High-Speed H.264/AVC CABAC Decoding

Yongseok Yi, *Student Member, IEEE*, and In-Cheol Park, *Senior Member, IEEE*

***Abstract*—The decoding of context-based adaptive binary arithmetic coding (CABAC) imposes a heavy performance requirement on H.264/AVC decoding systems particularly for large-scale video sequences. As a simple approach of elevating the operating frequency is not sufficient to meet the performance requirement, this paper proposes an efficient approach to accelerate the decoding, which is effective under relatively low operating frequency. Since the CABAC decoding procedure is highly sequential and has strong data dependencies, it is difficult to exploit parallelism and pipeline schemes. The proposed approach resolves the difficulties by modifying the operation chain based on a thorough analysis, eventually enabling both parallel operations and pipelining. More specifically, 1) several context models are simultaneously loaded from memory while context selection is performed in parallel and 2) bin-level pipelining is enabled by employing a small storage to remove structural hazards and data dependencies. Experimental results show that the proposed approach leads to the real-time decoding of HD sequences.**

***Index Terms*—Adaptive binary arithmetic code, entropy coding, H.264/AVC, syntax processing.**

I. Introduction

CONTEXT-BASED adaptive binary arithmetic coding (CABAC) [1], used in H.264/AVC [2], is an extension of the binary arithmetic coding (BAC), where the coding offset value is dynamically adjusted based on the syntax element being encoded/decoded. While the coding efficiency of CABAC is superior to the conventional Huffman coding [3], the improvement comes with an increasing performance requirement; at least 3 GHz of computing power is required for the real-time decoding of a HD sequence if a general-purpose, yet high-speed, RISC machine processes the syntax parsing. As HD digital TV broadcasting coded in H.264/AVC Main Profile and High Profile is being widely spread at the present, the necessity of a high-speed CABAC decoder is growing.

CABAC originates from the generic arithmetic coding [4] to which a majority of researches have been devoted to remove the multiplication. Skew Coder [4], [5], Q-Coder [6], and other variants [7]–[11] are the examples. Most of the works also have tried to approximate real-valued symbol probabilities to integer values. As a result, the generic arithmetic coding has evolved to CABAC. However, there have been relatively few works on the practical implementation of CABAC decoding. The CABAC encoder presented by Osorio *et al.* [12] separates block coefficients from ordinary syntax elements, and processes the former using an optimized hard-wired engine while the latter is processed by software. This approach is also found in [13], which speeds up the decoding by using pointer chains to retrieve the context models for the subsequent bins and storing the intermediate results of context selection instead of the original reference macroblock data. Chen *et al.* [14] have proposed a hardware accelerator for CABAC decoding. However, there is no specific acceleration scheme except that the decoding is controlled by an optimized finite state machine (FSM). Moreover, it is reported that the performance is just enough to accommodate CIF 30 fps. There are also several commercial products announced for CABAC encoding and decoding [15], [16], but their internal structures and mechanisms are not publicized.

This paper proposes efficient techniques to alleviate the performance problem that is more significantly imposed when a high resolution video stream should be decoded in real-time. More specifically, 1) several context models are simultaneously loaded from memory while context selection is achieved in parallel and 2) bin-level pipelining is enabled by employing a small storage to remove pipeline hazards.

II. CABAC Decoding

For every syntax element s, a series of bins are decoded to make a bin string. If the bin string is found to be valid by *binarization matching*, the decoding of the current syntax element is finished with the syntax element value produced by *de-binarization*.

There are three types of operations to acquire a bin: equiprobable, nonequiprobable, and terminate decoding. We focus only on the nonequiprobable decoding because the others are much simpler and can be performed using the nonequiprobable decoding routine with slight modifications.

To decode a bin b_i, a proper context model c_i should be provided. This operation usually involves a context memory read in which the context index γ_i acts as the address. γ_i is the sum of a context index increment $\gamma_{\mathrm{incr},i}$ and a base context index $\gamma_{\mathrm{base}}(s)$, the former of which should be derived by *context selection* and the latter is given uniquely for s.

After a context model is obtained, the binary arithmetic decoding takes place. The symbol is decided by comparing the coding offset with the most-probable symbol (MPS) subrange that is derived from the coding range and c_i. Then, the *renormalization* follows to keep the coding range and the coding offset to a fixed precision, in which the coding offset is compensated by appending the incoming bitstream.

