A Near Optimal Deblocking Filter for H.264 Advanced Video Coding

Shen-Yu Shih Cheng-Ru Chang Youn-Long Lin

Department of Computer Science
National Tsing Hua University
Hsin-Chu, Taiwan 300
Tel : +886-3-573-1072
e-mail: ylin@cs.nthu.edu.tw

Abstract - We propose a near optimal hardware architecture for deblocking filter in H.264/MPEG-4 AVC. We propose a novel filtering order and a data reuse strategy that result in significant saving in filtering time, local memory usage, and memory traffic. Every 16x16 macroblock requires 192 filtering operations. After a few initialization cycles, our 5-stage pipelined architecture is able to perform one filtering operation per cycle. Compared with some state-of-the-art designs, our architecture delivers the fastest level of performance while using much smaller gate count and memory. We have implemented and integrated the proposed deblocking filter into an H.264 main profile video decoder and verified it with an FPGA prototype.

I. Introduction

H.264/MPEG-4 AVC is an emerging video coding standard [1][2]. Compared with the most popular standard MPEG-2, it can save more than half of the bit-rate. The saving is gained from heterogeneous video coding algorithms, such as multi-mode intra-prediction, multi-frame variable-block-size quarter-pixel-accurate inter-prediction, integer discrete cosine transform (DCT), context adaptive binary arithmetic coding (CABAC), and deblocking filter. One of the most special features in H.264/MPEG-4 AVC is deblocking filter [3]. It is applied to reduce the blocking artifact generated by block-based motion compensated prediction, intra prediction, and integer discrete cosine transform. In H.264/MPEG-4 AVC, the filter for eliminating blocking artifact is embedded within the coding loop. Therefore, it is also called in-loop filter. According to some experiments, it is able to achieve up to 9% bit-rate saving [4] at the expense of large amount of computation. Even with the fastest CPU, it is hard to perform software-based real-time decoding or encoding of high quality video sequences. Consequently, a hardware accelerator is indeed required.

Fig. 1 shows an H.264 main profile decoder proposed by our research laboratory. The Variable-Length deCoding (VLC) module reads in encoded video stream and generates slice-level parameters for several other modules and macroblock-level bit-stream information for the CABAC module. The CABAC module generates syntax elements and stores them into *MBinfo mem* and *Coeff mem*. Then, according to the current slice type, one of either Motion Compensation (MC) or Intra Prediction (Ipred) module is activated to perform compensation. Meanwhile, the Inverse Quantization and Inverse DCT (IQ/IDCT) module reads coefficient data from *Coeff mem* and transforms them back to residuals. The Picture Reconstruction (Pic Rec) module combines the compensated data with residuals. Finally, the Deblocking Filter (DF) module gets reconstructed data to perform filtering and outputs the filtered macroblock to *refMB mem* for reference and display. This paper presents our design and implementation of the DF module.

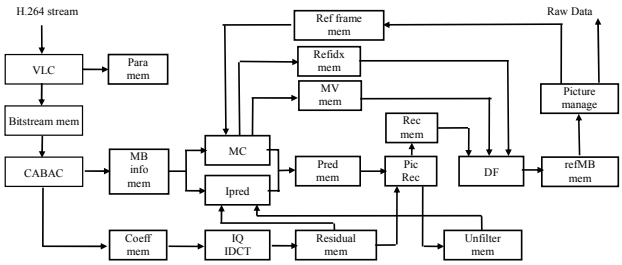

Fig. 1. An H.264 decoder that employees the proposed filter.

The rest of this paper is organized as following. Section II describes the deblocking filter algorithm. Section III presents our hardware architecture in detail. In Section IV, we present our synthesis and FPGA-prototyping results and compare it with previous work. Finally, we draw some concluding marks and point to possible directions for future research in Section V.

II. Deblocking Filter Algorithm

A. Overview

The deblocking filter is used to eliminate blocking artifact and thus generate a smooth picture. The inter prediction module finds a block similar to the current block from reference frames. The found block usually cannot perfectly match with the current block resulting in prediction error. For coding efficiency, the error is DCT-transformed and quantized. After the decoding process, the reconstructed block is different from the original block. Especially, discontinuity is likely to appear at the block edge. To alleviate the degree of discontinuity, the deblocking filter process is applied.

