Novel Hardware-Software Architecture for the Recursive Merge Filtering Algorithm

Piyush S Jamkhandi, Amar Mukherjee, Kunal Mukherjee, and Robert Franceschini*
School of Computer Science, University of Central Florida, Orlando FL 32816
*Institute for Simulation and Training, UCF
Email: {amar, mukherje, piyush}@cs.ucf.edu, *rfrances@ist.ucf.edu

Abstract:

As reconfigurable devices move to the forefront of mainstream processing, the spectrum of application areas for such devices is also increasing. One such area is image processing. In this paper we present a novel hardware-software codesign architecture for the computation of the Discrete Wavelet Transform (DWT), based on a new Recursive Merge Filtering (RMF) algorithm. The architecture aims at reducing the overall data routing during the computation of the DWT. The method shows how data routing can be transformed into a series of index computations carried out on the reconfigurable device. The architecture applies hardware/software codesign principles for task division among the available processing resources and uses memory resources closer to the FPGA to avoid the main memory accesses and this reduces the processing time and access time. The paper also proposes ideas based on memory bank to further enhance the performance of the proposed architecture. The architecture is based on a dynamically reconfigurable device, the Xilinx XC6200 on the H.O.T Works Board used to carry out the data routing by a series of index computations. The complex computations are restricted to the main processor and the simple addition/subtraction tasks for data routing are based on the FPGA.

Keywords: Reconfigurable Computing, Discrete Wavelet Transform, Recursive Merge Filtering, Hardware-Software Codesign.

Introduction

The use of FPGAs in various application areas has shown appreciable improvements in performance. Image processing has been one of them. In this paper, we present a hardware-software architecture for the Recursive Merge Filtering (RMF) algorithm for the computation of Discrete Wavelet Transform (DWT). The RMF algorithm [1] overcomes the main disadvantage of conventional wavelet coders-decoders, namely not being able to generate the code until the complete image has been transformed. The use of the RMF algorithm overcomes the performance, functionality and reliability drawbacks of the current DWT methods. This paper builds upon the RMF algorithm by introducing an efficient data routing technique, based on the transformation from data routing to simple computations. We show how the architecture can be used to compute the transform of an image bottom-up and then carry out hierarchical merging of the sub-blocks to obtain the wavelet transform. We also show how the use of an FPGA to perform the data routing, would be an ideal solution for the architecture presented. The essential idea is to separate the actual computation for the transform from the data routing. This can be achieved by using a virtual position mapping for each image position. Using this virtual mapping the data computation can be performed on the actual data inputs, while the data routing is performed on the virtual mapping. Using a virtual coordinate system to define the position of the data being routed, the data routing is transformed into a set of coordinate additions/subtractions. We give a set of generalized equations, which can be applied to any block of the input data to perform the wavelet transform of the entire image. These equations are computed on either the main processor or the FPGA device.

We first present the RMF algorithm from the FPGA perspective. We then describe the RMF algorithm formally. This is followed by an analysis of the various sub-tasks to be carried out for the DWT using RMF. We then present the formal notation for the transformation from data routing to simple coordinate addition and subtraction. The analysis is further developed to define the architecture for computation of the DWT using RMF. We then show how the architecture uses the FPGA resources and the memory resources available on the H.O.T. Works board. Results of the simulation of this architecture showing the reduction in the main memory accesses are finally presented.

RMF Algorithm and FPGA Implementation

The RMF algorithm [1] is based on the principle of preserving the spatial correlation between the inputs and the wavelet coefficients obtained at any stage. This allows the process to be stopped at any time during the computation and yet be able to correlate the coefficients with the original input. This forms an important part of the multiresolution hierarchical zero-tree method. The formal description of the RMF algorithm will be given in the next section. In this section, we justify the use of a hardware-software design

platform to optimize the implementation of the DWT using RMF. We begin by giving a brief description of the RMF algorithm without the formalism.

The DWT using the RMF algorithm is based on blocks of data movement at every stage during the computation. The process of merging blocks by the process of data movement is the basic operation in the RMF algorithm, and is repeated over and over again. Given a set of four blocks of size $2^k x 2^k$, the process of data movement for any block size is repeated 2^{3k} times for various block sizes with only a single compute operation for the smallest 2x2 block. This comparison shows that the total number of data movement operations far outnumber the actual computation. Thus, a technique to reduce the total number of data movements or make it more efficient is required. Because today's microprocessors are well equipped to handle common computations, as in this algorithm, we decided to emphasize the data movement aspect of the algorithm, and relegate the computation responsibility to the microprocessor. The main bottleneck on the microprocessor is the ability to route data to and from it. The access to the main memory forms the main bottleneck in most of the applications. As the reconfigurable devices are incorporated in the computational paradigm, the availability of reconfigurable resources and their applications will definitely broaden. With this perspective, hardware-software codesign architectures will be on the leading edge of application designs.

Reconfigurable devices conventionally provide the flexibility of reprogramming the hardware to implement any hardware of the user's choice. This flexibility has been extended by enabling the designer to make run-time reconfiguration decisions, as in the XC6200 device. The XC6200 device provides the ability to make decisions about the hardware to be configured on the device at run-time. Thus, this allows the designers to delineate certain types of tasks to the reconfigurable device allowing the rest to be computed on the microprocessor. This flexibility allowed us to divide the tasks involved in the RMF algorithm into sub-tasks such that the reconfigurable device and the microprocessor work together to reduce/eliminate the resource bottleneck.

As outlined in the above paragraph the main operation in the RMF algorithm is the merging process, which involves many data movements. These data movements involve the repeated access of the computer's main memory. If we can avoid these repeated accesses, but still achieve the goal of data transfer, then we are done. We go a step further and speed up the overall execution of the algorithm along with the main memory accesses. By the use of an indexing structure, which is discussed later, we show that the process of data shifting and computation can be separated.

