# On the Design of Fast Wavelet Transform Algorithms With Low Memory Requirements

Jose Oliver, *Member, IEEE*, and Manuel Perez Malumbres, *Member, IEEE*

***Abstract*—In this paper, a new algorithm to efficiently compute the two-dimensional wavelet transform is presented. This algorithm aims at low memory consumption and reduced complexity, meeting these requirements by means of line-by-line processing. In this proposal, we use recursion to automatically place the order in which the wavelet transform is computed. This way, we solve some synchronization problems that have not been tackled by previous proposals. Furthermore, unlike other similar proposals, our proposal can be straightforwardly implemented from the algorithm description. To this end, a general algorithm is given which is further detailed to allow its implementation with a simple filter bank or using the more efficient lifting scheme. We also include a new fast run-length encoder to be used along with the proposed wavelet transform for fast image compression and reduced memory consumption. When a 5-megapixel image is transformed, experimental results show that the proposed wavelet transform requires 200 times less memory and is five times faster than the regular one. If we consider the whole coding system, numerical results show that it achieves state-of-the-art performance with very low memory requirements and fast execution, becoming an interesting solution for resource-constrained devices such as mobile phones, digital cameras, and PDAs.**

***Index Terms*—Image coding, optimization methods, run length codecs, wavelet transform computation.**

## I. Introduction

THE discrete wavelet transform (DWT) is a new mathematical tool that has aroused great interest in the field of image processing due to its nice features. For example, it allows image multiresolution representation in a natural way, we can analyze the wavelet coefficients in both space and scale domains and, for natural images, the DWT achieves high compactness of energy in the lower frequency subbands, which is very useful in image compression. Thus, while the popular standard for image compression JPEG uses the discrete cosine transform (DCT), the new JPEG 2000 standard [1] proposes the use of the wavelet transform, since it offers better rate/distortion (R/D) performance, avoids blocking artifacts typical of block-based coders, and eases multiresolution.

Manuscript received August 25, 2005; revised February 16, 2007. This work was supported by the Spanish Ministry of Education and Science under Grant TIC2003-00339. This paper was recommended by Associate Editor R. Chandramouli.

J. Oliver is with the Department of Computer Engineering (DISCA), Polytechnic University of Valencia, 46022 Valencia, Spain (e-mail: joliver@disca.upv.es).

M. Perez Malumbres is with the Department of Physics and Computer Engineering, Miguel Hernandez University, 03202 Elche, Spain (e-mail: mels@umh.es).

Digital Object Identifier 10.1109/TCSVT.2007.913962

Unfortunately, despite the benefits that the wavelet transform entails, some other problems are introduced. Wavelet-based image processing systems are typically implemented by memory-intensive algorithms, with higher execution time than other transforms. In the usual DWT [2], the image decomposition is computed by means of convolution filtering and so, its complexity rises as the filter length increases. Moreover, in the regular DWT computation, the image is transformed first row by row and then column by column at every decomposition level, and hence it must be kept entirely in memory. These problems are not as noticeable in other transforms. For example, the DCT is applied in small block sizes and thus a large amount of memory is not specifically needed. In addition, due to the current importance of DCT-based coders, the complexity of the DCT is a well-studied issue [3], and many proposals exist to reduce the complexity of the Inverse [4] and Forward [5] DCT.

The memory requirement of the wavelet transform may seriously affect memory-constrained devices that deal with digital images, such as digital cameras and personal digital assistants (PDAs). The complexity of the wavelet transform is another issue that affects these devices, since they usually contain DSP or processors with lower computational power than regular desktop workstation processors. Both memory and complexity of the DWT impose severe restrictions on applications running on this kind of device, in terms of required working memory and processing time.

Since memory use and execution time of the DWT computation grow linearly with the image size, even high-performance workstations with plenty of memory can find it difficult to deal with the wavelet transform of large-scale images. This way, in a geographical information system (GIS), where large digital maps are handled [6], the uncompressed color map of the Iberian Peninsula (581 000 square meters approx.) needs more than 1.6 Terabytes to be stored (scale 1 pixel:1 square meter). If we use a DWT-based coder, we will need a prohibitive amount of memory to perform the regular DWT computation.