0-7803-9451-8/06/$20.00

Inputs to the deblocking filter include pixels, boundary strength, and threshold values as shown in Fig. **2**. The pixels of a macroblock are filtered by an edge filter in a specific order, and each pixel may be filtered multiple times. After the whole picture is filtered, it is ready for display as well as being a reference picture.

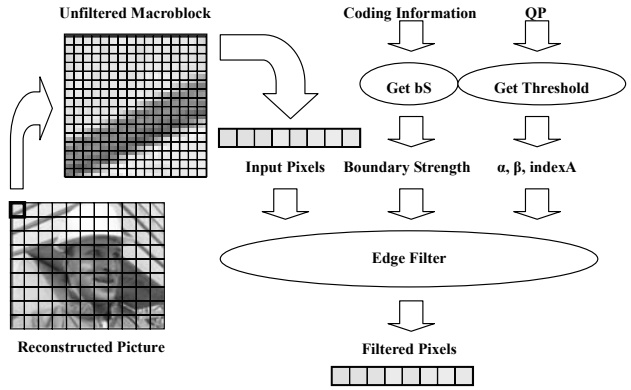

Fig. 2. Inputs to and outputs from the deblocking filter.

B. Filter Order

The deblocking filter process consists of a horizontal filtering across all vertical edges and a vertical filtering across all horizontal edges. Fig. 3(a) illustrates the filtering process for the 16x16 luma component of a macroblock. Each small box denotes a pixel, and a dotted one represents a pixel from neighboring macroblocks. The top part shows the horizontal filtering. Vertical edge 0 is filtered horizontally first from top to bottom, followed by edge 1, edge 2, and edge 3. For luma filtering, the edge filter takes as its inputs eight pixels, p3, p2, p1, p0, q0, q1, q2, and q3. At most 6 pixels will be modified by the filter as shown in the shadowed part of the figure. Because there are overlapping area between the filtering of two adjacent edges, some pixels (actually, half of them) may be filtered twice.

The vertical filtering shown in the bottom part of Fig. 3(a) is performed after horizontal filtering in a similar way. Edge 0 is vertically filtered from left to right, followed by edge 1, edge 2, and edge 3.

The filtering process of chroma components is similar to that of luma components as depicted in Fig. 3(b). It is first horizontally applied on edge 0 from top to bottom, followed by edge 1. After the vertical edges are filtered, the horizontal edges are then filtered from edge 0 to edge 1. Note that unlike a luma edge which is of length 16, a chroma edge is of length 8, and there are only 5 input pixels, p1, p0, q0, q1, and q2 with two possible pixel modifications per filtering.

C. Boundary Strength

The boundary strength (bS) is derived from the coding information [5] of the macroblock. Two adjacent 4x4 blocks share a bS value. Its value ranges from 4 to 0, 4 for the strongest filtering and 0 for no filtering. Fig. **4** gives a flowchart for calculating bS value. If any one of the two adjacent 4x4 blocks is coded with intra prediction mode and they are on the macroblock edge, the bS is set to 4. If any one of them is intra-coded and they are not on the MB edge, the bS is 3. If any of them contains non-zero transform coefficients, the bS is set to 2. Finally, if different reference pictures are used or the difference between two motion vectors of the two blocks is greater than or equal to 4 in units of quarter pixels, the bS shall be equal to 1. For the remaining cases, the bS is set to 0.