The algorithm is split into a software component and a hardware component, which communicate by the means of interprocess communication structures. This architecture is designed to make effective use of the local resources available to the FPGA device i.e. the local RAM on the PCI H.O.T Works board. The idea used in this architecture is the local RAM access by the FPGA for carrying out the data movement. This allows the microprocessor to be utilized for some other process, if need be. The details of the architecture are discussed in the later sections.

Formal Description of the Recursive Merge Filtering Algorithm

In this section we briefly describe the Recursive Merge Filter (RMF) algorithm to compute the DWT for 1-dimensional and 2-dimensional images. The RMF algorithm computes the DWT of a 1-dimensional array of length N (where $N=2^k$ for some positive integer k) in a bottom-up fashion, by successively "merging" two smaller DWTs (four in 2-D), and applying the wavelet filter to only the "smooth" or DC coefficients.

RMF Operator

We will first formally define our primitive, the RMF operator, *in terms of the array indices of two DWT arrays* being merged. This takes as inputs two DWTs, DWT_1 and DWT_2, each of length 2^k, and outputs a DWT of length 2^{k+1}:

$$RMF[DWT_1(0:2^k-1), DWT_2(0:2^k-1)]$$

$$= RMF[DWT_1(0:2^{k-1}-1), DWT_2(0:2^{k-1}-1)] \bullet DWT_1(2^{k-1}:2^k-1) \bullet DWT_2(2^{k-1}:2^k-1)] \quad \text{if k >0}$$

$$= RMF[DWT_1(0:0), DWT_2(0:0)] = h(DWT_1(0), DWT_2(0)) \bullet g(DWT_1(0), DWT_2(0)) \quad \text{if k=0}$$

The RMF operator is defined recursively on sub-arrays of the original DWTs. The first half of DWT_1 and DWT_2 are recursively passed to the RMF operator, and the remaining coefficients of the two DWTs are concatenated (symbolized by '•') at the end, as shown. The recursion terminates when the length of the DWTs being merged becomes equal to one - at this point, the RMF uses the Haar filters h and g [2] , to generate the low pass and high pass coefficients.

DWT in terms of the RMF operator

A recursive notation for the discrete wavelet transform (DWT) of an array x(n) of length $N=2^k$, which directly leads to a recursive procedure to compute the DWT is given below.

$$DWT[x(0:2^k-1)] = RMF[DWT[x(0:2^{k-1}-1), DWT[x(2^{k-1}:2^k-1)]] \quad \text{if } k > 1$$

$$DWT[x(0:1)] = [h(x(0),x(1)), g(x(0),x(1))] \quad \text{if } k = 1$$

The recursion terminates when the length of the array becomes two. At this point, the Haar filters h and g are applied to generate the low pass and high pass coefficients. For proof of the equivalence of the RMF algorithm and the FWT algorithm, refer to [1,2].

RMF Algorithm Computations and Data Shifting

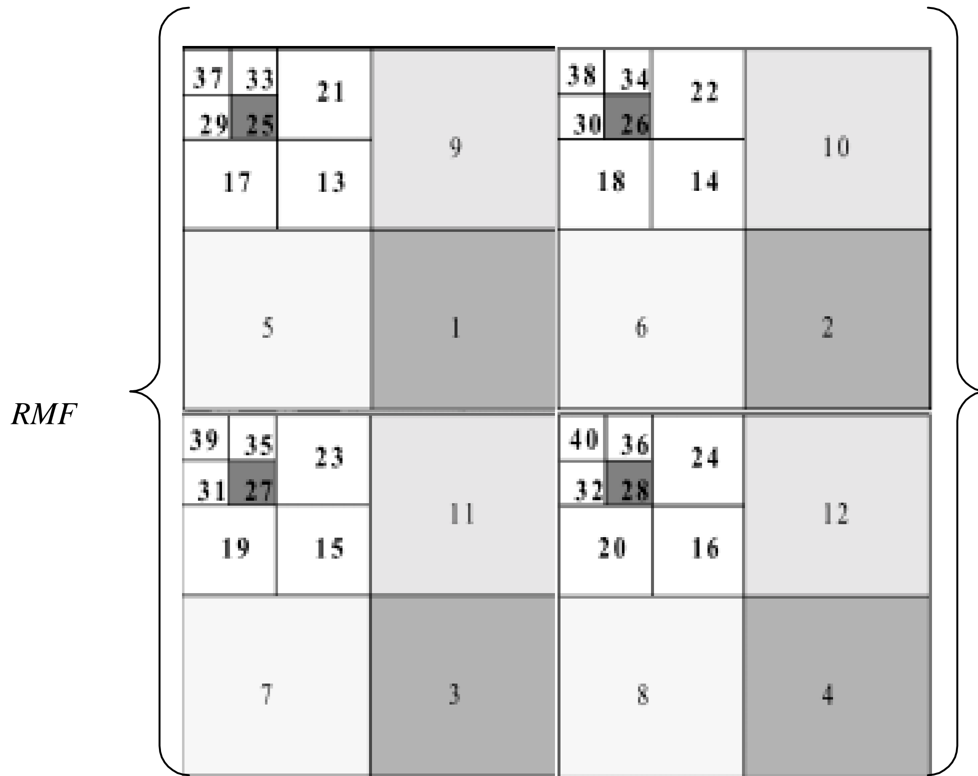
The process of DWT computation can be extended to 2-dimensions. Figure 1.2 shows the computation of the 2-dimensional DWT. The RMF process begins by computing the 2x2 transform of the 2D input data. This is done by selecting a 2x2-block row wise and computing its transform. This process is continued until all the 2x2 blocks in the image are transformed. Once all the blocks are transformed, sets of four adjacent 2x2 blocks are selected for the merging process. The process of merging the 2x2 blocks into 4x4 blocks continues until all the 2x2 blocks are merged into a set of 4x4 blocks. After the completion of this process, a set of four adjacent 4x4 blocks is merged to obtain the next level 8x8 block. This process of merging continues until all the blocks are merged to obtain a single merged block, at which time we are done with the process.

