A Comparison of 2-D Discrete Wavelet Transform Computation Schedules on FPGAs

Maria Angelopoulou[1], Konstantinos Masselos[1], Peter Cheung[1], Yiannis Andreopoulos[2]

[1]*Department of Electrical and Electronic Engineering, Imperial College London*
Exhibition Road, London SW7 2BT, UK
{m.angelopoulou, k.masselos, p.cheung}@imperial.ac.uk

[2]*Department of Electrical Engineering, University of California Los Angeles*
54-147 Eng. IV Building, 420 Westwood Plaza, Los Angeles, CA 90095-1594, USA
yandreop@ee.ucla.edu

***Abstract*— When it comes to the computation of the 2-D Discrete Wavelet Transform (DWT), three major computation schedules have been proposed, namely the row-column, the line-based and the block-based. In this work, the lifting-based designs of these schedules are implemented on FPGA-based platforms to execute the forward 2-D DWT, and their comparison is presented. Our implementations are optimized in terms of throughput and memory requirements, in accordance with the specifications of each one of the three computation schedules and the lifting decomposition. All implementations are parameterized with respect to the image size and the number of decomposition levels. Experimental results prove that the suitability of each implementation for a particular application depends on the given specifications, concerning the throughput and the hardware cost.**

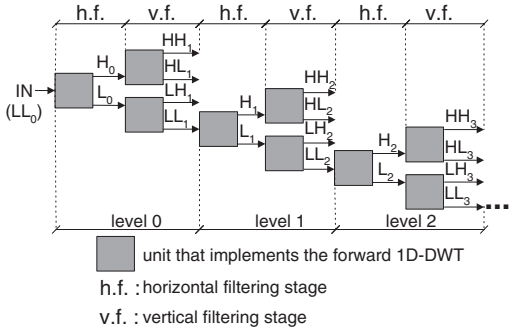

Fig. 1. The 2-D DWT decomposition.

I. Introduction

The two-dimensional Discrete Wavelet Transform (DWT) is a key operation in image processing, and is the kernel of both the JPEG-2000 still image compression standard [1] and the MPEG-4 still texture decoding standard [2]. The 2-D DWT is carried out by applying the 1-D DWT in both the horizontal and the vertical direction of the image. As shown in Fig. 1, each unit that executes the 1-D DWT produces two sets of coefficients: a low-frequency and a high-frequency set. The outputs of a horizontal filtering stage are vertically filtered to produce the 2-D subbands LL, LH, HL and HH. All LH, HL and HH coefficients are stored, to contribute later in the reconstruction of the original image from the LL set. The LL coefficients will either be the input of the horizontal filtering stage of the next level, if there is one, or will be stored as well, if the current level is also the last one.

The traditional convolution-based 1-D DWT [3] imposed high computational complexity. The *lifting scheme* ([4], [5]) overcomes this problem by factorizing the polyphase matrix of the DWT into elementary matrices.

For the implementation of the 2-D DWT, several computation schedules have been proposed. In practical designs, the most commonly used computation schedules are: the *row-column* (RC) [3], the *line-based* (LB) [6] and the *block-based* (BB) [7]. The simplest of these is RC, which adopts the level-by-level logic of Fig. 1. However, such an approach necessitates the use of large memory blocks, distant from the computational units, as the only source of the filter's inputs. Contrary to RC, both LB and BB involve an on-chip memory structure that operates as a cache for the original image, minimizing the accesses of the large memory blocks. Thus, memory utilization and memory-access locality are improved. The main difference among LB and BB concerns the way the original image is traversed. Specifically, in LB, non-overlapping groups of lines are processed, whereas, BB operates using non-overlapping blocks of the image.

In [8], [9] and [10] 2-D DWT computation schedules have been compared on a theoretical basis. In [11] and [12], they are compared on programmable architectures and on a VLIW DSP, respectively. Even though the above comparisons are particularly enlightening, none of them is based upon hardware implementations. Thus, the implementations involved do not take advantage of the implementation efficiency and the parallelism in data processing that hardware could offer. In addition, the vast majority of comparisons of the different alternatives focuses on convolution-based realizations and lifting is not considered.

Contribution of this paper—In this paper, the three major 2-D DWT lifting-based computation schedules are implemented on FPGA-based platforms and compared in terms of performance and area. The computation schedules are compared for different image sizes (M*M) and number of levels (L) of the transform. To the best of our knowledge no comparisons of detailed hardware implementations of