As mentioned earlier, the regular one–dimensional (1-D) DWT is computed by means of filtering operations. For a first decomposition level, and a given filter bank (which is defined by the selected wavelet family), the input samples are separately filtered by the lowpass and highpass filters from the filter bank, and then, the resulting transform coefficients are downsampled by two. Note that for finite-length sample sets, we need to specify the value of those samples beyond the signal limits in order to operate around the boundaries. An efficient option to prevent the appearance of artificial high frequency is symmetric extension, in which the samples around both ends

are copied in reverse order to extend the signal (e.g., a 3-sample long extension of the set **abcdefg** is *dcb***abcdefg***fed*).

As a result of the wavelet decomposition, the signal spectrum is split into a low-frequency $(L_1)$ and a high-frequency $(H_1)$ subband. Since in natural signals most energy concentrates in $L_1$, the same decomposition process can be recursively applied on the remaining low-frequency subbands (which is known as a dyadic decomposition), until a desired decomposition level $(N)$ is achieved. For a two-dimensional (2-D) wavelet transform, the 1-D DWT is separately applied on rows and columns, resulting in four wavelet subbands, which are usually called $\mathrm{LL}_1$, $\mathrm{LH}_1$, $\mathrm{HL}_1$, and $\mathrm{HH}_1$. The first letter in the subband name identifies the filter that has been applied horizontally, the second letter identifies the vertical filtering, and the number identifies the decomposition level. As in the 1-D case, the wavelet decomposition is recursively repeated on the low-frequency subband $(\mathrm{LL}_1)$ until the desired decomposition level is reached.

The lifting scheme [7], [8] is probably the best-known algorithm to calculate the wavelet transform in a more efficient way, as an alternative to the regular convolution method. Since it uses fewer filter coefficients than the equivalent convolution filter, it provides a faster implementation of the DWT. This scheme also provides memory reduction through in-place computation of wavelet coefficients. However, if in-place computation is applied, the low-frequency coefficients are interleaved with the high-frequency coefficients, and the subsequent wavelet processing can be non-optimal (especially in cache-based systems), requiring more careful processing. We can overcome this problem with coefficient reordering, at the cost of increasing the complexity of the algorithm.

Besides lifting, another way to speed up the execution time in cache-based architectures is to optimize the memory access. The regular 2-D wavelet transform computes the 1-D transform first on each row, and then on each column. This process is successively repeated on the low-frequency subbands. Due to the organization of images in memory (usually row by row), the column access does not exploit memory locality, and hence does not take advantage of the cache memory. If we can arrange the memory access strictly on rows, we will be able to improve the cache performance [14]. We will take this approach in our algorithm.

Other works have been focused on efficient hardware designs. In this manner, in [28] two efficient VLSI architectures are proposed to compute the DWT. One of these proposals is based on a folded architecture in which latency is reduced at the expense of increasing the hardware area, although it is not completely utilized. The second proposal is a digit-serial architecture, which requires simpler circuitry and lower power, but has higher latency.

Let us address now the problem of memory consumption. The simplest solution to reduce the amount of memory needed to compute the wavelet transform of an image is to split the whole image into smaller pieces, so that the DWT can be calculated on each one separately. This approach is called image tiling and is supported by JPEG 2000. However, it presents several problems. On the one hand, we have image blocks again, and then blocking artifacts may reappear, especially when we use small tile size and high quantization. On the other hand, we do not decorrelate the entire image, but only the part that is being transformed, and then the compactness is lower. In JPEG 2000, this low compactness causes the PSNR to drop by more than 1 dB at low bit rates, with a tile size of $128 \times 128$ instead of the whole image [9].

However, there are other strategies to save more memory. One of them is to get rid of wavelet coefficients as soon as they have been calculated. With this approach, we can compute all the decomposition levels simultaneously, and when a coefficient is not going to be used anymore, it is discarded (compressed, saved or processed according to the purpose of the wavelet decomposition). In the rest of this section, we survey some of these strategies, focusing on the line-based approach.