The merging process is inherently an exchange of blocks of data with the adjacent same-sized blocks representing four DWTs of four sub-images. Figure 1.1 shows four blocks to be merged into a single block of DWT for the entire image. If the size of the four quadrants of the merged data block is 2x2 then we apply the basic RMF-2D operation to the top-left matrix quadrant. In case the size of the quadrants is greater than 2x2 we recursively apply the *Merge* operation, until the size of the quadrants becomes 2x2 at which time we apply the RMF-2D operator to the top left quadrant. This process is evident from the recursive equations given below for DWT given in terms of the RMF operator.

Along with the recursive *Merge* operations and the final RMF-2D operation, after a set of four blocks have been merged, we apply a 1D RMF on the bottom left quadrant (D2) , in row wise manner. The1D RMF operator is also applied to the upper right quadrant (D3) in column wise manner. This process is included in the merge process of any given set of four blocks. After the computation of the $RMF_{1D,}$ both row and column wise we either recursively use the *Merge* operator or use the RMF_{2D} operator. One of the key aspects of our design is that all the transforms are computed using the main microprocessor and the FPGA plays no part in the computations. It is only in the data routing that the FPGA plays a vital role in the reduction of the total number of data moves.

The RMF algorithm is mainly based on the bottom up merging process. In the algorithm, two main operators are used: Filter and Merging. These basic operations are performed *k (*$N=2^k$) times on the input image of size NxN, finally resulting in the computation of the DWT. This process is represented as $(Filter, Merge)^{\log N}$. One of the main disadvantages of the RMF algorithm, in its current form, is the large amount of data movement that has to be performed at every merge step. We will depict this with a simple example.

The merge operation can be seen as a swap between two column bands and two row bands. To accomplish this step we carry out eight sets of data swap processes. Thus, for the 4x4 block size, we need to swap 8x4 =32 data values to accomplish the swap process for one merge step. Along with that we also need to perform the row wise and column wise 1D RMF for two sub blocks. Thus, for a merge process we perform the data shifting as shown in Figure 1.2. This is followed by the 1D row wise RMF on the third quadrant and 1D column wise RMF on the second quadrant. If the size of the blocks being merged is 4x4 then we need a total of 8 data movements to perform the complete merge process. We propose the transformation of the data shifting process to a series of simple arithmetic computations. This not only reduces the overall bandwidth requirement for the process, but also reduces the overall execution time of the algorithm. As seen in the above example, even for moderate image sizes the total amount of data movement can quickly degrade to a high bandwidth-high execution time task.



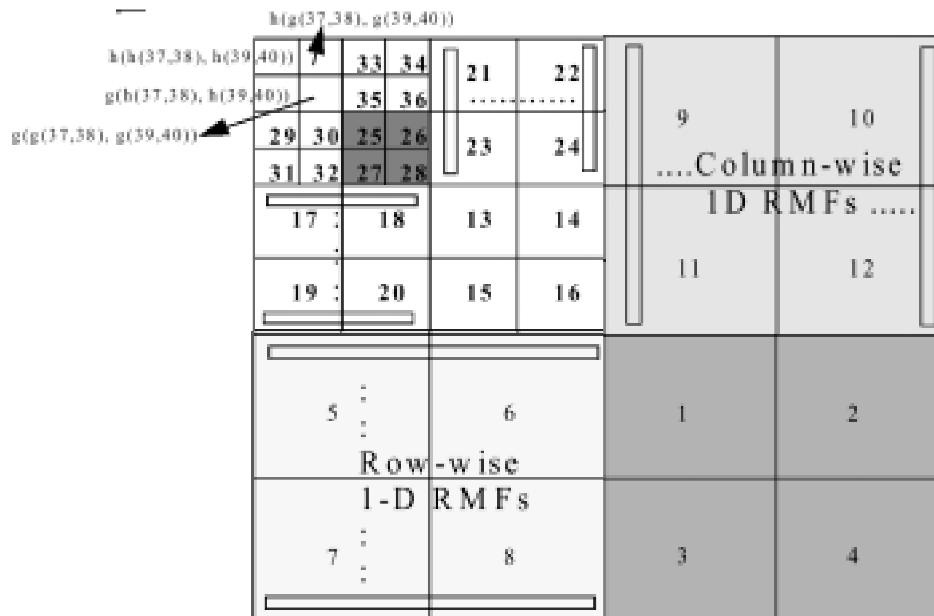


Figure 1.1 Merge process for 4 blocks [1]

$$RMF_{2D}\begin{bmatrix} WT_1(0:2^k-1,\ 0:2^k-1) & WT_2(0:2^k-1,\ 0:2^k-1) \\ WT_3(0:2^k-1,\ 0:2^k-1) & WT_4(0:2^k-1,\ 0:2^k-1) \end{bmatrix}$$

$$= \left[\begin{array}{c:c} \overset{LL}{RMF_{2D}\begin{bmatrix} WT_1(0:2^{k-1}-1,\ 0:2^{k-1}-1) & WT_2(0:2^{k-1}-1,\ 0:2^{k-1}-1) \\ WT_3(0:2^{k-1}-1,\ 0:2^{k-1}-1) & WT_4(0:2^{k-1}-1,\ 0:2^{k-1}-1) \end{bmatrix}} & \overset{HL}{\text{Col-wise 1-D RMFs}\begin{pmatrix} WT_1(2^{k-1}:2^k-1,\ 0:2^k-1) & WT_2(2^{k-1}:2^k-1,\ 0:2^k-1) \\ WT_3(2^{k-1}:2^k-1,\ 0:2^k-1) & WT_4(2^{k-1}:2^k-1,\ 0:2^k-1) \end{pmatrix}} \\ \hdashline \underset{LH}{\text{Row-wise 1-D RMFs}\begin{pmatrix} WT_1(0:2^{k-1}-1,\ 2^{k-1}:2^k-1) & WT_2(0:2^{k-1}-1,\ 2^{k-1}:2^k-1) \\ WT_3(0:2^{k-1}-1,\ 2^{k-1}:2^k-1) & WT_4(0:2^{k-1}-1,\ 2^{k-1}:2^k-1) \end{pmatrix}} & \underset{HH}{\begin{matrix} WT_1(2^{k-1}:2^k-1,\ 2^{k-1}:2^k-1) & WT_2(2^{k-1}:2^k-1,\ 2^{k-1}:2^k-1) \\ WT_3(2^{k-1}:2^k-1,\ 2^{k-1}:2^k-1) & WT_4(2^{k-1}:2^k-1,\ 2^{k-1}:2^k-1) \end{matrix}} \end{array}\right]$$