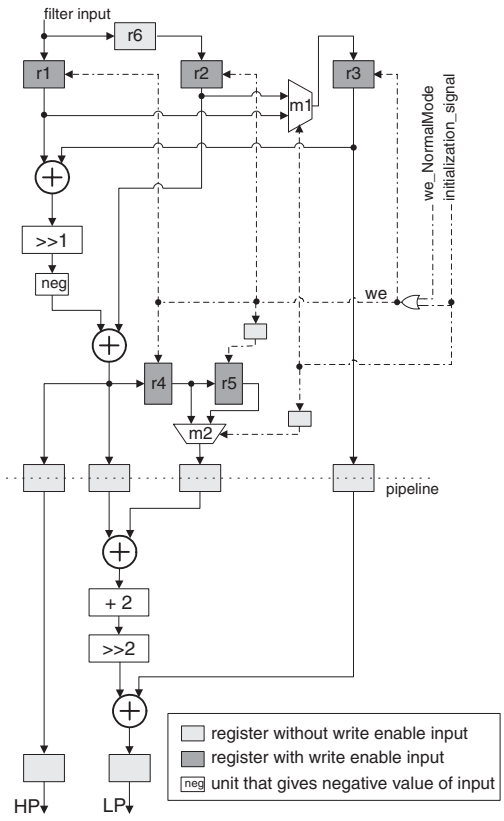

Fig. 2. Hardware implementation of the 5/3 lifting filter, designed to perform the 1-D DWT.

the three major 2D-DWT computation schedules exist in literature. This comparison will give significant insight on which schedule is most suitable for given values of the relevant algorithmic parameters.

Structure of this paper—Section II presents decisions we made that are, for comparison reasons, common in all our FPGA implementations. Sections III, IV and V describe how we implemented RC, LB and BB, respectively. In section VI we discuss the results of the three FPGA implementations. Finally, section VII offers our conclusions.

II. Common Implementation Decisions and Assumptions

The comparison of the main 2-D DWT computation schedules is performed on the basis of some common implementation decisions and assumptions. These are concerned with the memory block that stores the image (*image memory*) and the filtering structure used to compute the DWT.

A. Image memory

In this comparison, a single-port image memory is considered. The image memory is usually off-chip.

Traditionally, two memory blocks are used in image-processing systems: one to store the original image, and one

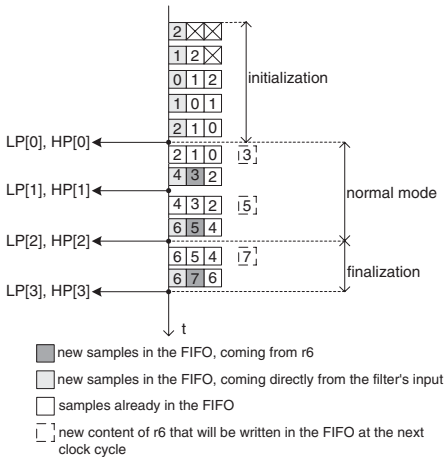

Fig. 3. The contents of the FIFO in respect to time for the filtering of an 8-pixel line.

for the outputs. To avoid the second block, we used the in-place mapping scheme: the filter's outputs are written over memory contents that are already consumed and no longer needed. To adopt this scheme, each memory location should have the same bit-width as the outputs of the transform.

B. Filter implementation

A single filter is used in all three implementations, introducing the minimum hardware cost. That is, the same single filter is *shared* among all levels and also among the vertical and the horizontal filtering stages within each level. The choice of using a single filter is mostly appropriate for an RC architecture that uses a single-port RAM.

We used a 5/3 lifting filter [5], as it involves less computational load than FIR or 9/7 lifting filters [4]. The lifting equations of this filter are the following :

$$HP[2n+1] = X[2n+1] - \lfloor \frac{X[2n] + X[2n+2]}{2} \rfloor \quad (1)$$

$$LP[2n] = X[2n] + \lfloor \frac{HP[2n-1] + HP[2n+1] + 2}{4} \rfloor \quad (2)$$

where X are the signal samples, HP is the high-frequency output coefficient, LP is the low-frequency output coefficient, and $\lfloor . \rfloor$ represents the *floor* operator. The floor operator ensures an integer-to-integer lossless transform.

In the hardware implementation of the filter, a three-word FIFO will store inputs $X[2n+1]$, $X[2n]$, $X[2n+2]$, and a register will store $HP[2n-1]$, which was calculated in the previous lifting step. According to the above equations, in order for a new pair of output coefficients to be computed, two (and not one) new filter inputs are needed. Thus, a filtering operation will take place, and a pair of output coefficients will be produced, every two cycles.

Our hardware implementation of the 5/3 lifting filter is shown in Fig. 2. Registers r1, r2 and r3 constitute the FIFO. Observing Fig. 3, one concludes that the filter's behavior is

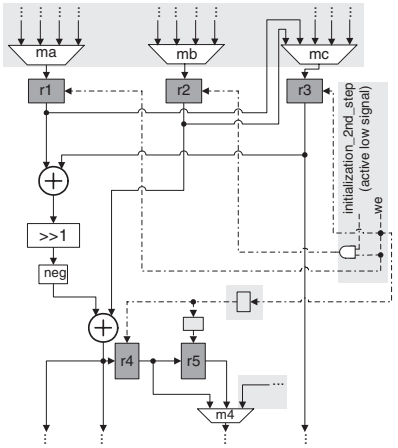

Fig. 4. This filter derives from that of Fig. 2, by applying a few changes (the shaded areas), to incorporate a multiple-input function. The lower part is not shown, as it remains the same.

