2005 IEEE International
Symposium on Signal Processing
and Information Technology

Fast Integer-to-Integer Reversible Lifting Transform with Reduced Memory Consumption

Jose Oliver, Elena Oliver, Manuel P. Malumbres
Department of Computer Engineering (DISCA)
Universidad Politécnica de Valencia
Camino de Vera 17, 46022 Valencia, Spain
Phone: +34 963877007, Fax: +34 963877579
e-mail: joliver@disca.upv.es

Abstract— **This paper addresses the problem of reducing the memory usage in the implementation of a reversible two-dimensional wavelet transform for image processing. In particular, we take a line-based approach by using a recursive algorithm to ease the synchronization among different buffer levels. In addition, since the reversible transform is non-linear, to preserve reversibility, we have to consider the order of the horizontal and vertical transforms in which the two-dimensional forward and inverse wavelet transform are decomposed. The proposed algorithm is suitable for integer-only devices (such as many FPGAs), reducing in more than 250 times the memory requirements, for a 5-Megapixel image with the well-known B5/3 wavelet transform. Moreover, it is several times faster (up to ten times) in cache-based systems, due to the better use of the cache memory if compared with the regular wavelet transform.**

I. INTRODUCTION

One of the new mathematical tools that has aroused great interest in the field of signal processing is the discrete wavelet transform (DWT). This interest is even greater in the field of image processing due to its nice features, such as multiresolution representation, space and frequency interpretation (useful for image vision and segmentation) and high compactness of energy in the lower frequency subbands, which is extremely useful in image compression.

The wavelet transform was earlier defined and implemented using a regular filtering operation following a multiresolution analysis [7], but a more efficient algorithm to compute it was introduced by Sweldens in [8]. This algorithm is called the lifting scheme. The main advantage of this approach is the reduction in the number of operations needed to perform the wavelet transform. An additional advantage is that it allows in-place computation, and hence no extra memory is required to store the resulting coefficients as it happens with any regular filtering method. The third benefit that the lifting scheme introduces is the feasibility of a reversible integer-to-integer wavelet transform with only a slight modification of the usual floating-point implementation. In this paper, we will deal with this type of integer wavelet transform.

An integer implementation of a signal transform is needed if the transform is implemented in hardware architectures that only support integer arithmetic, such as some DSPs and many FPGAs. In fact, doing floating-point on FPGAs is difficult due to large amount of hardware required. In addition, some specific applications, such as lossless compression, require reversibility, which is not guaranteed with regular floating-point operations due to the finite-precision of the operands. In this case, a reversible integer-to-integer implementation is needed, even if the hardware platform handles floating-point.

An additional restriction that is usually present in many devices, such as digital cameras and PDAs, is the low amount of RAM memory available in the system. This problem is not very important in some transforms because they are applied in small block sizes. Unfortunately, wavelet-based systems are typically implemented by memory-intensive algorithms, with higher execution time. In the usual DWT [7], the image is transformed at every decomposition level first row by row and then column by column, and hence it must be kept entirely in memory. In [5], an interesting line-based approach to reduce the amount of memory required to compute the wavelet transform was introduced. However, In [5], the explanation of a line-based strategy is given in an iterative way, and no detailed algorithm is described. Some major problems arise when the line-based DWT is implemented using an iterative algorithm. In addition, the transform algorithm introduced in [5] aims to provide a general-purpose wavelet transform. Since an integer-to-integer implementation of the wavelet transform is no longer linear, some new issues about the order in which the DWT is computed need to be considered.

The rest of this paper is structured as follows. In Section 2, there is a more detailed description of the lifting scheme, focusing on a reversible implementation with integer data types. Section 3 introduces the general line-based approach, which is used as a starting point for Section 4, in which we propose a recursive implementation of this approach for an integer transform. Finally, in Section 5, some experimental results are given.