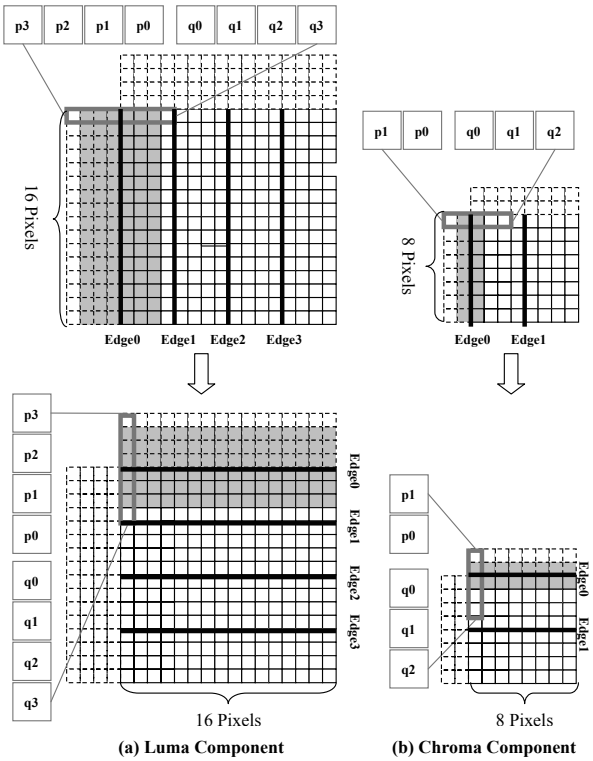

Fig. 3. Horizontal filtering and vertical filtering of luma component (a) and chroma component (b).

D. Threshold

Three threshold variables, α, β, and indexA, are used to prevent true edges from being filtered. Their values depend on the quantization parameters as described in [2]. The flag *filterSamplesFlag* is used to decide whether the filtering process should be carried out. It is set to true if (1) is true.

$$bS!=0 \;\&\&\; |p0-q0|<\alpha \;\&\&\; |p1-p0|<\beta \;\&\&\; |q1-q0|<\beta \qquad (1)$$

E. Edge Filter

The edge filter starts to filter when the input pixels, boundary strength, and threshold variables are ready. First, if the flag *filterSamplesFlag* is equal to 1, the current edge is very likely to be a blocking artifact instead of a true edge. Thus, the filtering process should be applied. If the filtering process needs to be performed, there is a branch depending

on the value of bS. If bS is smaller than 4, there are at most 4 pixels to be modified. Otherwise, there are at most 6 pixels to be modified. The detailed filtering operations are listed in [2].

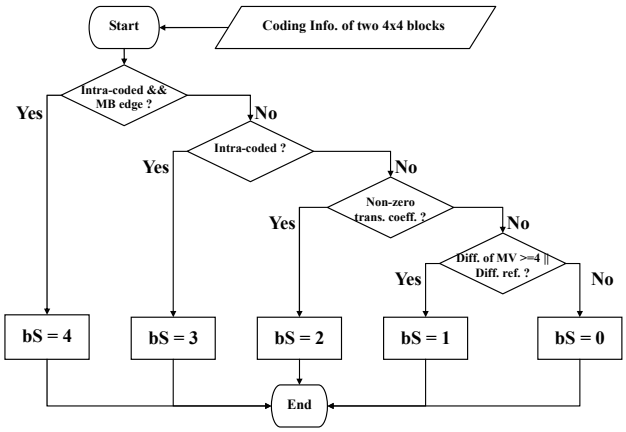

Fig. 4. Flowchart of boundary strength calculation.

III. Proposed Architecture

A. Deblocking Filter Architecture

Fig. **5** gives a top view of our deblocking filter architecture. The blocks outside the dotted box are local memories of our H.264 decoder. Through these memories or buffers, the deblocking filter gets data produced by other modules in different pipelined stages. For example, *reconstruct mem* stores the reconstructed pixels combining the data from the motion compensation unit and the IQ/IDCT unit. The coding information in such memories as *Ref idx mem*, *MV mem*, *Para mem*, *MBinfo mem* is used for calculating the boundary strength. After the filtering process completes, the output data is written back to *refMB mem*.

Inside the dotted box is our implementation of the deblocking filter. The module *Generate bS & Threshold* fetches data from external memories to calculate bS and threshold values. Two local memories *local mem 0* and *local mem 1* are used for storing pixels from neighboring macroblocks. Two transpose registers, *T0* and *T1*, are used for buffering and transposing pixels. In the center of the dotted box is the edge filter with 5 pipeline stages. After the pixels are filtered, the results will be written out via the *Write Back Unit*.