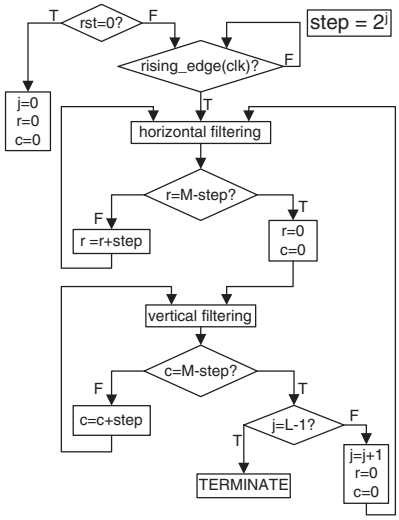

Fig. 5. Flowchart of the RC algorithm as implemented (r/c = current row/column, j = current level).

determined according to whether the initialization phase is over. Thus, the filter might behave in one of the two following ways:

1) The *initialization mode*, where the data flows in the FIFO with a step of one. During the initialization phase, two samples are mirrored around the first sample, and a simple shifting takes place in the FIFO, as shown in Fig. 3.
2) The *normal mode*, where the data flows with a step of two. After the initialization phase has been over, the FIFO is written every two clock cycles and a filtering operation is executed every second cycle. Thus, the pattern shown in Fig. 3 is achieved. The same pattern applies for the finalization mode, but the source of r1's input is now the FIFO itself, eliminating the need to access memory during this step.

In RC, only one input enters the filter per cycle, since the single-port image memory is the only source of input coefficients. Thus, for RC, the filter of Fig. 2 is ideal as it is. On the contrary, as we will see in the following sections, both LB and BB involve multi-port on-chip buffers, that can supply the filter with more than one inputs per cycle. In order to make the best possible use of the parallelism offered, we made a few small changes to the filter of Fig. 2, to incorporate a multiple-input function (Fig. 4). Now, two or three inputs can be inserted in the filter at a single cycle. However, during the horizontal filtering at level 0, the samples will be drown from the single-port memory, just like in the case of RC. Thus, during the horizontal filtering at level 0, the new filter behaves in a single-input mode, apart from the multiple-input mode in which it functions in any other case.

III. Row-column Implementation

The RC is implemented by applying the forward 1-D DWT in both the horizontal and the vertical direction of the image, for a chosen number of levels, in the way shown in Fig. 1.

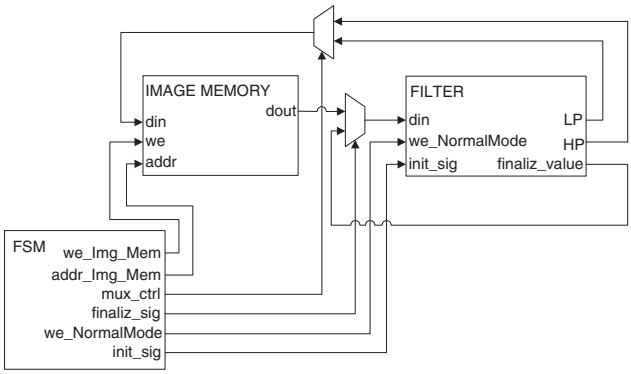

Fig. 6. Block diagram of the RC architecture.

Specifically, in any given level, in order to proceed to the vertical filtering, of the current level's LL image block, the horizontal filtering should have been completed. In addition, in order to proceed to the next level, the filtering at the previous level should have finished. Figures 5 and 6 present the flowchart and the block-diagram of RC, as implemented.

The RC architecture is the one with the simplest control path. The parallelism achieved during the filtering operations depends on the number of ports of the image memory. Its major disadvantage is the lack of locality, due to the use of large memory blocks, distant from the computational units. This decreases the performance.

IV. Line-based Implementation

The flowchart of LB, as implemented, is presented in Fig. 7. The LB uses on-chip buffers (Fig. 8, Fig. 9), to store coefficients of intermediate levels which will be used at subsequent levels. This improves the memory-access locality compared to RC, and improves, hence, the performance. Moreover, contrary to RC, where the single-port image memory imposes a serial