When implemented in hardware, the bin decoding procedure is realized by a series of elementary operations as illustrated in Fig. 1(a) in which the operations are delimited by tentative cycle boundaries. Given $\gamma_{\mathrm{base}}(s)$, the context selection produces γ_i which is fed to the context model loading operation to read the corresponding c_i from the memory. After that, the binary arithmetic decoding operation computes the value of b_i and a new

Manuscript received October 11, 2005; revised September 15, 2006. This work was supported by the Institute of Information Technology Assessment through the ITRC, by the Korea Science and Engineering Foundation through the MICROS and by the IC Design Education Center (IDEC). This paper was recommended by Associate Editor S-U. Lee.

Y. Yi is with Samsung Electronics Company, Ltd., Kyeonggi Do 443-742, Korea (e-mail: yongseok.yi@samsung.com).

I.-C. Park is with Korea Advanced Institute of Science and Technology, Daejeon 305-701, Korea (e-mail: icpark@ee.kaist.ac.kr).

Digital Object Identifier 10.1109/TCSVT.2007.893831

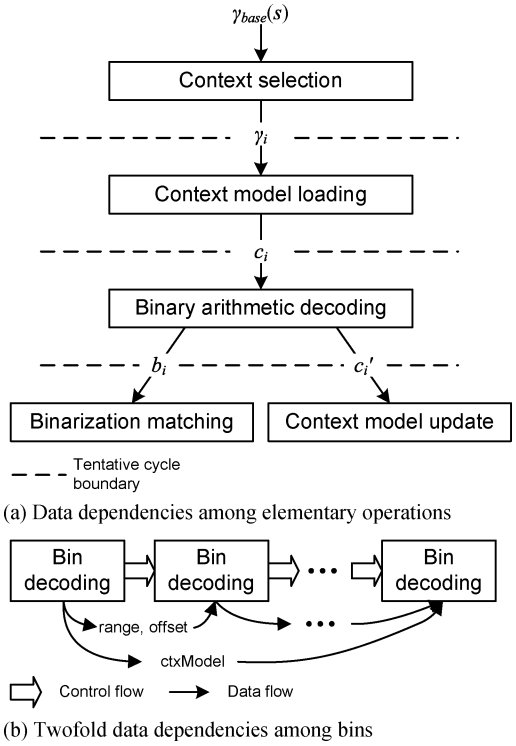

Fig. 1. Elementary operations of CABAC decoding and their data dependencies.

context model c_i'. The binarization matching operation evaluates the bin string constructed so far and decides whether to complete the syntax element decoding or not. At the same time, c_i' is stored to the memory. Due to these strict data dependencies, the elementary operations can hardly be processed in parallel.

Moreover, the internal variables of the binary arithmetic decoding, the coding range and the coding offset, introduce inter-bin dependency because they should be maintained consistently throughout all the bin decodings. The context model also participates in the dependency because it can be used again when the same type of syntax element is decoded later. Fig. 1(b) shows the twofold inter-bin data dependencies that prevent multiple bin decodings from being processed in parallel.

III. Acceleration Schemes

Given a chain of operations, the most effective way to boost the performance is to exploit the pipelining scheme. Each of the elementary operations enumerated in Fig. 1(a), i.e., context selection (CS), context model loading (CL), binary arithmetic decoding (BAD), binarization matching (BM), and context model update (CU), can be implemented to operate in a single cycle. Thus, the initial pipeline configuration is to assign each operation to a pipeline stage as depicted in Fig. 2, where BM and CU take places at the same cycle because they have no data dependency. Since all the syntax elements are made up of one or more bins and every bin is decoded by the same operation chain, the decoding can be greatly accelerated if the pipeline operates without stalls. However, there exist some data hazards and a structural hazard that impede the pipeline.

Although the purpose of the pipelining scheme is to boost the throughput of bin decodings, a shorter latency is also necessary

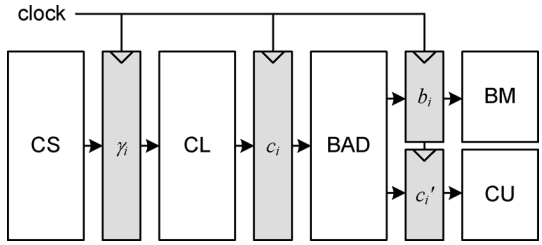

Fig. 2. Initial pipeline configuration.