B. Local Memory Organization

Fig. **6** shows our memory organization. There are three local memory modules. The pixels of the currently under-filtered macroblock are stored in *reconstruct mem*. The two-port SRAM, *local mem 0*, stores the intermediate results of filtering process. For data reuse, we use a single-port SRAM, *local mem 1*, to buffer a frame-wide row of 4x4 blocks. Note that chroma filtering requires only half of 4x4 block. The size of *local mem 1* depends on the frame width. For example, for CIF video, the memory is (1.5x352) x 32 bits.

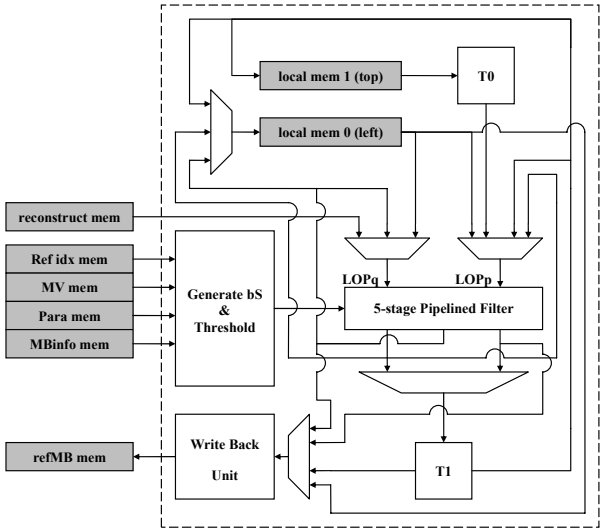

Fig. 5. Proposed deblocking filter architecture.

C. Filtering Order

Our filtering order is illustrated in Fig. **7**. Each circle stands for a step with 4 cycles. In order to preserve the rightmost column (i.e., B3, B7, B11 and B15 for luma component) of the current macroblock for the filtering of the next macroblock, we filter the edges from the left to right.

In Step 1, blocks L0 and B0 are read from *local mem 0* into the edge filter. In Step 2, Blocks L1 and B4 are filtered while Block B0 is stored back to *local mem 0*, and Block L0 is written out via *Write Back Unit*. In Step 3, Blocks L2 and B8 are filtered while Block B4 is stored back to *local mem 0*, and Block L1 is written out via *Write Back Unit*. In Step 4, Blocks B0, B4, B8, and B12 are filtered horizontally, and Block LT0 is loaded into transposed register *T0*. Note that Block B12 is still in the pipelined filter. In Step 5, blocks B0 and B1 are horizontally filtered. In Step 6, transposed blocks LT0 and B0 are vertically filtered. With the proposed filtering order, we can filter a macroblock in 192 cycles, which is optimal.

D. Pipelined Filter

Fig. **8** depicts our 5-stage pipelined filter architecture. Stage 1 reads pixels from various memories. Stage 2 calculates such parameters as *filterSamplesFlag* described in Section II. Stage 3 filters pixels with bS equal to 4. In Stage 4, pixels with bS equal to 3, 2, or 1 are filtered, and clipping performed. Finally, Stage 5 stores filtered pixels back to memory or transpose registers.

Multiplexers are added to resolve pipeline hazards. Let's take filtering Step 5 and Step 6 shown in Fig. 9 as an example. Filtering Step 6 requires the transposed pixels of Block B0 and LT0. However, pixels of Block B0 are still in

the pipeline. Therefore, we add some forwarding logic to get register transferring behavior as illustrated in Fig. 10.