II. THE REVERSIBLE INTEGER-TO-INTEGER LIFTING SCHEME

A. Wavelet transform using the lifting scheme

As we mentioned in the previous section, the lifting scheme implements an in-place DWT decomposition as an alternative algorithm to the classical filtering algorithm. In the filtering algorithm, in-place processing is not possible because each input sample is required as incoming data for the computation of its neighbor coefficients. Therefore, an extra array is needed to store the resulting coefficients, doubling the memory requirements. In addition, the lifting-scheme reduces the number of operations needed to compute the DWT.

In Figure 1, we present a diagram to illustrate the general lifting process. The whole process consists of a first lazy transform, one or several prediction and update steps, and coefficient normalization. In the lazy transform, the input

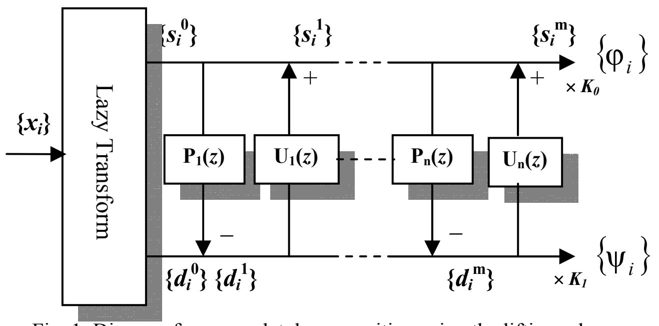

Fig. 1: Diagram for a wavelet decomposition using the lifting scheme.

samples are split into two data sets, one with the even samples and the other one with the odd ones. Thus, if we consider $\{x_i\}$ the input samples, we define both coefficient sets as:

$$\{s_i^0\} = \{x_{2i}\} \qquad \{d_i^0\} = \{x_{2i+1}\}$$

Then, in a prediction step (sometimes called dual lifting), each sample in $\{d_i^0\}$ is replaced by the error committed in the prediction of that sample from the samples in $\{s_i^0\}$:

$$d_i^1 = d_i^0 - P(\{s_i^0\})$$

while in an update step (also known as primal lifting), each sample in the set $\{s_i^0\}$ is updated by $\{d_i^1\}$ as:

$$s_i^1 = s_i^0 + U(\{d_i^1\})$$

After *m* successive prediction and update steps, the final low frequency coefficients (scaling $\{\varphi_i\}$) and high frequency coefficients (wavelet $\{\psi_i\}$) are achieved normalization:

$$\{\varphi_i\} = K_0 \times \{s_i^m\} \qquad \{\psi_i\} = K_1 \times \{d_i^m\}$$

A nice feature of the lifting scheme is that it is formed by very simple steps, and each of these steps is easily invertible, which leads to an almost trivial inverse transform. For the inverse transform, we only have to perform the inverse operations in the reverse order. Hence, from the subsets $\{\varphi_i\}$ and $\{\psi_i\}$, we can get $\{s_i^m\}$ and $\{d_i^m\}$ simply by dividing these coefficients by the scaling factors:

$$\{s_i^m\} = \{\varphi_i\}/K_0 \qquad \{d_i^m\} = \{\psi_i\}/K_1$$

Then, an inverse update operation can be done from these data sets as follows:

$$s_i^{m-1} = s_i^m - U(\{d_i^m\})$$

and at this moment, we can apply the inverse prediction step:

$$d_i^{m-1} = d_i^m + P(\{s_i^{m-1}\})$$

After *m* successive inverse update and prediction steps, we get the initial sets of even and odd samples, we can interleave these data sets to obtain the original set of samples $\{x_i\}$.

B. Integer-to-integer transform

With the above scheme, floating-point arithmetic is needed despite having integer input samples (e.g., image pixels), if the weighting factors employed for the prediction/update operations are floating-point and not integer or rational. Actually, even if rational filters are employed, the precision required to perform lossless operation with fixed-point arithmetic grows with each mathematical operation if we do not change the scheme described above.

Fortunately, the lifting scheme can be slightly modified to achieve reversible integer-to-integer wavelet transform [3]. Since the lifting scheme is formed by several simple steps, the whole process can be reversible if we perform each single step in a reversible way.

