (SS) or Double Sample (DS) per cycle transfers of CxD pairs into the CXD FIFO. It was observed that the rise in the coding time for more than 3 parallel coders is still present. It can be inferred that a memory bottleneck is not occurring.

Figure 4 shows the fractional reduction (averaged over the images Woman, Cafe and Bike) in the processing time as scheduler granularity is increased to give more than one job per row of codeblocks (if there are enough codeblocks in the row to support this). There are 3 distinct trends apparent. When the number of jobs per row is less than 5, it can be seen that additional reduction occurs in the processing time as the number of block coders increases. When the number of jobs is between 5 and 8, significant improvement is seen with the addition of multiple coders. With the number of jobs per row between 9 and 10, the multiple coders become less effective. Figure 5 shows the block coder activity for a system with 4 'simple' block coder peripherals with 1 CxD/Cycle MQ AC and no FIFO CxD . As can be seen, a system with 4 jobs per row still has a lot of idle time on the block coders. With 8 jobs per row, work appears more evenly distributed. However at 10 jobs per row, the scheduler is not distributing work evenly. This is because with the assumed codeblock size of 64×64 there are only 16 codeblocks in a row, resulting in some jobs with two codeblocks and others just with one codeblock, creating an unbalanced load. We carried out a similar analysis with a 'fast' block coder peripheral, with a 2 CxD/Cycle MQ AC, FIFO load concurrency of 4 CxD pairs, and a 16 word CXD FIFO. We noticed the same characteristics as using a slower 'simple' coder (although overall coding time is obviously improved). As predicted, it is necessary to segment rows into more than one job in order to better utilize parallel block coders. In the assumed system, memory bottlenecks do not seem to occur. Scheduler overhead appears to have more of an effect than was originally anticipated, as demonstrated by the poor scheduling pattern which leads to idleness of block coders when rows are segmented into more than 8 jobs.

Table I shows the improvement in coding time for each proposed system with a scheduler granularity of 8 jobs/row. Providing parallel block coders presents a clear improvement over a single block coder. Although significant reduction in the processing time up to 24.2% can be obtained by the use of a more costly 'fast' block coder, up to 69.6% reduction is available when using parallel coders. In terms of hardware cost, a single fast block coder requires 30% more logic cells and 3% times more memory bits and is 1.32 times faster. The fast parallel system, compared with the simple system requires 5.214 times more logic cells, 4.14 times more memory, and reduces the processing time by 3.18 times. Although substantially more costly, the parallel system provides a considerable reduction in the processing time.

Image/System	Simple	Fast	Simple 4-Parallel	Fast 4-Parallel
Woman	891081	685728.6	336199.1	285007.8
Cafe	1077183.8	816104.9	402738.5	338268.9
Bike	965111	747284.7	362226.1	306924.9

TABLE I: Coding times (μs)

V. Conclusion

The codesign on a systems with fast parallel block coders demonstrates that substantial improvements in image the processing time (about 70%) were investigated to be possible. However, such a performance improvement in the system would require proportional increase in the hardware cost. There is also an optimum number of parallel coders beyond which the performance would not increase. The system maintained the features of a completely soft implementation of the standard, while providing significant improvement in throughput via the use of hardware acceleration.

References

[1] "ISO/IEC international standard 15444-1 JPEG2000 image coding system," 2004.