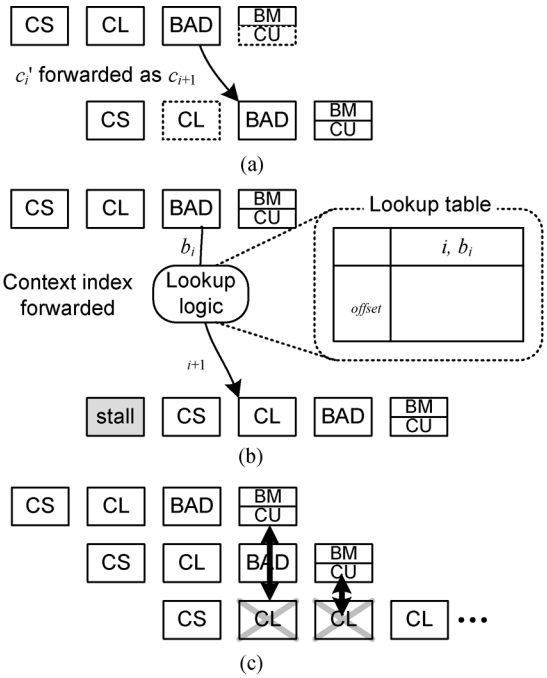

Fig. 3. Pipeline hazards. (a) Data hazard due to context model. It is resolved by forwarding the changed context model. (b) Data hazard caused by the context selection based on previously decoded bin values. (c) Structural hazard caused by CL and CU.

because the average number of bins for a syntax element is less than two for a usual H.264/AVC sequence. Reducing the latency will be discussed later in this section.

A. Pipeline Hazards

One data hazard comes from the data dependency by the context model, as the CL of a future bin might attempt to load the same type of context model used by the current bin before its new value is updated to the context memory. Since BAD is both the producer and consumer of the context model, this hazard can easily be resolved by forwarding the context model from the output of BM to the input as depicted in Fig. 3(a), in which the first CU and the second CL is ineffective due to the forwarding.

In some cases, context selection is performed based on the previously decoded bin values rather than based on the evaluation of neighbor data or by simply incrementing the context index. In most cases, c_i is calculated by referring to b_{i-1}, and in the remaining cases, c_i is calculated based on b_{i-2}. This type of context selection can be achieved by using a lookup logic that takes $\gamma_{\text{base}}(s)$, b_i, and bin index i as inputs and gives the corresponding γ_i as an output. An easy way to relieve the depen-

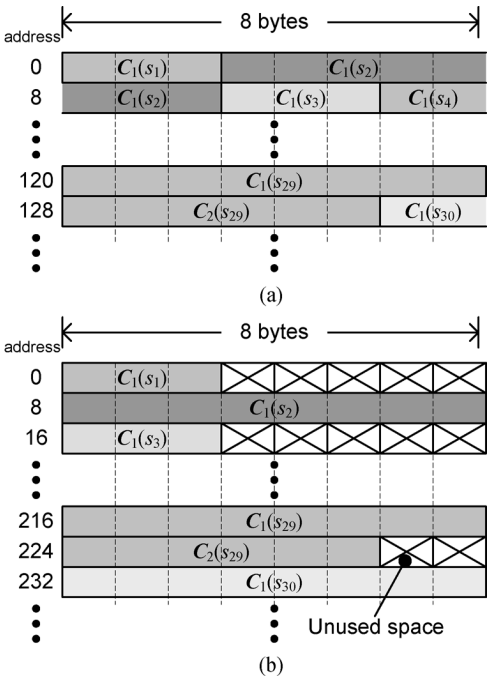

Fig. 4. Modification to the data arrangement of the context memory. (a) Optimal data arrangement of context memory. (b) Modified data arrangement of context memory.

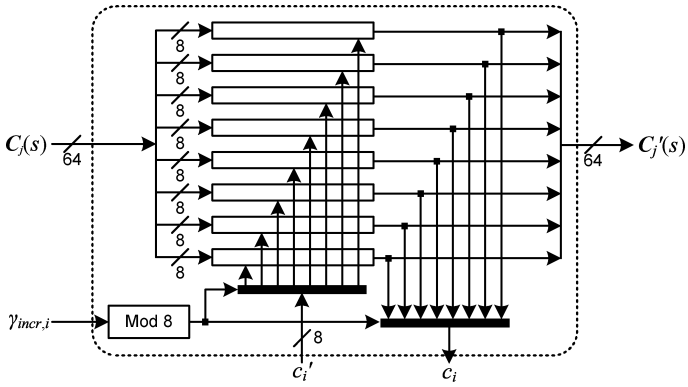

Fig. 5. Structure of CMR.