if $k > 1$, and for a 2-dimensional array of size 2x2 i.e. for k=1 we define the RMF_{2D} as :

$$RMF_{2D}\begin{bmatrix} WT_1(0:0) & WT_2(0:0) \\ WT_3(0:0) & WT_4(0:0) \end{bmatrix} = \begin{bmatrix} h\begin{pmatrix} h(WT_1(0:0),\ WT_2(0:0)) \\ h(WT_3(0:0),\ WT_4(0:0)) \end{pmatrix} & h\begin{pmatrix} g(WT_1(0:0),\ WT_2(0:0)) \\ g(WT_3(0:0),\ WT_4(0:0)) \end{pmatrix} \\ g & g \end{bmatrix}$$

Figure 1.2 2-Dimensional DWT using RMF_{2D}

Given a set of four $2^p x 2^p$ blocks, we need to perform $8x2^{p-1}$ data movements to carry out the data shifting process. Along with that we also need to compute the 1D RMF of the bottom left and top right quadrants , row wise and column wise respectively. This requires an additional 2^{p+2} data movements. If the value of p is greater than one then, the four sub-blocks in the first quadrant are recursively merged. This process continues until the size of the first quadrant is reduced to 2x2. The total number of data movements can be represented by the recursive relation given below:

$$f(2^p) = 2^{2p+2} - 2^{p+2} + f(2^{p-1})$$
$$f(2) = 0$$

Where $1 \le p \le k$ given an $2^k x 2^k$ image.

By expanding the above recursive relation, we obtain the total number of data movements as:

$$3x2^{2k+1} - 6 - 2^{k+3} \quad \text{where size of input image is NxN } (2^k x 2^k).$$

Therefore, in total the merge process and the row-column 1D RMF process require data movements of the order of $O(2^{2k+1})$. Therefore, the complexity of the data movements is linear by the size of the image (2^{2k+1}).

Transformation of Data Routing to Address Computation

As seen above, the data shifting constitutes a major part of the overall process of DWT computation using RMF. Thus, a method to decrease the total number of data shifts is necessary. We propose the use of a virtual mapping index called the *rMap*, which is a pointer from every image position to a data value. Thus, the *rMap* index maps every position on the image with a specific data value

The *rMap* points to a data value at location (k, l) if the image position at location (i, j) has the data value at location (k, l). The use of such a virtual mapping technique enables us to modify the value of the

data item independent of its position. The modification of the position of the data value can be carried out by addition or subtraction to its virtual mapping index. Thus, in case we need to move the data value located at position *(m,n)* to a location *(m+p,n+r)*, we can do the same by simply adding *(p,r)* to the *rMap* index of the data value at position *(m,n).*

At every instance during the computation, there is direct positional correlation between the initial image locations and the *rMap* index. This direct relation is made use of during the computation and the data shifting operation. All the operations carried out on the image are with respect to the original image matrix. Thus, given any coordinate on the original image, we use the *rMap* index to point to the data value at the given coordinate. By the use of the *rMap* index we can compute the data transform independent of the data shifting process. Thus, we can separate the process of *Filtering* from the process of *Merging*. Therefore, the use of this virtual mapping enables us to parallelize the overall implementation of the RMF algorithm, by making the two main operations independent of each other. As we have a series of *(Filter,Merge)* operations we try to perform them concurrently. Although the data values at any given position *(i, j)* have to be accessed through the *rMap* index, the total number of times that such accesses occur can be shown to be a very small percentage of the total accesses, as compared to the total number of accesses of the *rMap* index for data shifting operations. In the earlier implementation of the RMF algorithm [1], we carry out the *Filter* step and follow it up with a *Merge* step. However, in our implementation we carry out the data shifting operations on the pointers of the data values and not the data values themselves. In the next section we show how the data shifting operations are carried out using the *rMap* index by a series of additions/subtractions.

The function of the *rMap* structure is to serve as a mapping between the initial data and their final positions in the DWT after a set of *log n (Filter,Merge)* operations. By the use of the *rMap* index we transform the bandwidth intensive problem into a simple computation problem, which can be solved using the FPGA. Instead of moving the data around we compute the new positions of the data by the use of fixed mathematical equations.

Formal Equations for Data Shifting

In this section, we develop a generalized set of data movement equations, which can be applied to any block size. These set of equations constitute the complete DWT using the RMF algorithm. In Figure 1.3 we have a generalized block at any stage in the *log N* steps taken to complete the process. For any block, we assume the coordinate system shown in the figure. We know from the discussion of the *Merge* operator the type of data block movement that is to be carried out. For example, consider block 1,shown in Figure 1.3.

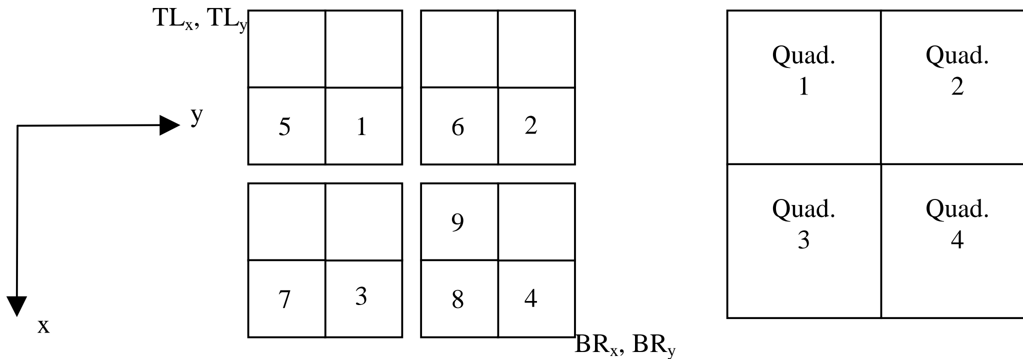


Figure 1.3 Block coordinate system and quadrants.