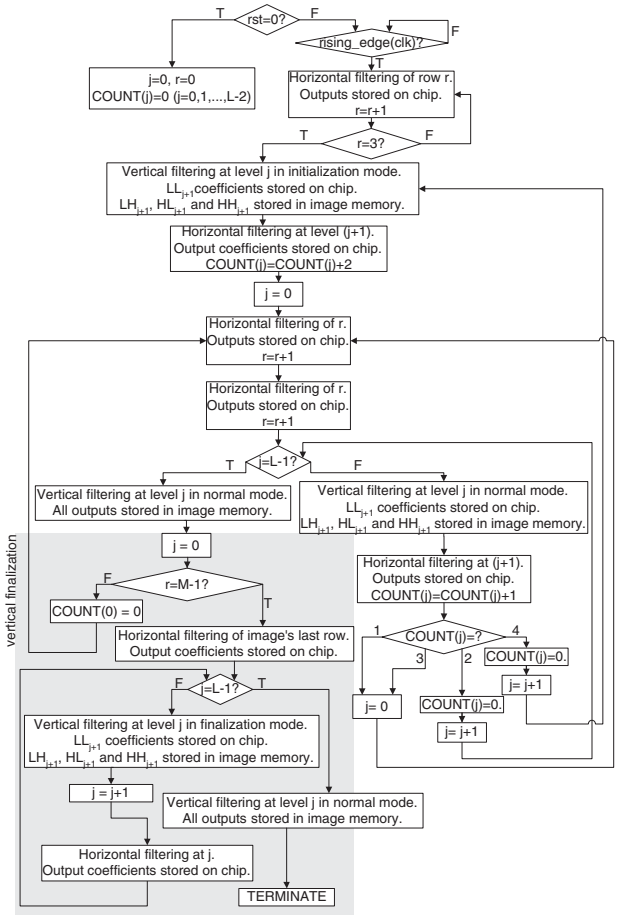

Fig. 7. Flowchart of the LB algorithm as implemented (r = current row, j = current level).

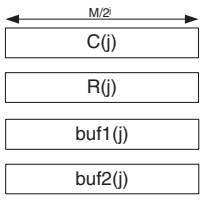

Fig. 8. On-chip line buffers of level j, used in LB.

nature to the filtering operations, these buffers may be multi-port to increase parallelism.

A group of lines is processed up to the final level, and a filter's output is stored in image memory only if it will be used as it is during reconstruction and never again during decomposition. Thus, LH, HL and HH coefficients are stored after the vertical filtering at any level, whereas the LL coefficients are stored only at the last level (Fig. 7).

After the completion of a vertical filtering at level $j-1$, the resulting LL_j coefficients are written in R(j). Buffer R(j) will then be horizontally filtered and the resulting coefficients will be written, depending on the current stage of level j's vertical filtering, in one of the following: C(j), R(j) (implementing the in-place mapping scheme) or buf1(j) (only during the initialization phase of the vertical filtering at level j, when buf1(j) is still empty).

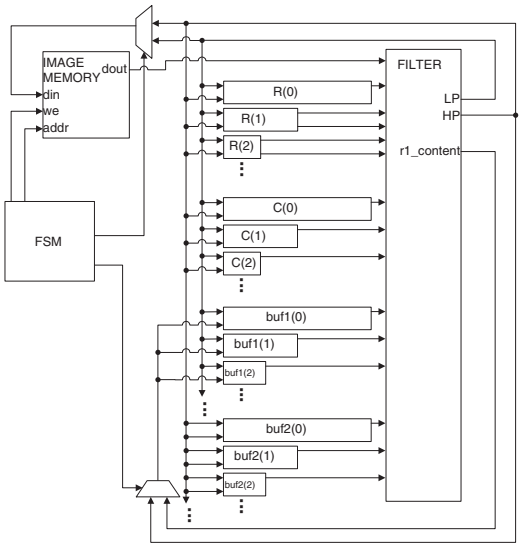

Fig. 9. Block diagram of the LB architecture.

Using the still empty buf1(j) during vertical initialization, we eliminate the extra line buffer that would store the extra information needed for the initialization mirroring to occur. When initialization is over, that is during the normal-mode vertical filtering, the value of the FIFO's first register (r1) and the high-frequency output of the current lifting step, are stored in buf1(j) and buf2(j), respectively. Thus, the coefficients of the preceding lifting step are retrieved at the current step from buf1(j) and buf2(j). In this way, a continuity in the vertical filtering of each column is created, since in LB the vertical filtering, contrary to the horizontal filtering, is *not* inherently continuous. During vertical finalization, the values of buf1(j) is also the samples that are mirrored.

The filter used in LB, is that of Fig. 4, which incorporates a multiple-input mode to take advantage of the multi-port nature of the on-chip RAMs. Fig. 10 depicts how the normal-mode vertical filtering of each column is executed, and shows the multi-port filter's behavior.