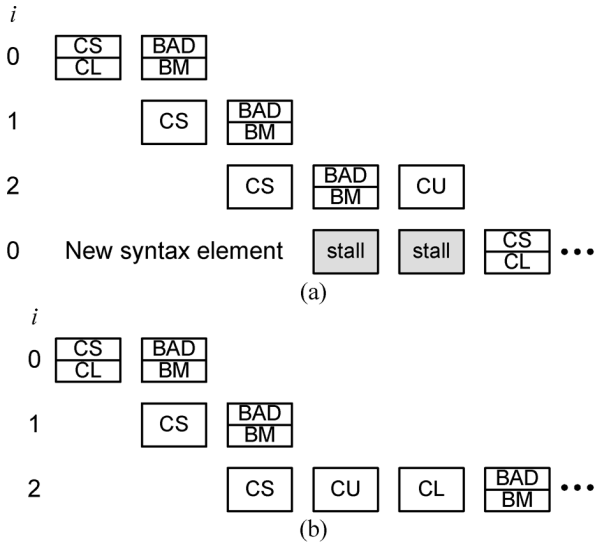

Fig. 6. Pipeline configuration after applying the proposed schemes. (a) Pipeline without CMR switching. Stalls of two cycles are necessary to begin a new syntax element. (b) Pipeline with CMR switching at $i = 2$.

dency is exploiting a forwarding path from BAD to CL that is associated with the lookup logic as illustrated in Fig. 3(b). Nevertheless, we cannot avoid the stall of one cycle in the current configuration.

The last problem in the initial pipeline arrangement is the structural hazard caused by CL and CU as both of the operations access the same memory, one to read an another to write, as depicted in Fig. 3(c). To resolve this conflict, one of the two operations should be removed from the operation chain. As CL supplies the context model, it is essential for the rest of the operations. The remaining one, CU, may be postponed because it is relevant only to the future bins, not to the current one. Using a small storage, hereinafter called the context model reservoir (CMR), we can postpone CU to make it out of the current decoding flow. The complete structure of CMR will be presented after another important effect of CMR is discussed.

B. Parallel Processing of Elementary Operations

Besides relieving the structural hazard, CMR also enables the parallel processing of CS and CL, eventually leading to a decreased latency and the elimination of the stall caused by the context selection based on previously decoded bins.

Let $\mathbf{C}(s)$ be the set of context models associated with a syntax element $s \in \mathbf{S}$. All the context models in $\mathbf{C}(s)$ are indexed by contiguous context indices. If s and t are different, $\mathbf{C}(s)$ and $\mathbf{C}(t)$ are distinct. Two distinct context model sets $\mathbf{C}_A$ and $\mathbf{C}_B$ are said to be adjacent if the maximum context index $\gamma_{\max}$ of $\mathbf{C}_A$ equals to $\mu_{\min} - 1$ in which $\mu_{\min}$ is the minimum context index of $\mathbf{C}_B$. In this case, $\mathbf{C}_B$ is upwardly adjacent to $\mathbf{C}_A$. $\mathbf{C}(s)$ can be broken down into adjacent and distinct subsets, $\mathbf{C}_1(s), \mathbf{C}_2(s), \ldots, \mathbf{C}_m(s) \subset \mathbf{C}(s)$, where $\mathbf{C}_1(s)$ is the first subset containing the context model corresponding to the minimum context index of $\mathbf{C}(s)$, *i.e.*, $\gamma_{\text{base}}(s)$, and $\mathbf{C}_{j+1}(s)$ is upwardly adjacent to $\mathbf{C}_j(s)$ for $j \geq 1$. The sizes of the subsets except for the last one are equal to n.

Then, we can envision reading $\mathbf{C}_1(s)$ using $\gamma_{\text{base}}(s)$ as the address and selecting a context model from $\mathbf{C}_1(s)$ using $\gamma_{\text{incr},0}$ as the index. Reading $\mathbf{C}_1(s)$ and calculating $\gamma_{\text{incr},0}$ are performed simultaneously and their results, n context models in $\mathbf{C}_1(s)$ and the value of $\gamma_{\text{incr},0}$ are latched. BAD takes place in the next cycle, where the proper context model $c \in \mathbf{C}_1(s)$ selected by $\gamma_{\text{incr},0}$ is used as the operand to BAD.