For the forward transform, we have seen that each prediction step has the form:

$$d_i^m = d_i^{m-1} - P(\{s_i^{m-1}\})$$

In a wavelet transform for integer implementation, the prediction operation $P(\{s_i^{m-1}\})$ involves rational weighting factors (e.g., division by two), and hence the resulting data are not integer. If a rounding operation is added after the prediction operation, an integer variable can be used to store the result of that operation, and hence each d_i^m can be computed from d_i^{m-1} and the $\{s_i^{m-1}\}$ set using integer values as follows:

$$d_i^m = d_i^{m-1} - \lfloor P(\{s_i^{m-1}\}) \rfloor$$

In the inverse transform, the exact value of each d_i^{m-1} can be recovered from d_i^m and the $\{s_i^{m-1}\}$ set as follows:

$$d_i^{m-1} = d_i^m + \lfloor P(\{s_i^{m-1}\}) \rfloor$$

Thereby, perfect reconstruction is guaranteed despite the rounding operation. The same analysis can be performed for an update operation with integer data type.

Although we have used the floor operator for rounding in the above equations, any other rounding operation, such as ceil or rounding to the nearest integer, can be used as long as the same operator is employed in both the forward and inverse transforms.

Finally, a reversible integer-to-integer transform can only be obtained if the normalization factors K_0 and K_1 are integer values.

A drawback of the use of rounding is that the new wavelet transform is no longer linear. Hence, for a 2D wavelet transform, the reverse column-row order of the forward transform has to be used in the inverse transform to achieve perfect reconstruction.

C. An implementation using the bi-orthogonal 5/3 transform

The 5/3 wavelet transform is a typical wavelet for integer-to-integer transform, being part of the JPEG2000 standard for lossless compression. In order to compute it in terms of the lifting scheme, after the lazy transform, the dual lifting is calculated as:

$$d_i^1 = d_i^0 - \left\lfloor \frac{1}{2}\left(s_i^0 + s_{i+1}^0\right) \right\rfloor$$

while the primal lifting is (notice the different rounding):

$$s_i^1 = s_i^0 + \left\lfloor \frac{1}{4}\left(d_i^1 + d_{i-1}^1\right) + \frac{1}{2} \right\rfloor$$

These operations can be easily performed with integer data types and integer arithmetic. For example, in C language, the two above equations can be efficiently computed as:

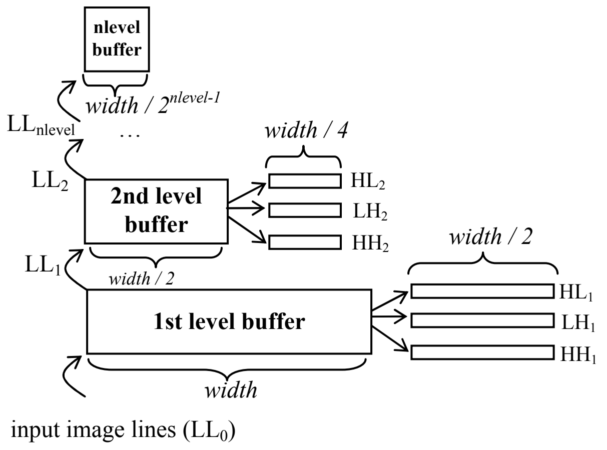

Fig. 2: Overview of a line-based wavelet transform

```
d1[i]=d0[i]-((s0[i]+s0[i+1])>>1);
s1[i]=s0[i]+((d1[i]+d1[i-1]+2)>>2);
```

Where `d0, d1, s0` and `s1` are arrays of integers, and >> is the right shift operator in C ($a >> b$ is equivalent to the division of a by 2^b with floor rounding).

For a lossless transform, the normalization factors K_0 and K_1 are equal to 1, achieving (1,2) normalization in this case. Thus, the set $\{d_i^{\,1}\}$ is directly the final wavelet coefficient set, and the set $\{s_i^{\,1}\}$ is the scaling one.