We know that the data in block 1 has to be shifted to the position in block 9. Given any data item in block 1 in position *(x, y)*, it is moved to position :

$$x + \frac{(BR_x - TL_x)}{4}, \; y + \frac{(BR_y - TL_y)}{4}$$

Where : (BR_x,BR_y) and (TL_x,TL_y) are the bottom right and top left coordinates of the block whose four quadrants are to be merged. Thus, we have managed to transform the data routing process into a simple arithmetic compute process.

We further define the height and width of the block as :

$$\delta_x = BR_x - TL_x$$
$$\delta_y = BR_y - TL_y$$

However, it should be noted that in all cases $\delta_x = \delta_y$.

Let us define a primitive for the merge operation to be carried out on a block whose top left and bottom right coordinates are given. Given (BR_x,BR_y) and (TL_x,TL_y), the primitive

$$Block(BR_x,BR_y,TL_x,TL_y)$$

computes the merge process for the four sub-blocks within the given top and bottom coordinates. This primitive implements all the data movements and the RMF operation on the basic 2x2 block. The *Move_Data* primitive is a part of the *Block* primitive, which is used to move the data . The *Move_Data* primitive is passed six arguments : top left coordinates, bottom right coordinates and the re-location pair *(m,n)* which is added to all the *rMap* within the given top and bottom coordinates. The *Move_Data* primitive is defined as below :

Move_Data(TL_X,TL_Y,BR_X,BR_Y, m,n)

Along with these primitives, we also define primitive to compute the 1D RMF of the bottom left and the top right quadrants. This primitive computes the row(or column) 1D RMF of all the rows (or columns) in the specified block. The arguments passed to the *Compute-1D* primitive are the coordinates of the block and whether the operation is row wise or column wise.

Initially, we compute the basic 2x2 RMF of all 2x2 sized blocks in an image. Once all such blocks are computed, we pass the coordinates of the top left and bottom right of 4 adjacent 2x2 blocks to the *Block* primitive. This process is repeated until all the 2x2 blocks are merged into 4x4 blocks. The process is repeated again, with a pair of top left and bottom right coordinates of four adjacent 4x4 blocks, being passed to the *Block* primitive. This process continues until all the blocks are merged to form a single block. The following set of equations using the primitives defined above, is the complete RMF operation for any given input block.

To generate Quadrant 1:

$$Move_Data(TL_x, TL_y + \frac{\delta_y}{2}, TL_x + \frac{\delta_x}{4}, TL_y + \frac{3\delta_y}{4}, 0, \frac{-\delta_y}{4})$$

$$Move_Data(TL_x + \frac{\delta_x}{2}, TL_y, TL_x + \frac{3\delta_x}{4}, TL_y + \frac{\delta_y}{4}, \frac{-\delta_x}{4}, 0)$$

$$Move_Data(TL_x + \frac{\delta_x}{2}, TL_y + \frac{\delta_y}{2}, TL_x + \frac{3\delta_x}{4}, TL_y + \frac{3\delta_y}{4}, \frac{-\delta_x}{4}, \frac{-\delta_y}{4})$$

To generate Quadrant 2:

$$Move_Data(TL_x, TL_y + \frac{\delta_y}{4}, TL_x + \frac{\delta_x}{4}, TL_y + \frac{\delta_y}{2}, 0, \frac{\delta_y}{4})$$

$$Move_Data(TL_x + \frac{\delta_x}{2}, TL_y + \frac{\delta_y}{4}, TL_x + \frac{3\delta_x}{4}, TL_y + \frac{\delta_y}{2}, \frac{-\delta_x}{4}, \frac{\delta_y}{4})$$

$$Move_Data(TL_x + \frac{\delta_x}{2}, TL_y + \frac{3\delta_y}{4}, TL_x + \frac{3\delta_x}{4}, BR_y, \frac{-\delta_x}{4}, 0)$$

To generate Quadrant 3:

$$Move_Data(TL_x + \frac{\delta_x}{4}, TL_y, TL_x + \frac{\delta_x}{2}, TL_y + \frac{\delta_y}{4}, \frac{\delta_x}{4}, 0)$$

$$Move_Data(TL_x + \frac{\delta_x}{4}, TL_y + \frac{\delta_y}{2}, TL_x + \frac{\delta_x}{2}, TL_y + \frac{3\delta_y}{4}, \frac{\delta_x}{4}, \frac{-\delta_y}{4})$$

$$Move_Data(TL_x + \frac{3\delta_x}{4}, TL_y + \frac{\delta_y}{2}, BR_x, TL_y + \frac{3\delta_y}{4}, 0, \frac{-\delta_y}{4})$$

To generate Quadrant 4:

$$Move_Data(TL_x + \frac{\delta_x}{4}, TL_y + \frac{\delta_y}{4}, TL_x + \frac{\delta_x}{2}, TL_y + \frac{\delta_y}{2}, \frac{\delta_x}{4}, \frac{\delta_y}{4})$$

$$Move_Data(TL_x + \frac{\delta_x}{4}, TL_y + \frac{3\delta_y}{4}, TL_x + \frac{\delta_x}{2}, BR_y, \frac{\delta_x}{4}, 0)$$

$$Move_Data(TL_x + \frac{3\delta_x}{4}, TL_y + \frac{\delta_y}{4}, BR_x, TL_y + \frac{\delta_y}{2}, 0, \frac{\delta_y}{4})$$

To Compute the 1D RMF:

$$Compute - 1D(TL_x + \frac{\delta_x}{2}, TL_y, BR_x, TL_y + \frac{\delta_y}{4}, Row)$$

$$Compute - 1D(TL_x, TL_y + \frac{\delta_y}{2}, TL_x + \frac{\delta_x}{2}, BR_y, Col)$$