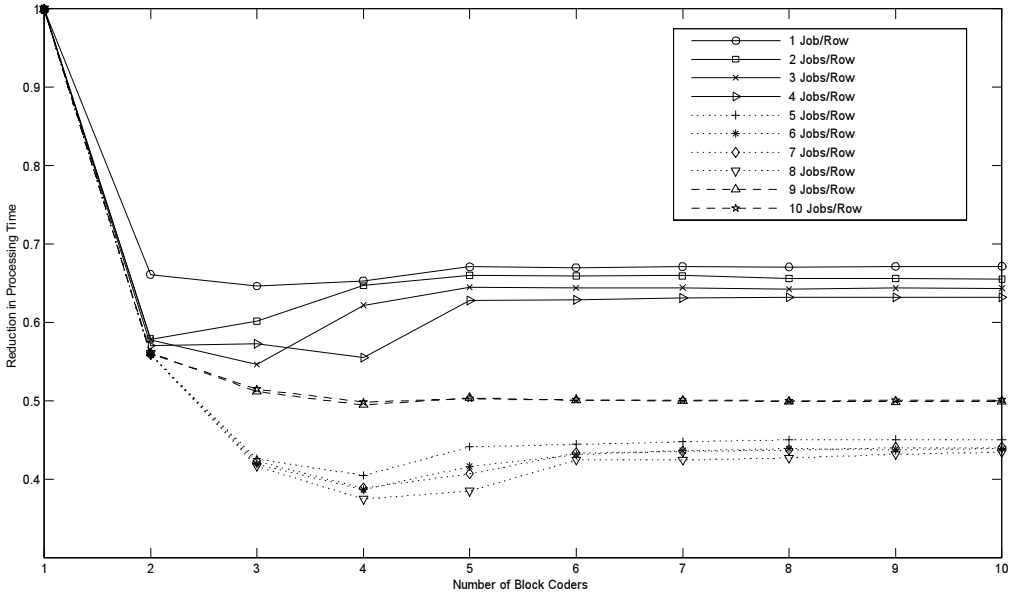

Fig. 4: Effect of Scheduler Granularity on Processing Time, with Varying Numbers of Block Coders

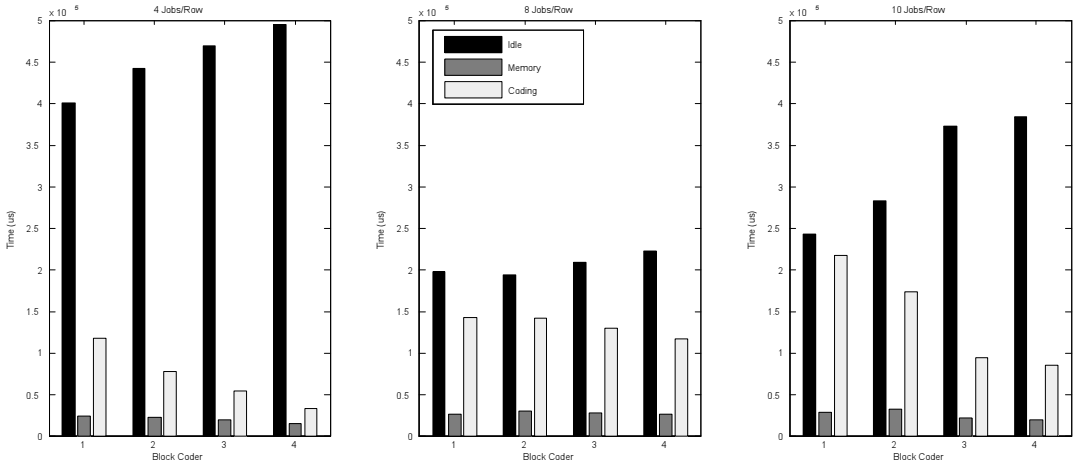

Fig. 5: Distribution of Coder Activity

[2] D. Taubman, E. Ordentlich, M. Weinberger, G. Seroussi, I. Ueno, and F. Ono, "Embedded block coding in JPEG2000," in *Proc. Int. Conf. on Image Process. (ICIP 02)*, vol. 2, Sept. 2000, pp. 33–36.

[3] D. Taubman, "Kakadu software library," http://www.kakadusoftware.com/.

[4] K.-F. Chen, C.-J. Lian, H.-H. Chen, and L.-G. Chen, "Analysis and architecture design of EBCOT for JPEG-2000," in *Proc. Int. Sym. on Cir. and Sys. (ISCAS'01)*, vol. 2, 2001, pp. 765–768.

[5] M. Dyer, A. K. Gupta, N. Galin, and S. Nooshabadi, "Case study: Hardware acceleration of the JPEG2000 kakadu library," in *Proc. Midwest Sym. on Cir. and Sys. (MWSCAS'06)*, 2006.

[6] A. K. Gupta, D. Taubman, and S. Nooshabadi, "High speed VLSI architecture for bit plane encoder of JPEG2000," in *Proc. Midwest Sym. on Cir. and Sys. (MWSCAS'04)*, vol. 2, 2004, pp. II233 – II236.

[7] M. Dyer, D. Taubman, S. Nooshabadi, and A. K. Gupta, "Concurrency techniques for arithmetic coding in jpeg2000," , *IEEE Trans. on Cir. and Sys. I: Regular Papers*, vol. 53, no. 6, pp. 1203–1213, 2006.

[8] Altera Corporation, "Nios II processor," http://www.altera.com/products/ip/processors/nios2/ni2-index.html.

[9] "uClinux - Embedded Linux Microcontroller Project," http://www.uclinux.org.

[10] Altera Corporation, "Avalon switch fabric," http://www.altera.com/literature/hb/qts/qts_qii54003.pdf.

[11] N. Zervas, G. Anagnostopoulos, V. Spiliotopoulos, Y. Andreopoulos, and C. Goutis, "Evaluation of design alternatives for the 2-d-discrete wavelet transform," *IEEE Trans. on Cir. and Sys. for Video Tech.*, vol. 11, pp. 1246 – 1262, 2001.