The inverse transform to recover losslessly the original samples is given by:

$$s_i^{\,0} = s_i^{\,1} - \left\lfloor \frac{1}{4}\left(d_i^{\,1} + d_{i-1}^{\,1}\right) + \frac{1}{2} \right\rfloor \qquad d_i^{\,0} = d_i^{\,1} - \left\lfloor \frac{1}{2}\left(s_i^{\,0} + s_{i+1}^{\,0}\right) \right\rfloor$$

Other reversible integer-to-integer wavelet transforms are given in [2], including an integer version of the popular bi-orthogonal 9/7 transform.

III. THE LINE-BASED APPROACH

For image wavelet transform (2D), the use of the lifting scheme shows little benefit, since the entire image has to be kept in memory. Therefore, it has to be applied along with other strategies that allow us to avoid keeping the entire image in memory. The line-based approach [5] can help us to overcome this problem. In the line-based approach, for the first decomposition level, we receive directly image lines, one by one. On every input line, a one-level 1D DWT is applied. Then, these lines are stored in a buffer associated to the first decomposition level. When there are enough lines in the buffer to calculate a line of each wavelet subband, we compute them. Then, the wavelet subband lines can be processed and released. However, the first line of the LL_1 subband does not belong to the final result, and is needed as incoming data for the following decomposition level. In order to get more lines, we have to update the buffer, filling it with more lines and discarding those that are no longer needed. At the second level, its buffer is filled with the LL_1 lines that have been computed in the first level. Once the buffer is completely filled, it is processed as we have described for the first level. As it is depicted in Figure 2, this process can be repeated until the desired decomposition level (*nlevel*) is reached.

In [5], the description of a line-based strategy is given in an iterative way, but no detailed algorithm is described. Some major problems arise when the line-based DWT is implemented using an iterative algorithm. The main drawback is the synchronization among buffers. Before a buffer can produce lines, it must be completely filled with lines from the previous buffer, therefore they start working at different moments, i.e., they have different delays. Moreover, all the buffers exchange their result at different intervals, depending on their level.

The time in which each line is passed to the following buffer depends on several factors, such as the filter size, the number of decomposition levels, the level and number of line being computed and the image size. In a hardware implementation, with a fixed image size and a constant decomposition level, a pre-computed unit control can be employed to establish the order of the computations in the buffers for a given filter-bank. Thus, several hardware implementations of this line-based strategy have been proposed, and they can be found in the literature [1] [4] [6]. However, a general case of this algorithm cannot be easily implemented in hardware or software due to the synchronization problems exposed above.

In addition, the line-based algorithm described in [5] is based on a filtering algorithm and not on the lifting scheme. The advantage of the lifting scheme in a line-based algorithm is not only the reduction in number of operations, but also the reduction in number of lines that the buffers showed in Figure 2 need to keep.

Finally, another issue to consider for a reversible integer-to-integer transform is the order of the inverse wavelet transform with respect to the forward one. In the regular wavelet transform, the entire image is available and then, it is easy to compute the inverse transform in the reverse row-column order that has been applied in the forward one. However, the line-based approach changes the order of the wavelet transform, and interleaves the horizontal and vertical DWT computation.

In the next section, we propose a general recursive algorithm that clearly specifies how to perform this communication among buffers, solving the synchronization problem in an automatic way by means of a recursive definition. We will present this algorithm with a lifting-based DWT, and we will tackle the problem of the correct order of the horizontal and vertical DWT so as to preserve the reversibility of the line-based approach.