V. Block-based Implementation

The BB is implemented bringing on chip blocks of the original image. Traditionally, the size of these blocks is equal to $2^L * 2^L$, to allow the generation of either an LL_L/LH_L or an HL_L/HH_L pair, L denoting the final level. Thus, an on-chip memory of equal size should be used, where blocks are temporally stored. This memory is known as Inter-Pass Memory(IPM). The RC algorithm is then applied on the block, up to the last level, the decomposition of the block is written back to image memory, and the next block is brought on chip. The traditional version of BB demands complicated control and addressing, is not effective in streaming applications and imposes high memory requirements (for six levels of

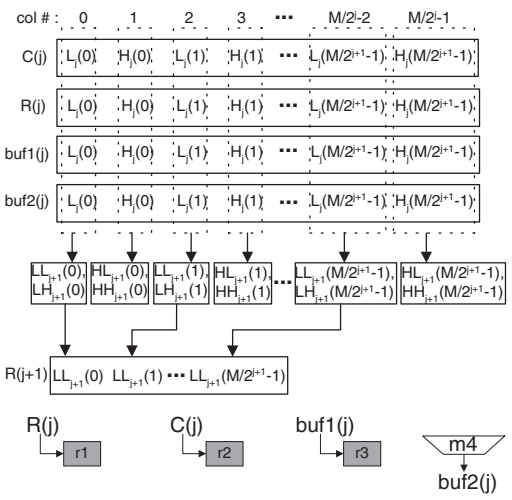

Fig. 10. Vertical filtering in normal mode in an LB architecture. Three inputs are loaded in parallel into the FIFO, while buf2(j) passes through multiplexer m4 of the filter.

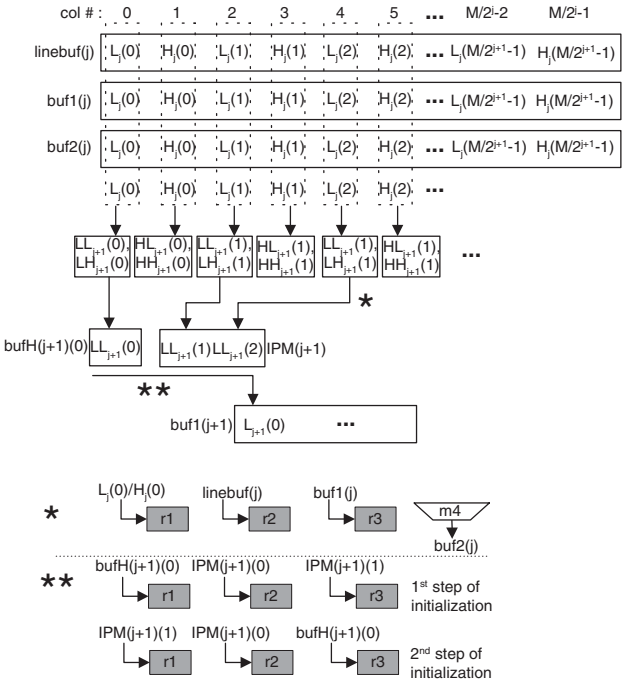

Fig. 11. The normal-mode vertical filtering at level j, in BB, is followed by initialization-mode horizontal filtering at level $j+1$. Being at the beginning of the vertical initialization stage at level $j+1$, L_{j+1} is written in buf1(j+1).

decomposition the size of the IPM would be 4096 words).

The BB version that we implemented is the one that requires the minimum local buffering and simplifies the most the control and the addressing. Each level has its own IPM, where coefficients LL are stored to be filtered horizontally. The size of IPM(j) is such that allows the generation of an L_j/H_j pair at level j. As we have seen, for a new L_j/H_j pair to be generated, two new input coefficients should enter the filter. Thus, the size of IPM(j), where $j = 1, 2, ..., L-1$, will be only 2 words. No IPM is needed for level 0, as the filter's inputs come straight from image memory.

In the previous section, we saw that the vertical filtering is not inherently continuous in LB. In order for the vertical filtering to be executed correctly, line-buffers buf1(j) and buf2(j) were used. In BB, the horizontal - and not only the vertical - filtering is deprived of inherent continuity. Therefore, apart from the use of buf1(j) and buf2(j) in vertical filtering, in exactly the same way as in LB, a two-word register, bufH(j), will be needed to store the two intermediate results of horizontal filtering. As we have seen, buf1(j) will also be used to implement the initialization mirroring without additional hardware cost, storing the mirroring samples. In the same manner, the still empty bufH(j)(0) will be used to store the horizontal mirroring sample.

An L_j/H_j pair, produced after the horizontal filtering of IPM(j), is either stored in a line-buffer (for the odd rows of level j) or consumed at once (for the even rows of level j) to produce either an LL_{j+1}/LH_{j+1} or an HL_{j+1}/HH_{j+1} pair (Fig. 11). Hence, in the case of even rows, a vertical filtering action is undertaken after the generation of every L_j/H_j pair, and no supplementary line-buffer is needed. On the contrary, in Fig. 10, in the even rows of vertical filtering, a whole row of L_j and H_j coefficients had to be written in R(j), so that continuous horizontal filtering would be applied on it. Only after the completion of the horizontal filtering of the even row of level j, the vertical filtering would begin.