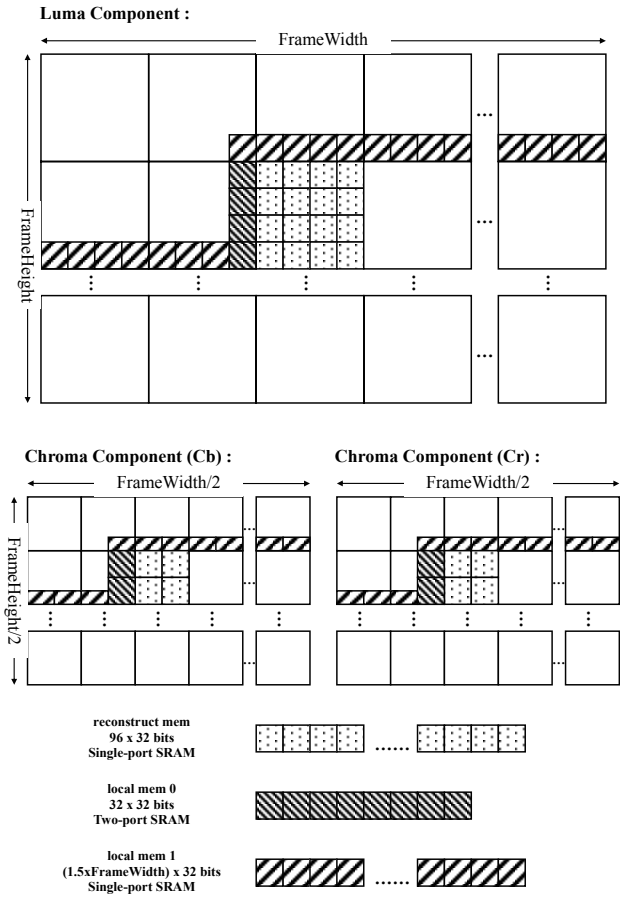

Fig. 6. Local memory organization.

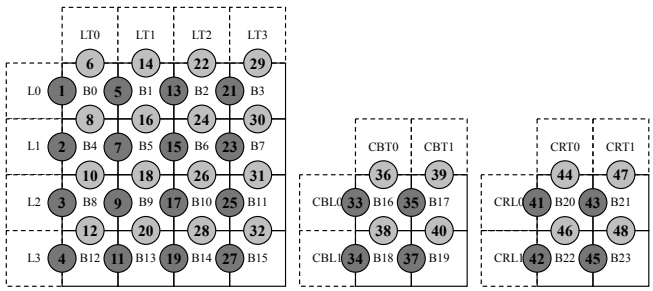

Fig. 7. Proposed filtering order.

When we begin to do the first filtering (Line 5) of Step 6, we need pixels of Line 1. As described in Sub-Section C, we have put Block LT0 into transpose register *T0* in Step 4, and thus each column of Block LT0 can be read. To get the lower part of Line 5, which is still inside the pipe stage, we insert a forwarding logic to select pixels marked with a00, a10, a20, a30 from different pipeline stages.

At the next cycle, parameters such as *filterSamplesFlag* have been calculated. Note that it requires 8 pixels, a30, a31, a32, a33, b30, b31, b32, and b33, to perform filtering for block B0 and B1 in Stage 3. The forwarding pixels are not fed back to the edge filter. Instead, it is directly output to the transpose register *T1* or *Write Back Unit* in some cases such as Step 7. We use the register *p3fwd* as shown in Fig. 8 to keep the required pixel, a30.

The multiplexers in the data path depicted in Fig. **8** denoted the forwarding paths.