IV. A RECURSIVE LINE-BASED IMPLEMENTATION FOR INTEGER-TO-INTEGER DWT

In this section, we present a forward and inverse wavelet transform algorithm (FWT and IWT) that solve the synchronization problems that have been addressed in the previous section. In order to overcome these drawbacks, both algorithms are defined with a recursive function that obtains the next low-frequency subband (LL) line from a contiguous level. The wavelet transform is implemented with the lifting-

```
function GetLLlineBwd( level L)
1) First base case:
    If there are no more lines to return at this level
            return EOL
2) Second base case:
    If $L = 0$
            return ReadImageLineIO( )
3) Recursive case
3.1) If $B_L$ is empty
            for $i = 2 \ldots 0$
                $B_L(i) =$ 1DFWT(GetLLlineBwd( $L-1$))
3.2) else if no more lines can be read from $L-1$
            SymmetricExtension( $B_L$ )
3.3) else
            repeat twice
                Shift( $B_L$ )
                $B_L(0) =$ 1DFWT(GetLLlineBwd($L-1$) )
    $B_L(1) = B_L(1) - \left\lfloor \frac{1}{2}(B_L(0) + B_L(2)) \right\rfloor$
    $B_L(2) = B_L(2) + \left\lfloor \frac{1}{4}(B_L(1) + B_L(3)) + \frac{1}{2} \right\rfloor$
    $\{LLline, HLline\} = B_L(2)$
    $\{LHline, HHline\} = B_L(1)$
    Process the high freq. lines $\{HLline, LHline, HHline\}$
    return $LLline$
end of fuction
```

```
function LowMemUsageFWT_B53( nlevel )
    set $B_L = empty \quad \forall L \in nlevel$
    repeat
        $LLline$ = GetLLlineBwd( nlevel )
        if ($LLline$!=EOL) Process the low freq. line( $LLline$ )
    until $LLline$=EOL
end of function
```

Algorithm 1: Recursive B5/3 FWT computation

```
function GetLLlineFwd( level L)
  $updateB_L = \neg updateB_L$
1) First base case:
    If there are no more lines to return at this level
        return EOL
2) Second base case:
    If $L = nlevel$
        return DecodeLLline( )
3) Recursive case
3.1) If $B_L$ is empty
        for $i = 1 \ldots 0$
            $B_L(i) =$ GetMergedLineFwd( $L+1$)
3.2) else if no more lines can be read from $L+1$ and $updateB_L$
        SymmetricExtension( $B_L$ )
3.3) else if $updateB_L$
        repeat twice
            Shift( $B_L$ )
            GetMergedLineFwd ($L+1$)
  if $updateB_L$
        $B_L(1) = B_L(1) - \left\lfloor \frac{1}{4}(B_L(0) + B_L(2)) + \frac{1}{2} \right\rfloor$
        $B_L(2) = B_L(2) + \left\lfloor \frac{1}{2}(B_L(1) + B_L(3)) \right\rfloor$
        return 1DIWT( $B_L(2)$ )
  else
        return 1DIWT( $B_L(1)$ )
end of function
```

```
subfunction GetMergedLineFwd( L )
    $odd_L = \neg odd_L$
    if $odd_L$ return { GetLLlineFwd( L ) + DecodeHLline( L ) }
    else return { DecodeLHline( L ) + DecodeHHline( L ) }
end of subfunction
```

```
function LowMemUsageIWT_B53( nlevel )
    set $B_L = empty \quad \forall L \in nlevel$
    set $odd_L = updateB_L =$ false $\quad \forall L \in nlevel$
    repeat
        imageLine = GetLLlineFwd( 0 )
        if (imageLine!=EOL) WriteImageLineIO(imageLine )
    until imageLine =EOL
end of function
```

Algorithm 2: Recursive B5/3 IWT computation

scheme, which is faster and requires less memory than the filtering algorithm. In addition, we will take the considerations needed to allow a reversible integer-to-integer decomposition. For the sake of simplicity, we will describe this algorithm using the B5/3 wavelet transform, which is probably the most widely used transform for integer implementation; however, this algorithm is valid for any integer wavelet transform.

A. Forward Wavelet Transform (FWT)

The main task of the FWT is carried out by a recursive function that successively returns lines of a low frequency (LL_n) subband at a given level (*n*). Thus, the whole FWT is computed by requesting LL lines at the last level (*nlevel*). As seen in Figure 2, the *nlevel* buffer must be filled with lines from the *nlevel*-1 level before it can generate lines. In order to get them, the function calls itself in a backward recursion, until the level zero is reached. At this point, it no longer needs to call itself since it can return an image line, which can be read directly from the input/output system. Notice that the buffer must be able to keep the lines to be predicted and updated in each step, and the lines from which these lines are predicted/updated. In the case of a B5/3 DWT, there are two prediction/update steps, and two additional lines are needed to compute them (the contiguous lines), so the buffer height must be four, as we will see later.