To include the basic RMF_{2D} operation for the basic 2x2 block, we include the equation derived from the definition of the DWT computation using the RMF_{2D} operator as the RMF_{2D} operator is used only when the size of the block is greater than 2x2. Thus, we recurse only if the dimensions of the input blocks are greater than 2x2, otherwise we apply the basic RMF_{2D} operator on the block. We therefore define the equation for the basic RMF_{2D} operator as below:

$$RMF_{2D}(TL_x, TL_y, TL_x + 1, TL_y + 1) \qquad if\ (\delta_x = 4, \delta_y = 4)$$

Thus, after shifting the data values and applying the RMF_{1D} we check if the size of the quadrants is 2x2. In case the size of the quadrants is 2x2, we apply the RMF_{2D} primitive. Otherwise, we recursively use the *Block* operation on the first quadrant. The recursive call to the same set of equations is represented as:

$$Block(TL_x, TL_y, TL_x + \frac{\delta_x}{2}, TL_y + \frac{\delta_y}{2}) \qquad if\ (\delta_x > 4, \delta_y > 4)$$

The above sets of equations represent the complete DWT using the RMF algorithm. Making use of the above equations, we show how the DWT using RMF can be implemented on our proposed architecture. In the next section, we discuss the details of the proposed architecture and show how the above primitives can be implemented using the architecture.

Hardware-Software Architecture

The main aim of this architecture is to exploit the inherent parallelism between the data transforms and the data shifting process. This is achievable by the use of a reconfigurable device along with the main microprocessor. The overall architecture of the system is shown in Figure 1.4. The main components of the architecture are :

a. Primitive Block Computation Software Unit (PBCSU)
b. Hardware-Software based Merge Process (MPS and the FPGA)
c. Main memory based Queuing structure.

The two other components included are the queue Q1 exclusion zone and the *rMap* access area. These two are used to allow only a single process access to the structures, which is a classic operating

systems problem. We use these exclusion areas to maintain the consistency of both the data and the current queuing structure.

We briefly outline the working of the architecture. Initially the input image is stored in the main memory in an array of data values. The primitive block computation software unit (PBCSU) , reads 2x2 blocks from the main memory and computes the basic 2D RMF. The data values of the resulting 2x2 matrix are written back to the main memory. However, the coordinates of the top left and bottom right of each 2x2 block computed by the PBCSU are written to queue one. This process continues as a thread until all the 2x2 blocks are computed and the thread then terminates.

In addition to this thread, we also have the merge process software unit (MPSU) which checks the status of queue1. Whenever the length of queue is 4, the MPSU reads four sets of coordinates from the queue. These four coordinates represent the four blocks that are to be merged. The MPSU then carries out the data movement operation using the hardware configured on the FPGA during setup. The data for the addition process is read from the FPGA on-board RAM, which stores the *rMap* index. Once the merging process is completed, the coordinates of the merged block are written to queue Q2. The MPSU then repeats the process.

The process continues until all the blocks in the queue Q1 are processed at which time all blocks of size 2x2 have been merged into 4x4 blocks. The MPSU then begins to read the block coordinates from the queue Q2, merges the blocks and writes the resulting coordinates to queue Q1. This process of switching between queue Q1 and queue Q2 is repeated until all the blocks are processed and we have a single entry in one of the queues.

The architecture is designed to delineate the process of shifting data from the actual computation of the transform. This is done by creating two processes: the Primitive Block Computation and the MPSU process. These processes run concurrently. The two queuing structures are used for the sharing of data between the two processes. The PBC process computes the 2x2 primitive block transform and the MPSU process merges the smaller blocks into larger blocks.

In the conventional RMF implementation, the processing of the 2x2 blocks is completed before the actual merge process begins. However, as both the processes are being run concurrently, atleast four 2x2 transformed blocks are necessary to start merging them. Thus, the PCB process, computes a basic 2x2 transform and writes it to queue Q1. The MPSU process, which is based on a semaphore to avoid busy waiting, is activated once the number of data items in queue Q1 is atleast four. Upon activation the MPSU process reads the four data items from the queue Q1, merges them using the *Merge* primitive defined above and writes the result to queue Q2. The use of the queue Q2 will be explained shortly.

This process of reading data items from queue Q1 and writing the result to queue Q2 continues until all the 2x2 blocks in the original image are processed. The MPSU process is said to be working in R1W2 context, where it reads the data items from queue Q1 and writes the results to queue Q2. Once all the 2x2 blocks written to queue Q1 by the PBC process, are merged the MPSU process changes its context to R2W1 i.e. read the data items from queue Q2 and write the results to queue Q1. Note that at this instance all the basic blocks in queue Q2 are of 4x4 size and not 2x2 size. During the process of merging of 4x4 blocks we directly apply the RMF_{2D} operator on the first quadrant and do not recurse. However, on larger sized blocks we need to recurse in which case the data items are written back to the same queue they were read from. This is the only variation for the recursion process. In general, for any recursive applications of the MPSU process the recursed block is added at the end of the queue from which the blocks to be merged were read from. Once all the 2x2 blocks are transformed, the PBC process is terminated until a 2x2 block needs to be transformed. The process of reading from one queue and writing to another queue continues until the length of both the queue reduces to zero. At this time, we are done with the process, and we terminate.

During the concurrent execution of both the PBCSU and the MPSU process, we need to allow only one of the processes access to the queue to maintain its consistency. Therefore, we use a mutual exclusion methodology for accesses to queue Q1. The same holds true for accesses to the *rMap* index. Access to the index is required by both the MPSU process and the PBCSU process . The PBCSU process needs access to the *rMap* array to obtain the data values at a given position (x, y). Thus, as concurrent access to the *rMap* index cannot be allowed through the FPGA we use mutual exclusion to separate the accesses.