In the following bin decodings, CL does not have to be performed if $\gamma_{\text{incr},i}$ is in $[0,\ n-1]$ because we already have $c_i \in \mathbf{C}_1(s)$ corresponding to $\gamma_{\text{incr},i}$. If $\gamma_{\text{incr},i}$ is greater than $n-1$, another CL is performed to read the next subset $\mathbf{C}_2(s)$. In such cases, i.e., when the required context model is not in the subset held by CMR, the current subset should be updated to the context memory before loading the next subset. This CMR switching causes a pipeline stall of two cycles, one cycle for updating and another for loading the context models. As long as the current subset is effective for the bin decodings, CMR acts as a cache to the datapath.

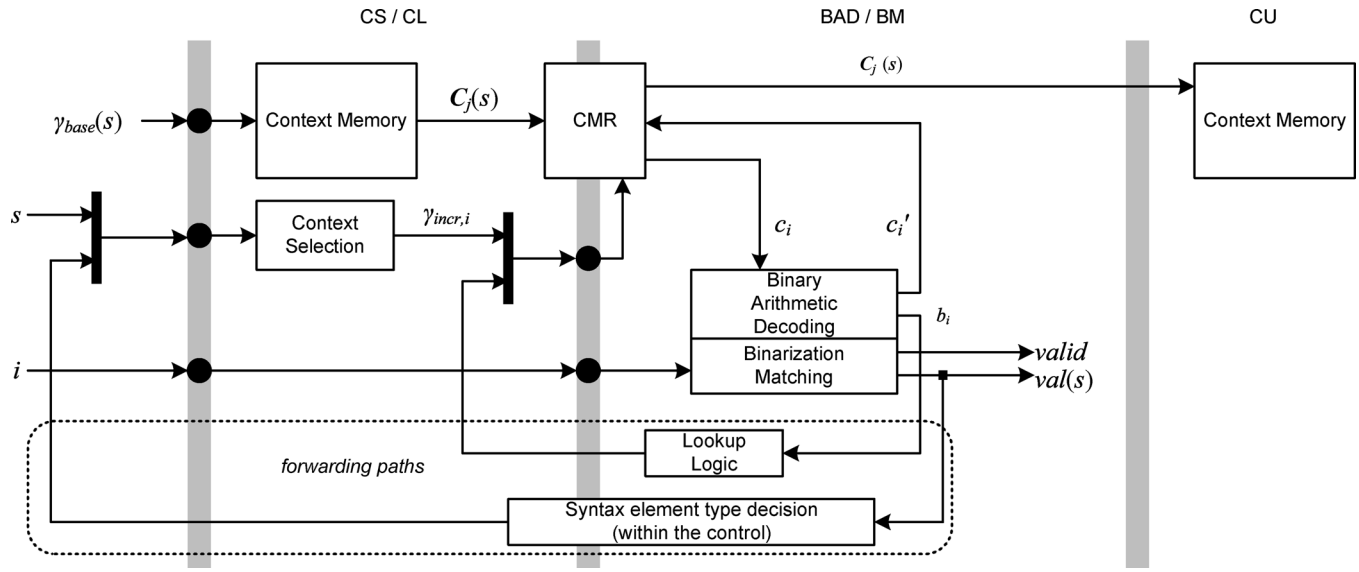

Fig. 7. Essential parts for CABAC decoding with low latency and high throughput. Shaded boxes distinguish pipeline stages.

The implementation of this scheme requires a specific value of n. Since the range of context index increments lies in [0, 7] for most of the syntax elements, we set the value of n to 8. With $n = 8$, context switching arises only for a part of coefficient data; all the syntax elements except the coefficient data have only one context model subset, i.e., $\mathbf{C}_0(s) = \mathbf{C}(s)$. To load the 8 context elements at once, we need a 64-bit-wide memory configuration when the context model, originally requiring 7 bits, is assigned to a byte.

To guarantee that all the context models simultaneously read by CL are in $\mathbf{C}(s)$, the subsets of $\mathbf{C}(s)$ should be aligned to the word boundary in the context memory. Fig. 4(a) shows a context memory layout that is optimal in the sense of memory size required, in which the adjacent context model sets are arranged contiguously. In this arrangement, if syntax element s_2 is to be decoded, it takes two cycles to read $\mathbf{C}_1(s_2)$. In the modified layout depicted in Fig. 4(b), all the elements in each subset can be read in one cycle. This modification results in some vacancies in the memory. The memory overhead is 177 bytes, a 44%-increase from the contiguous arrangement for Main Profile, and 205 bytes, a 45%-increase for High Profile.