### A. Related Work

Several proposals have dealt with low-memory DWT implementations. In [10], it is introduced a first solution to overcome this drawback for the 1-D DWT. Later, one of the first approaches to reduce memory consumption in wavelet processing was done in [11]. The proposed algorithm includes image coding by means of zerotrees [12], [13]. To reduce the memory requirements, the encoder reorders the output bit stream so that wavelet coefficients from several subbands are placed together, and this way, the decoder can compute a fragment of the inverse DWT, and produce several image lines. Once this group of lines is decoded, the memory used by these coefficients can be released and more lines can be read in the same way.

The first line-based algorithm was proposed in [14], where 1) reduction of memory is dealt in both the forward and inverse transform (in [11] it is done only in the decoder); 2) the order of the coefficients is rearranged with some extra buffers to allow efficient use of memory in the encoder and the decoder; and 3) the zerotree encoder is replaced with a new entropy coding algorithm.

In [15], a block-based implementation of the DWT was proposed in order to match with block-based encoders and to reduce the memory usage of the wavelet transform. Although it is an interesting proposal for block-based encoders, the required memory resources are similar to those in [14]. This work also includes a block-based encoder that is able to get state-of-the-art coding performance, but the encoder is quite complex and requires too much memory.

Finally, in [16], the authors show the importance of a good use of cache memory, reducing the computation time of the DWT by means of proper memory organization over the spatial combinative lifting algorithm (SCLA) proposal.

### B. The Line-Based Approach

Recall that in the regular DWT, the image is transformed level by level, by using the 1-D DWT first on rows, and then on columns, and so it must be kept entirely in memory. In order to keep in memory only the part of image strictly necessary, and therefore to reduce the amount of memory required, the order of the regular wavelet algorithm must be changed. We cannot compute every decomposition level successively but this computation has to be interleaved. In this section, we describe the line-based DWT using a different approach from the one used in [14].

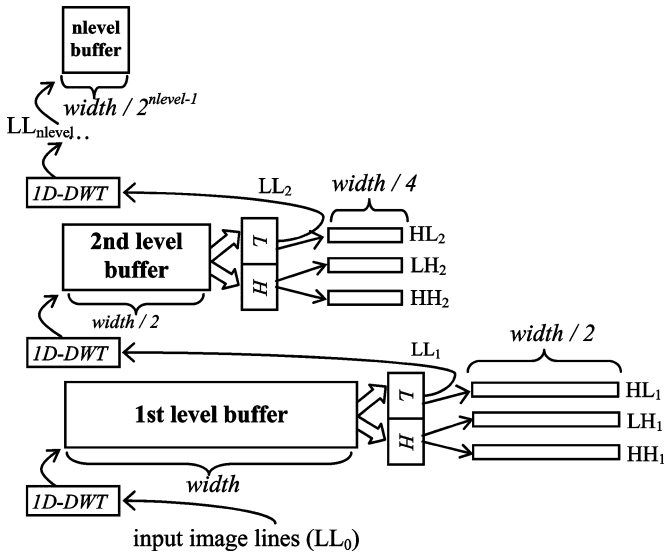


Fig. 1. Overview of a line-based forward wavelet transform. The boxes labeled $L$ and $H$ represent the lowpass and highpass vertical filtering. The box labeled *1D-DWT* indicates that a line is transformed as usual.

For the first decomposition level, the algorithm directly receives image lines, one by one. On every input line, a one-level 1-D wavelet transform algorithm is applied so that it is divided into two parts, representing the horizontal details and a low-frequency smaller version of this line. Then, these transformed lines are stored in a first decomposition level buffer. When there are enough lines in the buffer to perform a step of a column wavelet transform, the vertical transform is computed and the first line of the $\mathrm{HL}_1$, $\mathrm{LH}_1$, and $\mathrm{HH}_1$ wavelet subbands, and the first line of the $\mathrm{LL}_1$ subband are calculated. At this moment, for a dyadic wavelet decomposition, we can process (e.g., compress) and release the first line of every wavelet subbands. However, the first line of the $\mathrm{LL}_1$ subband is not part of the result but it is needed as incoming data for the following decomposition level. When the lines in the first level buffer have been employed, this buffer is shifted twice (using a rotation operation) so that two lines are discarded while another two image lines are input at the other end. Note that the downsampling by two inherent to the wavelet transform (see the Introduction) is achieved by shifting the buffer by two positions instead of by one. Once the buffer is updated, the process can be repeated and more lines are obtained.