The block diagram of the BB architecture is shown in Fig. 12. The on-chip memory needed for level j is illustrated in Fig. 13. The control logic, implemented with the FSM, is a lot more complicated in the case of BB, compared to that of LB. Mainly, it differs from the control logic presented in Fig. 7 in the following:

1) Successive columns of L_j and H_j coefficients are no longer filtered vertically in a successive manner. After a vertical filtering of an even column occurs, if the current level is not the final one, LL_{j+1} is either written in bufH(j+1) or in IPM(j+1), depending on the current stage of horizontal filtering at level $j+1$. A single step of horizontal filtering at level $j+1$ may occur (depending on the horizontal filtering stage) interrupting the vertical filtering of level j.
2) The horizontal filtering is no longer continuous. After a discrete step of horizontal filtering at level $j+1$, it should be decided if a vertical filtering at level $j+1$ can occur. If it *does* occur, a single step of horizontal filtering at level $j+2$ might follow, and so on. This domino effect in the worst case reaches the final level, imposing highly frequent interchanges between successive levels, that complicate the control logic.

As in LB, the filter used in BB should make full use of the parallelism multi-port buffers offer. Thus, the version of Fig. 4

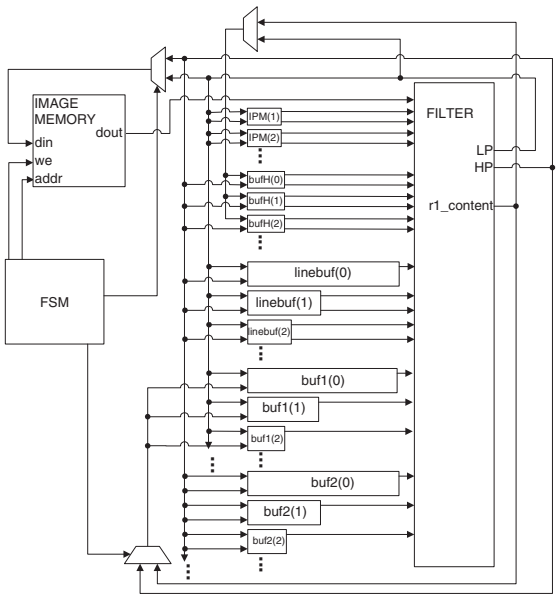

Fig. 12. Block diagram of the BB architecture.

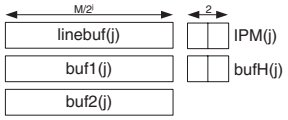

Fig. 13. On-chip memory of level j, used in BB.

is used, so that the filter operates in a single-input mode during the horizontal filtering at level 0 and functions in a multiple-input mode in any other case.

VI. Results and Comparisons

The architectures were implemented in VHDL, synthesized with Synplify Pro 7.7, and placed and routed on Xilinx Virtex 4 XC4VLX15 FPGA, using Xilinx ISE v.8.1. In all the implementations, the image memory shares the same clock with the rest of the system. This simplification renders our comparison as generic as possible, since the image memory is sometimes on-chip (e.g. in large FPGA devices). Under this assumption, it should be noted that the computation schedules with larger traffic towards the image memory are favored, and no interface circuit is needed.

A. Throughput

The RC, the LB and the BB operate on the XC4VLX15 device at 172.4, 113.6 and 117.6 MHz respectively. The three schemes have similar data paths. However, the frequency varies because the critical path lies in the control path.

The LB obtains the highest throughput among the three. This is due to the small number of cycles it requires to complete the 2-D DWT; it starts from 150,024 (M=256, L=3), and reaches 2,374,740 cycles (M=1024, L=6). The throughput of LB reaches 757 frames/sec (M=256, L=3) and drops at 47 frames/sec (M=1024, L=6).

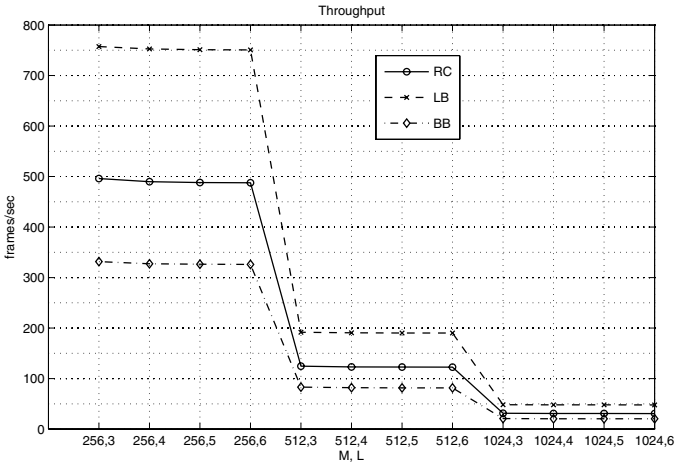

Fig. 14. Throughput results.