Fig. 5 shows the structure of CMR. CMR has eight registers to hold the context models in the read subset $\mathbf{C}_j(s)$. One of these context models is selected by $\gamma_{\mathrm{incr},i}$. At the time of context switching, the values of the registers are concatenated into an altered subset $\mathbf{C}'_j(s)$ which is updated to the context memory.

Consequently, we have a two-stage pipeline with an additional stage that is conditionally executed as illustrated in Fig. 6. When a new syntax element is to be decoded, the pipeline is stalled for two cycles as depicted in Fig. 6(a). Within a syntax element decoding, a syntax element with n bins is decoded in $n + 2$ cycles if there is no CMR switching. However, as exemplified in Fig. 6(b), if CMR switching occurs, the decoding is stalled for two cycles to update and load the context models. Fig. 7 shows the implementation of the pipeline. The two forwarding paths, one for the context selection based on previously decoded bin values and another for the syntax element type decision, minimizes the possibility of stalls.

For the equiprobable decoding and the terminate decoding, the CS/CL stage and the CU stage are inactivated because context models are unnecessary. The binary arithmetic decoding block in Fig. 7 is designed to perform those kinds of decodings as well, in which the decoding takes n cycles for a syntax element with n bins.

IV. Experimental Results

The H.264/AVC syntax parsing system, including a RISC and the proposed CABAC decoder, is implemented in a hardware description language. Since there is currently no available literature that quantitatively reports the decoding performance, we compare the performance of our architecture with a conventional implementation which does not exploit the proposed schemes, that is, the elementary operations in Fig. 1(a) are not pipelined and the memory organization of Fig. 4(a) is used.

Table I summarizes the decoding performances of both architectures, all measured in RTL simulations. All the test streams have the resolution of 1920×1080, the chroma format of 4:2:0, and the frame rate of 25 fps except *riverbed* and *tractor* that have the frame rate of 30 fps. All the sequences are encoded in Main Profile at Level 5.0. The columns of *Decoded bins* and *Cycles* show the average number of symbols and the number of decoding cycles to process one second of coded video streams.

As shown in Table I, the uses of pipelining and parallel processing result in almost two times speedup on the average compared to the conventional architecture. The number of cycles includes the time for the RISC to process the parameter sets and slice headers, the context memory initialization time for each slices, and the macroblock initialization time. The cycles also include the time taken by equiprobable symbol decodings and terminate decodings. Thus, the actual throughput solely used for the symbol decoding is higher than specified.

TABLE I
IMPROVEMENT OF DECODING PERFORMANCE USING THE PROPOSED SCHEMES

Test Sequence	Initial QP	Bitrate (Mbps)	PSNR (dB)	Decoded bins	Conventional Scheme Cycles	Conventional Scheme Cycle/Bin	Proposed Scheme Cycles	Proposed Scheme Cycle/Bin	Speedup
station	24	13.6	41.8	14,661,786	111,096,959	7.58	58,983,985	4.02	1.88
sunflower	24	19.9	45.2	20,196,679	134,534,514	6.66	77,964,440	3.86	1.73
blue_sky	24	20.3	44.4	26,155,172	174,340,104	6.67	95,642,981	3.66	1.82
riverbed	24	20.4	36.0	29,705,109	176,851,196	8.05	96,035,505	3.23	1.84
rush_hour	24	20.4	43.3	26,339,687	185,476,365	7.04	100,435,452	3.81	1.85
pedestrian_area	12	34.5	44.3	41,178,091	307,431,549	7.47	171,300,859	4.16	1.79
tractor	12	63.3	44.2	65,369,918	556,090,113	8.51	309,853,411	4.74	1.79
Average						7.43		3.93	1.81

TABLE II
SYNTHESIS RESULTS

Technology	0.18-μm standard CMOS
Max. Frequency	225MHz
Equivalent Gate Count	81,162 gates
Context Memory	662 Bytes
Data Memory	11.52 KBytes

Table II summarizes the synthesis results. The critical path delay of 4.42 ns arises from the path beginning at a state register in the control, going through the context selection logic, and ending at the context index register within the datapath. According to Tables I and II, the decoder can accommodate the bitrate as high as 45 Mbps with 0.18-μm CMOS technology. Compared to using a general-purpose processor, which requires a 3-GHz performance, the decoder processes the high resolution video sequences while running 13 times slower. Ateme's CABAC encoder is capable of producing 37.5 Mbps coded bitstream at the operating frequency of 150 MHz [15]. The performance of the proposed decoder and that of the commercial encoder are almost the same. However, the decoding process is more complex and has more dependency because it cannot anticipate the syntax ahead while the encoder has the information.