The function that implements this recursive algorithm is called GetLLlineBwd() (see Algorithm 1). This function receives a decomposition level as a parameter, calculates a line of each wavelet subband (LH, HL and HH) at that level, and returns a line from the low-frequency (LL) subband at that level. In order to get all the subband lines, the first time that this function is called at a certain level, it computes the first line of every subband at that level, the following time it computes the second one, and so forth.

When this function is called for the first time at a level (L), its buffer (represented by the variable B_L in Algorithm 1) is empty, and so it has to be recursively filled with lines from the previous level (case 3.1). Once a line is input, it must be transformed using a 1D DWT before inserting it into the buffer. On the other hand, if the buffer is not empty, it simply has to be updated by discarding two lines and introducing two additional lines from the contiguous level. We do it by means of a recursive call again (case 3.3). However, if there are no more lines from the previous level, this recursive call returns *End Of Line* (EOL) (case 3.2), in this case, if we need additional lines we duplicate them from the lines in the buffer using symmetric extension. In these three cases, once the buffer is updated, we predict the line at the buffer position 1, and then we update the line at the position 2. Notice that these steps can only be computed if the lines at position 0 and 3 are kept in the buffer. For this reason, the total buffer height must be 4 and not 2. This way, we compute a subband line from LH and HL (they are in the second line in the buffer), and from HL and HH (the first line in the buffer). The wavelet lines are processed and released depending on the application purpose (e.g., compression), and the function returns an LL line.

Every recursive function needs at least one base case to stop recursion. This function has two base cases. The first one is reached when all the lines at this level have been read. In this case, the function returns EOL. The second base case is reached when the backward recursion gets the level zero, and then no further recursive call is needed because an image line is read and returned directly from the I/O system.

Once we have defined this recursive function, we can compute the wavelet transform with *nlevel* decomposition simply by using this function to compute the whole LL_{nlevel} subband, as the function *LowMemUsageDWT_B53*(*nlevel*) does in Algorithm 1.

This algorithm can be implemented more easily because the synchronization among buffers and the problem of different buffer delays are solved directly with recursion, which automatically sets the rhythm of the transformation steps.

B. Inverse Wavelet Transform (IWT)

The inverse DWT algorithm (IWT), which is described in Algorithm 2, is similar to the forward one, but applied in reverse order.

Since the recursive function goes forward, the second base case is changed from the FWT to be reached when the parameter level is equal to *nlevel*, and then a line from the low-frequency subband LL_{nlevel} is retrieved directly from I/O.

In the recursive case, there are mainly three changes with respect to the FWT. The first modification is the introduction of a new function, *GetMergedLineFwd*(*level*), which is used to get buffer lines. This function alternatively returns the

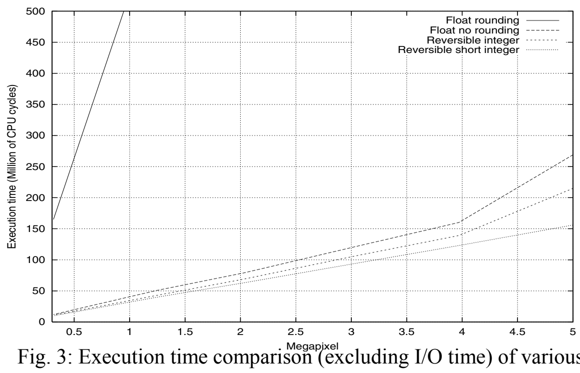

Fig. 3: Execution time comparison (excluding I/O time) of various implementations using float (with and without rounding), integer and short integer coefficients, with the B5/3 transform and the lifting proposal.