There is no great difference between the number of cycles needed for RC and BB. Specifically, RC needs 347,651-5,607,174, while BB requires 354,827-5,767,246 clock cycles. However, the higher frequency in which RC operates, improves the throughput, resulting in the difference observed in Fig. 14. If the image memory operates in a smaller frequency than the rest of the system, BB will outperform RC.

The larger number of cycles in RC, compared to LB, is due to the fact that a single-port memory is used as the only source of inputs for RC. Thus, inputs enter the filter in a serial manner and no parallelism is involved in the filtering operations. On the contrary, in LB and BB, two or three inputs might enter the filter in parallel. Also, in RC the filter's output pair cannot be written in image memory in a single cycle; one of the outputs should be buffered to be written at the next cycle. The results that we got prove that using multi-port buffers, even with a single filter, halves the number of cycles for the LB architecture. But this is not the case for the BB, even if multi-port buffers are *also* used to increase memory-access locality. This is due to the streaming nature of the operations taking part in LB, which is no longer the case for BB. Thanks to that, in LB at many points the next action is predetermined, for example the horizontal filtering is continuous and successive columns are successively filtered. As a result, many low level actions can occur in parallel, as it is pre-decided that they wouldn't affect each other. Things are different for BB, since the control is deprived of such a streaming behavior and many options should be considered at a specific point, instead of following a predetermined route. As a result, in the case of BB, the number of cycles is increased, compared to LB.

B. FPGA slices

The FPGA slices used in RC are much fewer than in LB and BB (Fig. 15). This is due to the simplicity of the control associated with the RC algorithm. The number of slices for RC covers a range from 280 (M=256, L=3) up to 329 slices (M=1024, L=6). For LB and BB this range is 2659-3001 and 2646-3597 slices, respectively.

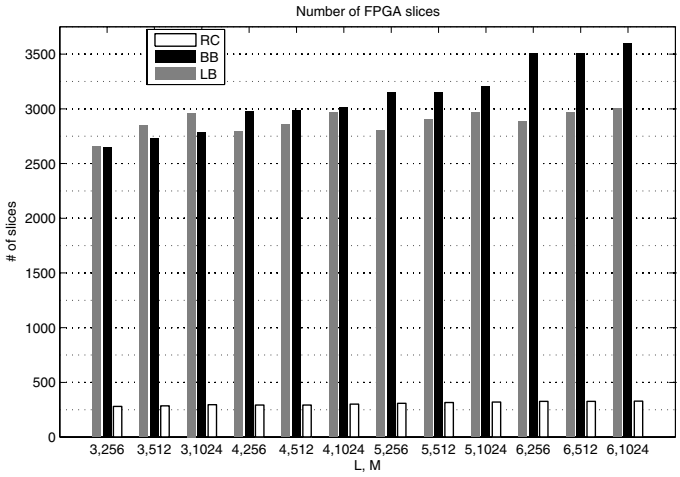

Fig. 15. Number of FPGA slices.

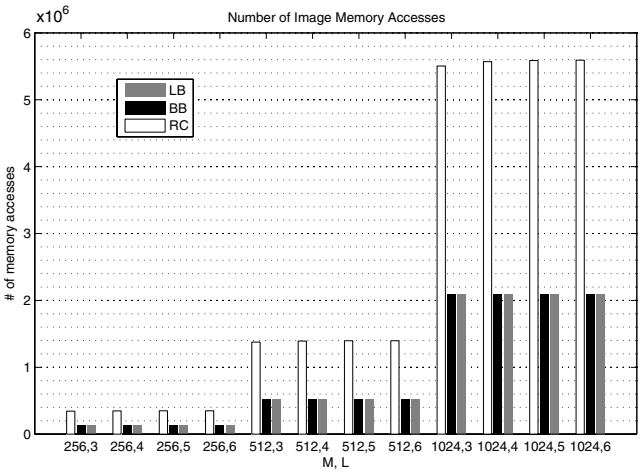

Fig. 16. Total number of all accesses (contains both read and write accesses) of the image memory.

C. Memory issues

The RC does not involve any on-chip buffers, contrary to LB and BB. Thus, in RC the image memory is the only source of inputs for the filter. As a result, the number of image memory accesses is significantly larger in the RC case (Fig. 16). In the cases of LB and BB this number is the same, and does not vary when the number of levels varies, as it is, in both cases, equal to $2 * M^2$.

In LB and BB, the on-chip local memory of each level is accommodated in BRAMs (Fig. 17), generated by the Xilinx ISE Coregenerator, and registers. The bit-width used is 16 bits. The Virtex-4 BRAMs used are 18 K dual-port BRAMs.