At the second level, the buffer is filled with the $\mathrm{LL}_1$ lines that have been computed in the first level. Once the buffer is completely filled, it is processed in the same way as we have described for the first level. In this manner, the lines of the second wavelet subbands are calculated, and the low-frequency lines from $\mathrm{LL}_2$ are passed to the third level. As it is depicted in Fig. 1, this process can be repeated until the desired decomposition level (*nlevel*) is reached.

In [14], 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. The main drawback is the synchronization among the buffers. Before a buffer can produce lines, it must be completely filled with lines from previous buffers, 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 [17]–[20]. However, a general case of this algorithm cannot be easily implemented in software due to the synchronization problems exposed above. In our proposal, we will take a different approach to solve this problem in software by using a recursive definition that will be detailed later.

### C. Contributions and Paper Organization

In Section II, we propose a general recursive algorithm to compute the DWT in a line-based fashion [11], [14]. We give two implementations of this recursive algorithm, first by means of a simple filter bank, and later using the lifting scheme to improve its efficiency. Both convolution and lifting algorithms are fully described and can be straightforwardly implemented. In fact, we give an implementation in ANSI C language [21], which differs little from the pseudo-code description. While in the iterative approach [14] we need to handle several buffers with different delay and rhythm, which is a difficult task, the main contribution of our proposal is to solve the problem of how to perform this communication among buffers. Moreover, the proposal based on the lifting scheme reduces the memory requirements by half when compared to the filter bank implementation, and improves the execution time.

Although the wavelet transform that we describe in this paper is general purpose, we will use it in an image compression environment. An important restriction is introduced by the fact that only a part of each subband is available at every moment, and therefore we need coding schemes that do not require global knowledge of the image. Moreover, progressive coding (with SNR scalability) is not possible in just one-pass, since we have to encode an entire coefficient as soon as we receive it. Therefore, multiple image scans focusing on a different bit-plane is not feasible, unless we rearrange the coefficients later on, after encoding all the coefficients, as JPEG 2000 does.

The line-based DWT that we propose in Section II can be used along with JPEG 2000 in a standard coding/decoding system. However, the block-coding algorithm used in JPEG 2000 (EBCOT [22]) is rather complex, due to the use of bit-plane coding with a rate-distortion optimization algorithm and a large number of contexts. Since we aim at low complexity, in Section III we will depict a new run-length wavelet encoder (RLW) to be used along with the proposed wavelet transform. This RLW coder presents good compression performance, is fast and does not introduce memory overhead, so it meets the requirements with which we are dealing.

Note that run-length coding has been used several times in wavelet-based compression (see [23], [24]), and even in the

cleanup pass of JPEG 2000's EBCOT [22], but these proposals always work with bit-plane coding. Run-length coding is a good way to achieve fast compression, but we are going to speed it up by encoding the coefficients as a whole. Moreover, due to the use of a line-based DWT, with neither rearrangement nor post-processing of the wavelet coefficients, we do not aim at SNR scalability, and therefore bit-plane coding is not necessary.

Finally, in Section IV, we compare the proposed implementations of the DWT with the regular wavelet transform in terms of memory usage and execution time by using real implementations. In the same section, the proposed coder is also compared with state-of-the-art image coders like SPIHT and JPEG 2000.

## II. Recursive Algorithm for Buffer Synchronization

In this section, we present the Forward and Inverse Wavelet Transform algorithms (FWT and IWT) that solve the synchronization problems that have been addressed in the introduction. In order to solve these problems, 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 first by a simple filter bank, and then using the lifting scheme, which is faster and requires less memory.