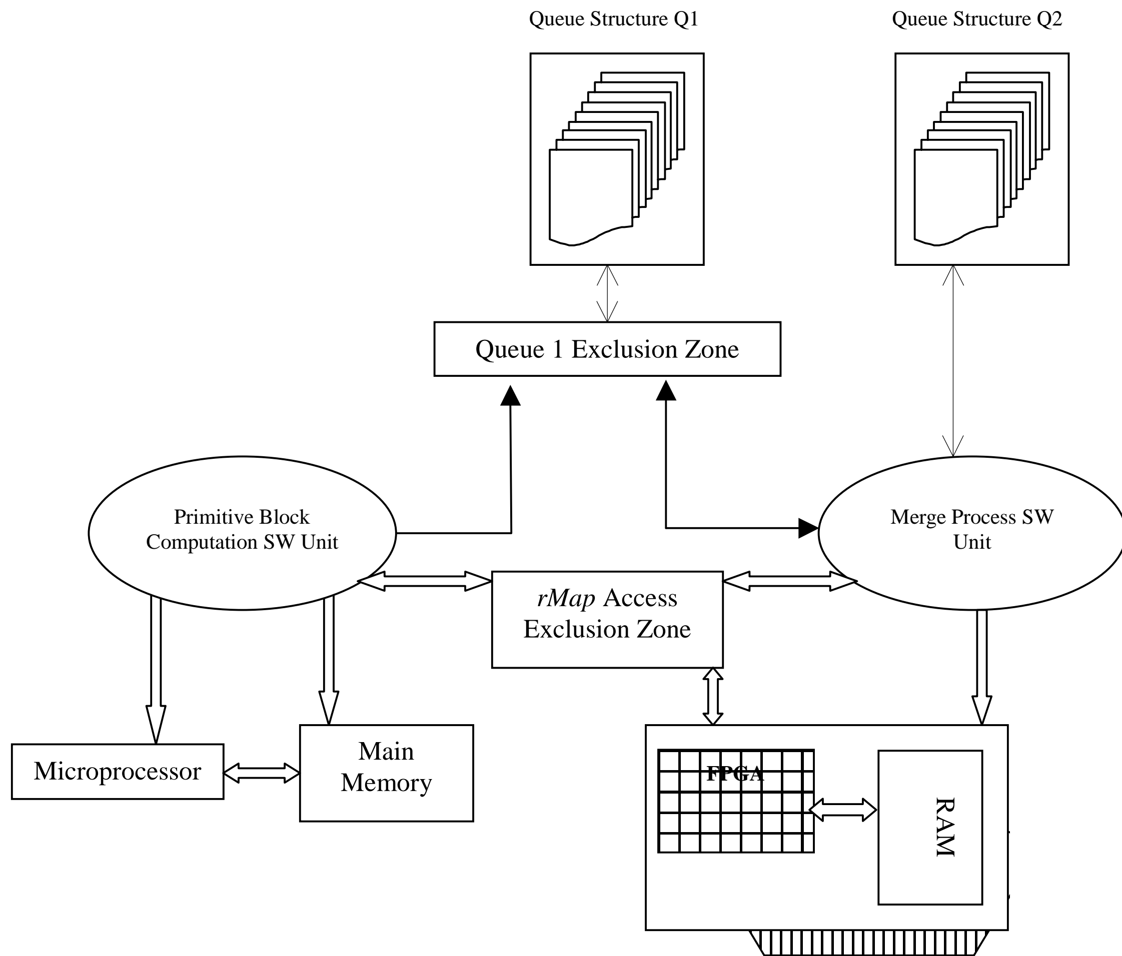


Figure 1.4 Hardware Software Architecture for DWT using RMF.

One important point to be noted in the MPSU process, is the terminating condition and how the termination comes about. The MPSU process reads items from either queue. Depending upon whether the size of the read blocks is 4x4 or larger the block may need merging and the sub-block Q1 may need further merging if its size is larger than 2x2. However, to avoid this process of addition to the queue from becoming infinitely recursive, we differentiate between the blocks which have been added after the completion of the merge process and (added for further higher level merging) those blocks which have been recursively added during higher sized processing so that the quadrant 1 can be merged/operated upon by RMF. This is necessary for the process of merging to stop and ensure that only the correct blocks are processed.

FPGA Implementation and Resource Use

The H.O.T Works board from VCC [4] has been provided with onboard RAM, which can be used to store the *rMap* index. This technique of implementation of the *rMap* index use is efficient as the data shifting process can be carried out by the means of an addition/subtraction circuit configured on the FPGA. The *rMap* index contains a series of (x,y) pairs which point to a specific location in the original data matrix. The figure 1.5 shows the implementation technique on the FPGA.

Initially, the RAM onboard the FPGA is reset and the address values stored in the RAM locations are their addresses. The mapping from the 2D input address space to the 1D RAM space is performed by simple arithmetic calculation. During the transform computation, the FPGA RAM is accessed and the addresses stored at the specified location are returned. Using the returned values, the original array is indexed thereby obtaining the data values. The process of computation of the data shifting is also similar.

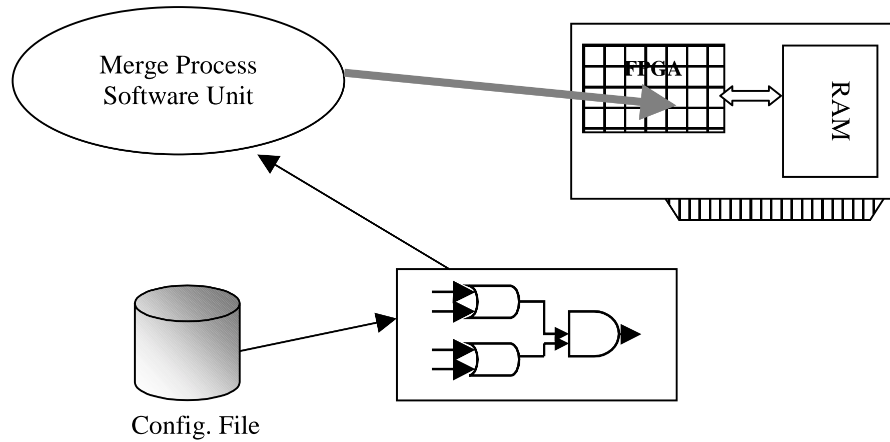


Figure 1.5 Usage of the H.OT. Works onboard RAM for *rMap*

The aim of the setup is to obtain the address of a given position from the FPGA RAM and do simple arithmetic on these values to shift the data location and finally write back the results to the RAM. This process, is carried out easily using the software control program which reads the contents of a given RAM location and passes these values on to a circuit configured on the FPGA. Once the computation of the new address is complete, the new address values are written back to the same location of the RAM. Thus, most of the accesses are local in nature restricted to the FPGA and its on-board RAM. By the use of RAM- banks we can parallelize the overall data shifting operation which can lead to further improvement in the performance of the algorithm.