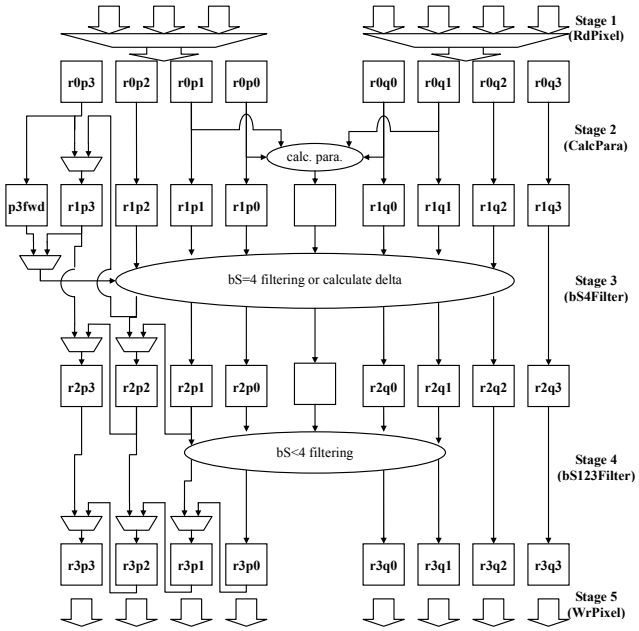

Fig. 8. Proposed 5-satge pipelined edge filter.

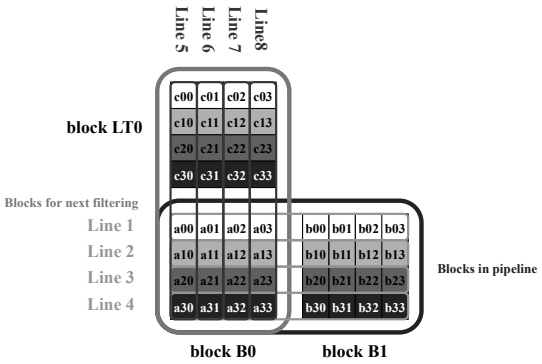

Fig. 9. Pipeline hazard illustration.

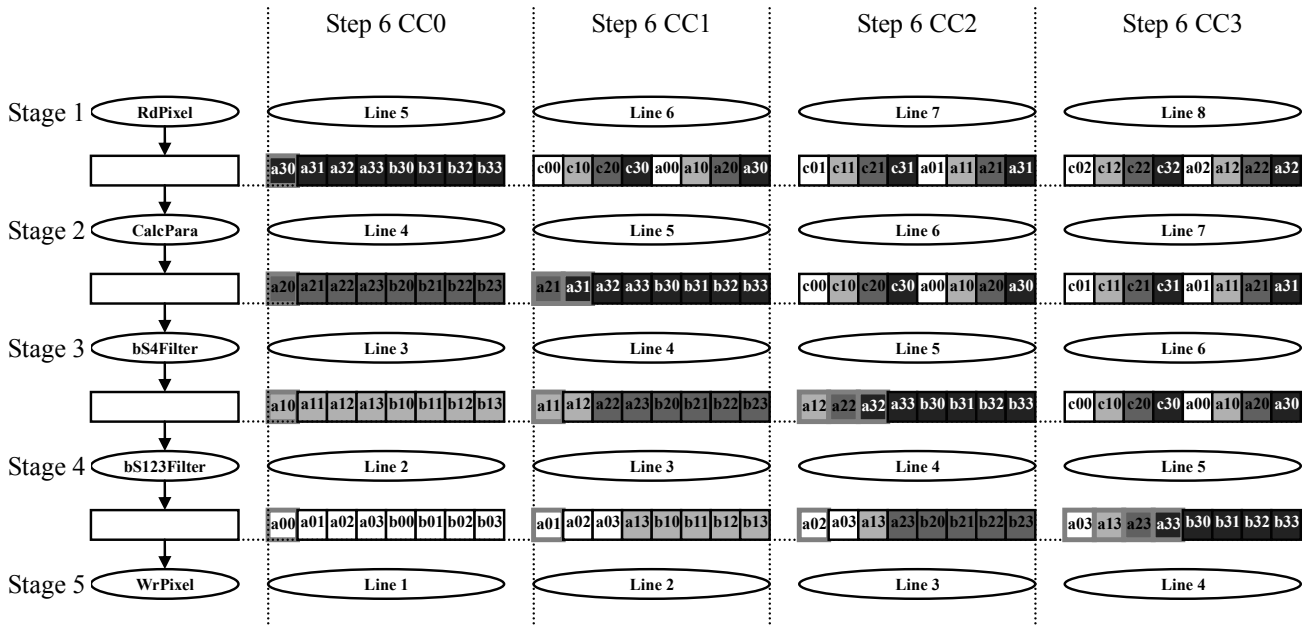

Fig. 10. Pipeline forward for filtering step 6.