### A. General Algorithm

Let us depict our algorithm briefly. The main task of the FWT is carried out by a recursive function that successively returns lines of a low-frequency $(\mathrm{LL}_i)$ subband at a given level $(i)$. The FWT is computed by requesting LL lines at the last level (*nlevel*). As seen in Fig. 1, the *nlevel* buffer must be filled up 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 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. Although we are calculating a forward wavelet transform, we do it by means of backward recursion, since we go from *nlevel* to zero.

The function that implements this recursive algorithm is called GetLLlineBwd() (see Algorithm 1 in Fig. 2). This function receives a decomposition level $i$ as a parameter, and calculates a line of each wavelet subband ($\mathrm{LH}_i$, $\mathrm{HL}_i$ and $\mathrm{HH}_i$) and returns a line from the low-frequency $(\mathrm{LL}_i)$ subband. In order to get the subband lines at level $i$, the first time that this function is called at that level, it computes the first line of every subband at level $i$, the following time it computes the second one, and so forth.

When this function is called for the first time at level $i$, its buffer $(buffer_i)$ is empty, and so it has to be recursively filled with lines from the previous level $i-1$ as shown in Algorithm 1 (case 3.1). Recall that once a line is input, it must be transformed using a 1-D 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 some lines and introducing additional lines from level $i-1$. We do it by means of a recursive call again (case 3.3). However, if there are no more lines available at level $i-1$, this recursive call returns *End Of Line* (EOL). That points out that we are about to finish the computation at this level, but we still have to calculate some subband lines from the remaining lines in the buffer (case 3.2). We will give more details of each

```
function GetLLlineBwd( i )
1) First base case:
    If there are no more lines to return at this level (i)
        return EOL
2) Second base case:
    If i = 0
        return ReadImageLineIO( )
3) Recursive case
3.1) If buffer_i is empty
        Fill up buffer_i calling GetLLlineBwd(i-1)
3.2) else if no more lines can be read from i-1
        Start cleaning buffer_i
3.3) else
        Update buffer_i calling GetLLlineBwd(i-1)
    Get subband lines from buffer_i
    Process the high freq. subband lines {HLline, LHline, HHline}
    return LLline
end of fuction
```

```
function LowMemUsageFWT( nlevel )
    set buffer_i = empty    ∀i ∈ [1 ... nlevel ]
    repeat
        LLline = GetLLlineBwd( nlevel )
        if (LLline!=EOL) Process the low freq. line( LLline )
    until LLline=EOL
end of function
```

Fig. 2. Algorithm 1: Recursive FWT computation with *nlevel* decompositions. The backward recursive function $GetLLlineBwd(i)$ returns a line from the low-frequency subband $\mathrm{LL}_i$. The first time that this function is called at level i, it returns the first line of the $\mathrm{LL}_i$ subband, the following time it returns the second line, etc. If there are no more lines at level i, it returns the EOL tag. As the $n$th line of the $\mathrm{LL}_i$ subband is computed and returned, the corresponding $n$th lines of the $\mathrm{HL}_i$, $\mathrm{LH}_i$ and $\mathrm{HH}_i$ subbands are also computed, processed and released.

recursive case in Sections II-B–D, since they are handled in a different way depending on whether we are dealing with a convolution algorithm or with the lifting scheme. In both convolution and lifting, we have a wavelet subband line from $\mathrm{LH}_i$, $\mathrm{HL}_i$ and $\mathrm{HH}_i$ at the end of the recursive case. These lines are processed and released depending on the application purpose (e.g., compression), and the function returns an $\mathrm{LL}_i$ line.

Every recursive function needs at least one base case to stop recursion. This function has two base cases. The first case is 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 the recursive function, we can compute a wavelet transform with *nlevel* decompositions simply by using this function to compute the whole $\mathrm{LL}_{nlevel}$ subband. This is done by the function *LowMemUsageDWT*(*nlevel*) in Algorithm 1, which calls *GetLLlineBwd*(*nlevel*) until it returns EOL.