concatenation of a line from the LL and HL subbands, or from the LH and HH subbands, at a specified level. Contrary to the lines from HL, LH and HH, which are retrieved directly from I/O, the LL line is computed recursively from the following level, and therefore this is the recursive point in the function. The second difference is the introduction of a logical variable, $updateB_L$, which defines whether the buffer needs to be updated or not. In the IWT, two LL lines can be computed once a buffer is full or updated. Therefore, this variable shows if the buffer is updated, and if so, another line can be computed without updating it. Finally, the third modification aims to guarantee reversibility. Despite not having the whole image in memory, we still can take a reversible approach. In Algorithm 1, the forward transform is performed first horizontally, when a line is input, and then vertically, by applying one step of the wavelet transform. Thereby, for a reversible transform, the order of the inverse transform has to be changed. The horizontal 1-D IWT is not applied once a compound line is read (in GetMergedLineFwd() function) and introduced into the buffer, but it is delayed until the end of the GetLLlineFwd() function (see Algorithm 2). This way, we follow the correct order (i.e., horizontal FWT, vertical FWT, vertical IWT, horizontal IWT) and the transform is fully reversible.

A drawback that has not been considered yet is the need to reverse the order of the subbands, from the FWT to the IWT. The former starts generating lines from the first levels to the last ones, while the latter needs to get lines from the last levels before getting lines from the first ones. This problem can be solved using some additional buffers at both ends to reverse the coefficients order, so that data are supplied in the right order [5]. Other simpler solutions are: to save every level in secondary storage separately so that it can be read in a different order and, if the WT is used for image compression, to keep the compressed coefficients in memory.

V. EXPERIMENTAL RESULTS

In order to compare the regular wavelet transform and our proposals, we have implemented them, using standard ANSI C language on a regular PC computer with 256 KB L2 cache. Moreover, the coefficients for the transform are implemented as floating-point, integer and short integer values in order to assess the effects of employing different data types.

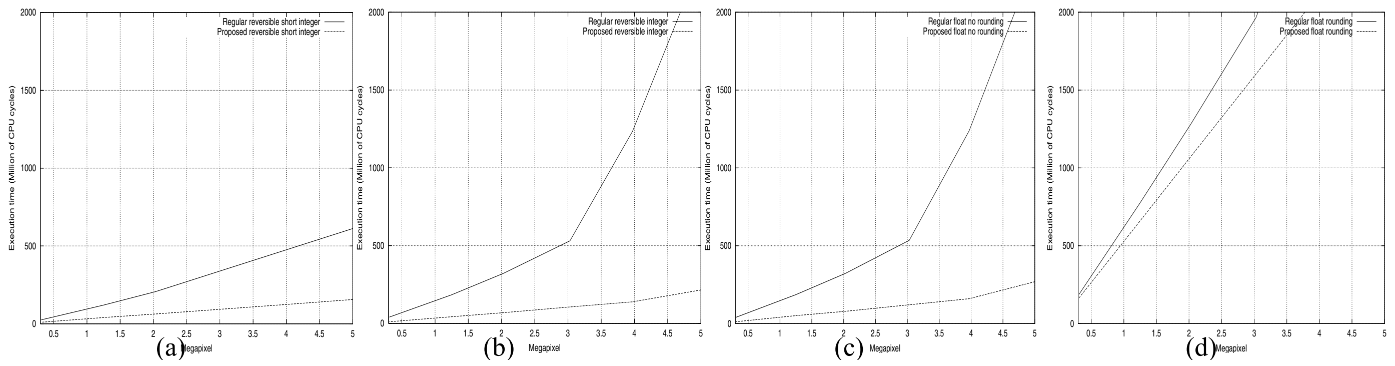

Fig. 4: Execution time comparison (excluding I/O time) of the regular wavelet transform and the lifting proposal, applying the B5/3 transform, with (a) short-integer coefficients, (b) integer coefficients, (c) floating-point arithmetic without rounding, and (d) floating-point arithmetic with rounding.

For these tests, we have used the standard Lena (512x512) and Woman (2048x2560) images. With six decomposition levels, the regular WT needs 1030 KB for Lena and 20510 KB for Woman, while our proposal using 32-bit data types requires 19 KB for Lena and 79 KB for Woman, i.e., it uses 54 and 260 times less memory. If short-integer data type is employed, it requirements are reduced to 9 KB and 39 KB respectively.