V. CONCLUSION

A CABAC decoder that is fast enough to decode a HD H.264/AVC sequence in real-time under relatively low operating frequency has been proposed. To achieve the speed-up, the pipelining scheme is exploited. Since the data dependencies inherent in the CABAC decoding procedure lowers the benefit of pipelining by a considerable amount, additional parallel processing techniques are proposed as well. The key to the pipelining is to process context selection and context model loading in parallel. In spite of the natural data dependency lying between the context selection and the context model loading, the two operations are overlapped by taking advantage of the characteristics of context models and the corresponding data arrangement. As a result, the proposed CABAC decoder has an average throughput of 3.93 cycles/bin at the slice level.

REFERENCES

[1] D. Marpe, H. Schwarz, and T. Wiegand, "Context-based binary arithmetic coding in the H.264/AVC video compression standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 7, pp. 620–636, Jul. 2003.
[2] *Text of ISO/IEC 14496 10 Advanced Video Coding 3rd Edition*, ISO/IEC JTC1/SC29/WG11 and ITU-T SG16 Q.16, 2004, .
[3] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRE*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.
[4] G. G. Langdon, Jr, "An introduction to arithmetic coding," *IBM J. Res. Develop.*, vol. 28, no. 2, pp. 135–149, Mar. 1984.
[5] G. G. Langdon, Jr. and J. J. Rissanen, "A simple general binary source code," *IEEE Trans. Inf. Theory*, vol. IT–28, no. 5, pp. 778–779, Sep. 1982.
[6] W. B. Pennebaker, J. L. Mitchell, G. G. Langdon, and R. B. Arps, "An overview of the basic principles of the Q-coder adaptive binary arithmetic coder," *IBM J. Res. Develop.*, vol. 32, pp. 717–726, 1988.
[7] J. Rissanen and K. M. Mohiuddin, "A multiplication-free multialphabet arithmetic code," *IEEE Trans. Comput.*, vol. 37, no. 2, pp. 93–98, Feb. 1989.
[8] D. Chevion, E. D. Karnin, and E. Walach, "High efficiency, multiplication free approximation of arithmetic coding," in *Proc. IEEE Data Compression Conf.*, Apr. 1991, pp. 43–52.
[9] B. Fu and K. K. Parhi, "Generalized multipilcation-free arithmetic codes," *IEEE Trans. Commun.*, vol. 45, no. 5, pp. 497–501, May 1997.
[10] L. Bottou, P. G. Howard, and Y. Bengio, "The Z-coder adaptive binary coder," in *Proc. IEEE Data Compression Conf.*, Mar.-Apr. 1998, pp. 13–22.
[11] D. Marpe and T. Wiegand, "A highly efficient multiplication-free binary arithmetic coder and its application in video coding," in *Proc. IEEE ICIP*, Barcelona, Spain, Sep. 2003, vol. 2, pp. 263–266.
[12] R. R. Osorio and J. D. Bruguera, "Arithmetic coding architecture for H.264/AVC CABAC compression system," in *Proc. IEEE EUROMICRO Symp. Dig. Syst. Design (DSD'04)*, Aug.-Sep. 2004, pp. 62–69.
[13] R. Prakasam, A. G. MacInnis, O. Tao, and X. Xie, "Context adaptive binary arithmetic decoding engine," U.S. Patent 20 040 240 559 (USPTO Class 375), 2004.
[14] J.-W. Chen, C.-R. Chang, and Y.-L. Lin, "A hardware accelerator for context-based adaptive binary arithmetic decoding in H.264/AVC," in *Proc. IEEE ISCAS*, May 2005, vol. 5, pp. 4525–4528.
[15] *Ateme MPEG-4 AVC-H.264 CABAC/CAVLC IP Datasheet* [Online]. Available: http://www.ateme.com
[16] Broadcom, BCM7411 Product Brief [Online]. Available: http://www.broadcom.com