IV. Experimental Results

We have implemented the proposed deblocking filter architecture in Verilog RTL and verified its integration with an H.264 decoder with FPGA prototyping. We synthesize our design using Synopsys Design Compiler targeted towards an Artisan UMC 0.18um cell library. The clock frequency is set to 100MHz.

Fig. 11 analyzes the number of required processing cycles of the proposed architecture. At the beginning, we take 14 cycles to read coding information [5] necessary for filtering the first pair of 4x4 blocks. It takes 192 cycles for both filtering and writing out processed pixels. This is optimal because there are exactly 192 filtering operations needed according to the following calculation. For the luma component, in each of the vertical and horizontal filtering process, there are 4 edges each requires 16 filtering operations. Therefore, we need 2x4x16 = 128 filtering operations. For the two chroma components, the number is 2x2x2x8 = 64. Therefore, the total is 128+64 = 192.

After initialization, the calculation of boundary strength is overlapped with filtering. After filtering, we need 8 extra cycles for flushing the pipeline. If the current macroblock is not the rightmost macroblock of a row of the picture, filtering one MB requires 14+192+8 = 214 cycles; otherwise, 32 additional cycles are needed to write out the rightmost column of 4x4 blocks to the external memory. Taking these 32 cycles into account, we need 246 cycles to filter one MB in the worst case. The average number of cycles per MB for a video sequence in CIF format is about 216 cycles.

Table I compares our work with some state-of-the-art designs. In terms of total number of cycles needed per MB, ours is better than every one. References [7][9][10] do not give the numbers while Reference [8]'s numbers varies due to skip mode implementation. In terms of the number of cycles spent in the kernel filter, 192 is the optimal. Reference [7] reduces it to 136 by employing two filters. From the table, we can conclude that most of the work can do very well in the kernel filter design. However, it is the data transportation that makes the difference. Most of the design spends more time in data transport than filter operation. By means of overlapping filtering and data transport, we are able to achieve near optimal performance. We use less local memory because only one half of chroma blocks are stored. Our hardware design can easily meet the requirements for real-time decoding of video sequences with 1280x720p and 30fps resolution.

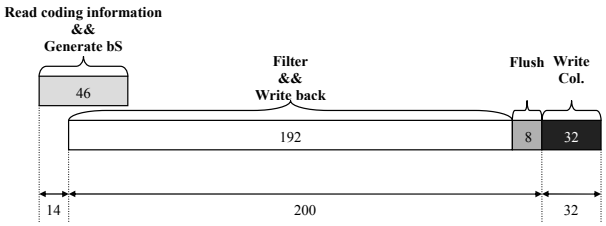

Fig. 11. Processing cycles analysis.

TABLE I Comparisons among various deblocking filters

	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	Proposed
Cycles/MB	614	566	N/A	Max:342 Min:50 Avg.:86-241[3]	N/A	N/A	446	238[6]	250	214 or 246[9]
Filtering Cycles/MB	240	192	136	0-342	214[5]	336	192	236	250	192
SRAM for Pixels[1]	2P 96x32 2P 64x32	8 DP 80x8	DP 88x32 DP 72x32 1P 32x32	1P 96x32	DP 16x32	1P 80x32	DP 64x32 2 2P 96x32	DP 96x32 DP 64x32 1P (2xFW[7])x32	2 1P 96x32 1P (2xFW[7])x32	1P 96x32 2P 32x32 1P (1.5xFW[7])x32
# of 4x4 Arrays	4	0	11	2	6	2	8	9	4	2
# of Edge Filters	1	1 Pipelined	2	1	1	1	1	1	1	1 Pipelined
Process(um)	.25	.35	N/A	.18	N/A	.18	.25	.18	.18	.18
Gate Count[2]	20.66K	9.35K	N/A	11.8K[4]	N/A	9.16K[4]	24K	14.5K[8]	19.64K	20.9K