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

```
function GetLLlineFwd( i )
    updatebuffer_i = ¬updatebuffer_i
1) First base case:
    If there are no more lines to return at this level (i)
        return EOL
2) Second base case:
    If i = nlevel
        return DecodeLLline( )
3) Recursive case
3.1) If buffer_i is empty
        Fill up buffer_i calling GetMergedLineFwd (i+1)
3.2) else if no more lines can be read from i+1 and updatebuffer_i
        Start cleaning buffer_i
3.3) else if updatebuffer_i
        Update buffer_i calling GetMergedLineFwd (i+1)
    if updatebuffer_i
        Get the first LLline from updated buffer_i
    else
        Get the second LLline from updated buffer_i
    return LLline
end of function
```

```
subfunction GetMergedLineFwd( i )
    odd_i = ¬odd_i
    if odd_i
        return { GetLLlineFwd( i ) + DecodeHLline( i ) }
    else
        return { DecodeLHline( i ) + DecodeHHline( i ) }
end of subfunction
```

```
function LowMemUsageIWT( nlevel )
    set buffer_i = empty     ∀i ∈ [1...nlevel]
    set odd_i = updatebuffer_i = false     ∀i ∈ [1...nlevel]
    repeat
        imageLine = GetLLlineFwd( 0 )
        if (imageLine!=EOL) WriteImageLineIO(imageLine )
    until imageLine =EOL
end of function
```

Fig. 3. Algorithm 2: Recursive IWT computation with *nlevel* decompositions. The forward recursive function $GetLLlineFwd(i)$ returns a line from a low-frequency subband as Algorithm 1 does, although using forward recursion. First, it retrieves a line of the $\mathrm{HL}_{i+1}$, $\mathrm{LH}_{i+1}$ and $\mathrm{HH}_{i+1}$ subbands from the compressed bit stream, and an $\mathrm{LL}_{i+1}$ line from the following level using a recursive call. Then, with these lines, it can compute two lines of the $\mathrm{LL}_i$ subband and return them alternatively.

sets the rhythm and order of the transformation steps. The iterative alternative is more difficult because a simple nested loop is not enough, and a complicated control to trigger the operations at the correct moment for each level is required.

The inverse DWT algorithm (IWT), which is described in Algorithm 2 in Fig. 3, is similar to the forward one, but applied in reverse order. Thus, it carries out forward recursion, from zero to *nlevel*, and so it builds a low-frequency line at level $i$ $(\mathrm{LL}_i)$ from an $\mathrm{LL}_{i+1}$ line, which is computed recursively from $i+1$, along with the corresponding $\mathrm{LH}_{i+1}$, $\mathrm{HL}_{i+1}$ and $\mathrm{HH}_{i+1}$ lines, which are input from the compressed bit stream. Since the recursive function goes forward, the second base case is changed to be reached when the parameter level is equal to *nlevel*, and then a line from the low-frequency subband $\mathrm{LL}_{nlevel}$ is retrieved directly from the compressed bit stream. In the recursive case, there are mainly two changes with respect to the backward function. The first modification is the introduction of a new function $GetMergedLineFwd(i)$, which is used to get the lines for the buffer. This function alternatively returns the concatenation of a line from the $\mathrm{LL}_i$ and $\mathrm{HL}_i$ subbands, or from the $\mathrm{LH}_i$ and $\mathrm{HH}_i$ subbands, at a specified level $i$. Contrary to the lines from $\mathrm{HL}_i$, $\mathrm{LH}_i$ and $\mathrm{HH}_i$, which are retrieved directly from the compressed bit stream, the $\mathrm{LL}_i$ line is computed recursively using *GetLLlineFwd()*. The second difference is the introduction of a logical variable, $updatebuffer_i$, which defines whether the buffer needs to be updated or not to produce another line. In the IWT, two lines can be computed once a buffer is full. Therefore, this variable shows if the buffer is updated and if so, another line can be computed without updating it. More details on how the recursive case is implemented are given in Sections II-B–D.

Once the recursive function for the IWT is defined, all the image lines can be computed just by calling this recursive function, with the $i$ parameter set to zero, until no more lines are available.