In addition, Table 1 shows that our proposal is much more scalable than the usual DWT. In this table, we present the amount of memory needed to apply the transform to images ranging from low-resolution VGA to 20-Megapixel with various data types.

TABLE I. MEMORY REQUIREMENT (KB) COMPARISON

Image size (megapixel)	Regular WT (float and integer)	Regular WT (short integer)	Proposed lifting (float and integer)	Proposed lifting (short integer)
20 (4096 x 5120)	81,980	40,990	158	79
16 (3712 x 4480)	65,013	32,506	143	71
12 (3200 x 3968)	49,647	24,823	123	62
8 (2560 x 3328)	33,319	16,659	98	49
5 (2048 x 2560)	20,510	10,255	79	39
4 (1856 x 2240)	16,266	8,133	71	36
3 (1600 x 1984)	12,423	6,211	61	31
2 (1280 x 1664)	8,340	4,170	49	24
1.2 (1024 x 1280)	5,125	2,562	39	19
VGA (512 x 640)	1,288	644	19	9

In Figure 4, we present an execution time comparison of different data types using our proposal. The most noticeable result in this graph is the high execution time of the floating-point implementation that uses the floor operator, that is to say, with rounding, due to the temporal complexity of this operation. If we avoid using rounding (we can simply omit it when the target variable of the operation is floating-point), the execution time of the floating-point implementation is significantly reduced, although being still above the implementations with integer or short-integer coefficients.

Finally, in Figure 5, an execution time comparison between the regular wavelet transform and our proposal is given for different data types. Figure 5(b) (integer implementation) and 5(c) (floating-point without rounding) show that, while our algorithms display linear behavior, the regular wavelet transform approaches to an exponential curve. This behavior is mainly due to the ability of our algorithms to fit in cache for all the image sizes. On the contrary, the usual wavelet transform rapidly exceeds the cache limits. In the short-integer implementation, the lower memory usage of the regular wavelet transform prevents the exponential behavior. In the last graph, where floating-point arithmetic is used with floor operations, the cost of rounding causes both the regular and proposed transform to be highly complex.

VI. CONCLUSIONS

A new reversible integer-to-integer lifting algorithm has been presented that solves the existing problems about different delays and rhythm among buffers in the line-based approach. It can be used as a part of compression algorithms, like JPEG 2000, speeding up its execution time and reducing its memory requirements compared with the usual DWT algorithm.

REFERENCES

[1] T. Acharya, P. Tsai, JPEG 2000 Standard for Image Compression: Concepts, Algorithms and VLSI Arquitectures, Chapter 5, Wiley, Oct 2005.

[2] M. Adams, F. Kossentini, Reversible Integer-to-Integer Wavelet Transforms for Image Compression: Performance Evaluation and Analysis, IEEE Transactions on Image Processing, vol. 9, pp. 1010-1024, June 2000.

[3] R. C. Calderbank, I. Daubechies, W. Sweldens, B. L. Yeo, Wavelet transforms that map integers to integer, Journal of Applied Computational and Harmonic Analysis, vol. 5, pp. 332-369, 1998.

[4] W. Chang, Y. Lee, W. Peng, C. Lee, A Line-Based, Memory Efficient and Programmable Architecture for 2D DWT using Lifting Scheme, International Symposium on Circuits and Systems (ISCAS), 2001.

[5] C. Chrysafis, A. Ortega, Line-based, reduced memory, wavelet image compression, IEEE Trans. on Image Processing, vol. 9, March 2000.

[6] G. Dillen, B. Georis, J. Legat, O. Cantineau, Combined Line-Based Architecture for the 5-3 and 9-7 Wavelet Transform of JPEG 2000, IEEE Trans. on Circuits and Systems for Video Technology, vol. 13, Sept 2003.

[7] S. Mallat, A Theory for Multiresolution Signal Decomposition, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 11, July 1989.

[8] W. Sweldens, The lifting scheme: a custom-design construction of biorthogonal wavelets, Journal of Applied Computational and Harmonic Analysis, vol. 3, pp. 186-200, 1996.