To obtain maximum throughput, the buffers of the same type and of different levels can be grouped together in a single BRAM, if the space provided is enough. This way, during the vertical filtering of successive columns, R(j), C(j), buf1(j) and buf2(j) can feed the filter simultaneously. At the same time, locations of buf1(j) and buf2(j), that have already been read, are overwritten with the new intermediate results, to be read at the next lifting step. Moreover, R(j) can be read while R(j+1) is written with the output of a previous column's vertical filtering. Therefore, four dual-port BRAMs should be used in the cases of image sizes 256 and 512. Four additional dual-port BRAMs will be needed for size 1024, to accommodate the larger buffers of level 0. The number of BRAMs remains the same for 3, 4, 5 and 6 levels, to guarantee high throughput and enough space for the larger buffers of the lower levels.

In BB, buffers IPM and bufH of each level are implemented as two-word registers. Contrary to LB, the vertical filtering of successive columns is no longer successive. Thus, the constraints that guarantee maximum throughput are not so strict for BB. During vertical filtering, three filter inputs should be read simultaneously, thus, at least two dual-port BRAMs should be used. To provide enough memory space, 2, 3 and 6 BRAMs should be used for image sizes 256, 512 and 1024, respectively. These choices, which also respect the minimum of two BRAMs, remain the same for 3, 4, 5 and 6 levels, since the larger buffers of the lower levels are the ones that determine the memory space needed.

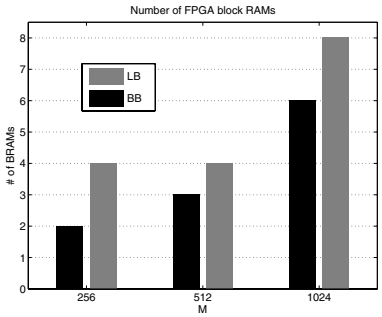

Fig. 17. Number of BRAMs.

VII. Conclusion

The three major 2-D DWT lifting-based computation schedules have been compared in terms of throughput, area and memory requirements on the Virtex-4 FPGA family. The conclusions of this work are the following. The LB has the highest throughput among the three schedules. The BB has the lowest throughput, due to control complexity, as well as to the frequency in which it operates, which is not as high as in the RC case. The RC has by far the lowest hardware cost. Not only it involves no on-chip buffering, but also the FPGA slices used are significantly fewer than in the cases of the other two. The LB and the BB offer similar memory-access locality, higher than RC, minimizing the number of image memory accesses, compared to RC. However, BB achieves this by using a smaller number of BRAMs.

Future work would involve extending the current comparative analysis by broadening the range of the comparison parameters. For instance, using dual-port image memory and more filters would be considered.

REFERENCES

[1] ISO/IEC FCD15444-1: 2000, "JPEG 2000 image coding system," May 2000.

[2] ISO/IEC JTC1/SC29/WG11, FCD 14496-1, "Coding of moving pictures and audio," May 1998.

[3] S. Mallat, "A theory for multiresolution signal decomposition: The wavelet representation," *IEEE Trans. PAMI*, vol. 2, no. 7, pp. 674–693, 1989.

[4] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting schemes," *J. Fourier Anal. Appl.*, vol. 4, pp. 247–269, 1998.

[5] K. Andra, C. Chakrabarti, and T. Acharya, "A VLSI architecture for lifting-based forward and inverse wavelet transform," *IEEE Trans. Signal Processing*, vol. 50, 2002.

[6] C. Chrysafis and A. Ortega, "Line-based, reduced memory, wavelet image compression," *IEEE Trans. Image Process.*, vol. 9, no. 3, pp. 378–389, March 2000.

[7] G. Lafruit, L. Nachtergaele, J. Bormans, M. Engels, and I. Bolsens, "Optimal Memory Organization for Scalable Texture Codecs in MPEG-4," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 9, no. 2, 1999.

[8] N. D. Zervas, G. P. Anagnostopoulos, V. Spiliotopoulos, Y. Andreopoulos, and C. E. Goutis, "Evaluation of design alternatives for the 2-D-discrete wavelet transform," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 11, pp. 1246–1262, December 2001.

[9] M. Weeks and M. Bayoumi, "Discrete Wavelet Transform: Architectures, Design and Performance Issues," *J. VLSI Signal Process.*, vol. 35, 2003.

[10] C. Chakrabarti, M. Vishwanath, and R. M. Owens, "Architectures for wavelet transforms: A survey," *J. VLSI Signal Process.*, vol. 14, pp. 171–192, 1996.

[11] Y. Andreopoulos, P. Schelkens, G. Lafruit, K. Masselos, and J. Cornelis, "High-level cache modeling for 2-D discrete wavelet transform implementations," *VLSI Signal Processing (special issue on Signal Processing Systems)*, vol. 34, no. 3, pp. 209–226, July 2003.

[12] K. Masselos, Y. Andreopoulos, and T. Stouraitis, "Performance comparison of two-dimensional discrete wavelet transform computation schedules on a VLIW digital signal processor," *IEE Proc. Vision, Image and Signal Process., to appear, preprint available from: www.ee.ucla.edu/ yandreop.*