1. DP : Dual-port SRAM with two R/W ports; 1P : Single-port SRAM with one R/W port; 2P : Two–port SRAM with one read and one write ports
2. Gate Count does not include SRAM
3. The performance is evaluated by QCIF video sequences with 1I+149P
4. The gate count does not include boundary strength calculation logic and coding information registers
5. The filtering cycles do not include filtering chroma components
6. The cycles do not include boundary strength calculation time
7. FW stands for Frame Width
8. The gate count does not include coding information registers
9. It takes 246 cycles to filter one MB at right picture boundary

V.Conclusions

We have proposed a near optimal architecture for deblocking filter in H.264/AVC. We implemented the design in synthesizable Verilog RTL and verified it with reference software [14]. The result shows that the performance of our design is near optimal but the usage of local memory is less compared with previous work. Besides, with a pipelined architecture, our design can achieve higher performance with increasing clock frequency. We have integrated the hardware accelerator into our H.264 decoder and verify it on a FPGA development board. The result shows that our design works correctly and the performance for decoding greatly increases compared with pure software solution or platform-based methodology.In the future, we will work on reducing the power consumption of our decoder. Meanwhile, we will use the deblocking filter in the development of both H.264 encoder and CODEC.

Acknowledgement

This research is supported in part by the National Science Council of Taiwan, the Ministry of Economic Affairs of Taiwan, and Taiwan Semiconductor Manufacturing Company under grant no. NTHU-0416.

References

[1]T. Wiegand, G. J. Sullivan, G. Bjntegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 13, pp. 560-576, 2003.

[2]"Draft ITU-T recommendation and final draft international standard of joint video specification (ITU-T Rec. H.264 | ISO/IEC 14496-10 AVC)," JVT G050, 2003.

[3]A. Luthra, G. J. Sullivan, and T. Wiegand, "Introduction to the special issue on the H.264/AVC video coding standard," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 13, pp. 557-559, 2003.

[4]P. List, A. Joch, J. Lainema, G. Bjntegaard, and M. Karczewicz, "Adaptive deblocking filter," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 13, pp. 614-619, 2003.

[5]Y. W. Huang, T. W. Chen, B. Y. Hsieh, T. C. Wang, T. H. Chang, and L. G. Chen, "Architecture design for deblocking filter in H.264/JVT/AVC," *IEEE Int'l Conf. on Multimedia and Expo*, 2003.

[6]L. Li, S. Goto, and T. Ikenaga, "An efficient deblocking filter architecture with 2-dimentional parallel memory for H.264/AVC," *Asia South Pacific Design Automation Conf.*, 2005.

[7]V. Venkatraman, S. Krishnan, and N. Ling, "Architecture for deblocking filter in H.264," *Picture Coding Symposium*, 2004.

[8]S. C. Chang, W. H. Peng, S. H. Wang, and T. Chiang, "A platform based bus-interleaved architecture for deblocking filter in H.264/MPEG-4 AVC," *IEEE Trans. on Consumer Electronics*, vol. 51, pp. 249-255, 2005.

[9]M. Sima, Y. Zhou, and W. Zhang, "An efficient architecture for adaptive deblocking filter of H.264/AVC video coding," *IEEE Trans. on Consumer Electronics*, vol. 50, pp. 292-296, 2004.

[10] C. C. Cheng, and T. S. Chang, "An hardware efficient deblocking filter for H.264/AVC," *IEEE Int'l Conf. on Consumer Electronics*, pp. 235–236, 2005.

[11] B. Sheng, W. Gao, and D. Wu, "An implemented architecture of deblocking filter for H.264/AVC," *IEEE Int'l Conf. on Image Processing*, vol. 1, pp. 665–668, 2004.

[12] G. Zheng, and L. Yu, "An efficient architecture design for deblocking loop filter," *Picture Coding Symposium*, 2004.

[13] T. M. Liu, W. P. Lee, T. A. Lin, and C. Y. Lee, "A memory-efficient deblocking filter for H.264/AVC video coding," *IEEE Int'l Symposium on Circuit and Systems*, 2005.

[14] JVT H.264/AVC Reference Software JM 8.3