Figure 1.6 shows the comparative number of data accesses for the conventional RMF and the hardware-software implementation of RMF. We see a substantial decrease in the total number of direct accesses. Although we need to reset the data array to the correct positions after the completion of all blocks of a certain level, we can do so by using the block access mechanism rather than singular data accesses. This blocks access mechanism is a fraction of the initial data accesses. Figure 1.7 shows the original gray map image along with the reverse-transformed image.

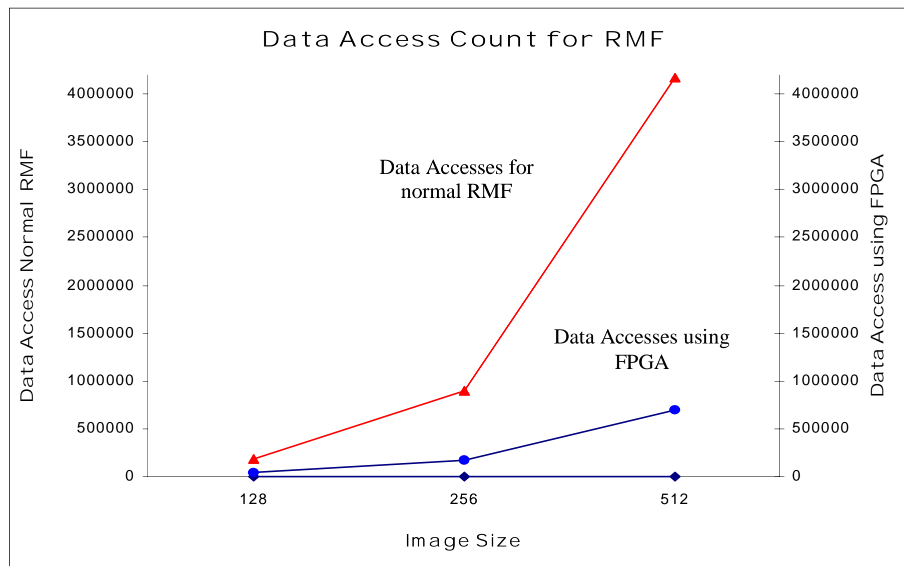


Chart 1.6 Data accesses comparison between the normal RMF and FPGA based RMF

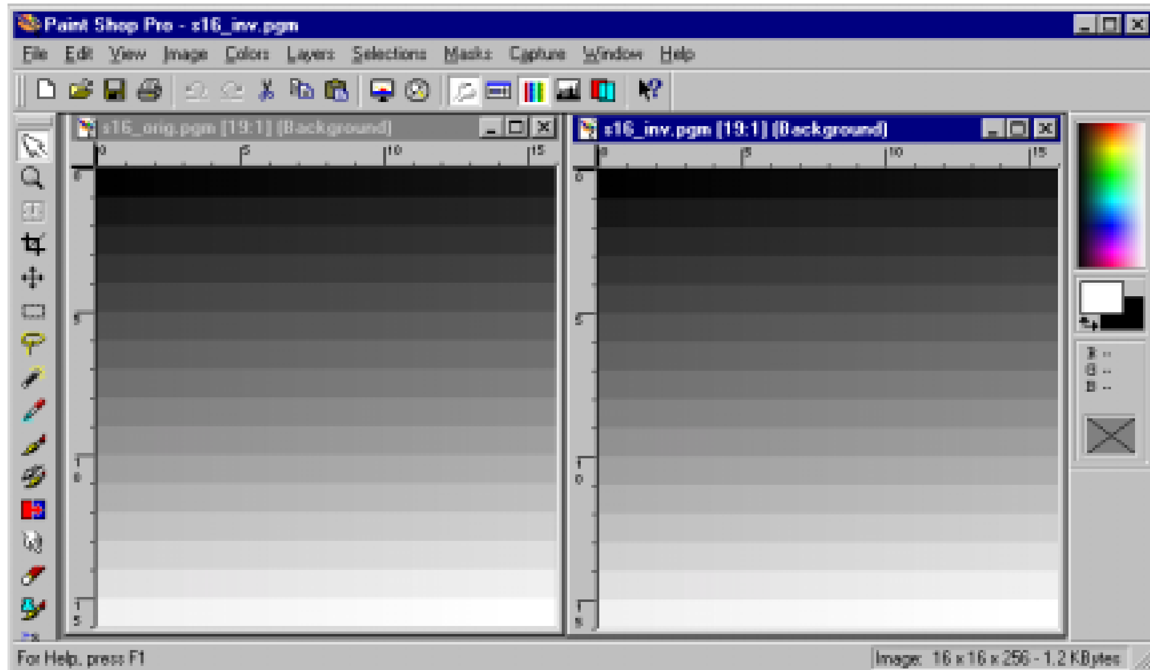


Figure 1.7 Original and gray map reverse-transformed from the wavelet coefficients.

References:

[1]. K. Mukherjee, "Image Compression and Transmission using Wavelets and Vector Quantization, Ph.D. Dissertation, University of Central Florida, 1999.

[2]. S. G. Mallat, "A Theory for Multiresolution Signal Decomposition: The Wavelet Representation", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol.11, no.7, pp. 674- 693, July 1989.

[3]. Xilinx XC6200 Field Programmable Gate Array, Xilinx Inc., April 1997.

[4]. VCC H.O.T Works Board Manual