### B. Filter Bank Implementation

In the previous subsection, we described a general recursive algorithm for the DWT computation. Now, we will use that description to implement the DWT computation using a filter bank. In this implementation, we insert lines into the buffer until we can apply one-step of a vertical lowpass and a vertical highpass filter bank on it. Therefore, each buffer must be able to keep $2N+1$ lines, where $2N+1$ is the number of taps for the largest filter bank (lowpass and highpass filters). We only consider odd filter lengths because they have higher compression efficiency, however this analysis could be extended to even filters as well.

Since the base cases are completely defined in Algorithm 1, we only have to describe the recursive case. In this case, when a buffer is empty (case 3.1), its upper half (from $N$ to $2N$) is recursively filled with lines from the previous level. Once the upper half is full, the lower half is filled using symmetric extension (in which the $N+1$ line is copied into the $N-1$ position, the $N+2$ into the $N-1$, ..., the $2N$ is copied into the 0 position). On the other hand, if the buffer is not empty, we only have to update it (case 3.3). Therefore, we shift it by one position so that the line contained in the first position is discarded and a new line can be introduced in the last position $(2N)$ by using a recursive call. This operation is repeated twice because after applying the highpass and lowpass filters with the lines in the buffer, we discard the results of the following filtering operation (which could be performed with the lines in the buffer after the first shift) to efficiently implement a downsampling by two. Finally, if no more lines can be computed from the previous level (case 3.2), we fill the buffer by using symmetric extension again. Actually, this is the same case as in 1-D DWT, in which symmetric extension is used at both ends.

In all the cases, once there are enough lines in the buffer to perform one step of a column wavelet transform, the convolution process is calculated vertically twice, first using the lowpass

*3.1)* **if** $buffer_i$ is empty
  **for** $j = N \ldots 2N$
    $buffer_i(j) = 1\text{DFWT}(\text{GetLLlineBwd}(i\text{-}1))$
  FullSymmetricExtension($buffer_i$)
*3.2)* **if not**( more_lines( $i$-1) )
  **repeat** twice
    Shift($buffer_i$)
    $buffer_i(2N) = \text{SymmetricExt}(buffer_i)$
*3.3)* **else**
  **repeat** twice
    Shift($buffer_i$)
    $buffer_i(2N) = 1\text{DFWT}(\text{GetLLlineBwd}(i\text{-}1))$
For *3.1)*, *3.2)* and *3.3)*
  $\{LLline, HLline\} = \text{ColumnFWT\_LowPass}(buffer_i)$
  $\{LHline, HHline\} = \text{ColumnFWT\_HighPass}(buffer_i)$

Fig. 4. Algorithm 3: Filter bank implementation, recursive case.

filter and then with the highpass filter. This way, we get a line of every wavelet subband. This whole process is described in Algorithm 3 (Fig. 4).

For the inverse transform, we need to change Algorithm 3 slightly. We only have to replace every *1-DFWT()* function by *1-DIWT()*, change all $i - 1$ for $i + 1$, and use *GetMergedLineFwd()* instead of *GetLLlineBwd()*. Then, we can incorporate this modified version of Algorithm 3 as the recursive case in Algorithm 2, but recall that we have to use the $updatebuffer_i$ variable to execute cases 3.2 and 3.3 only when needed (as described in Algorithm 2), and to compute an $\text{LL}_i$ line using the $ColumnIWT_{LowPass}(buffer_i)$ and $ColumnIWT_{HighPass}(buffer_i)$ functions alternatively (the lowpass filter is applied when $updatebuffer_i$ is true).

### C. Implementation With the Lifting Scheme

The convolution implementation that has been presented in the previous subsection introduces wide benefits in memory usage, since we only keep in memory a few low-frequency lines $(2N + 1)$ for each decomposition level instead of the whole subbands. However, we can still reduce the amount of memory required and speed up its execution time by using the lifting scheme [7].

In the lifting scheme, the wavelet coefficients are computed by means of several steps on the input samples (see Fig. 5). In the first step, the samples in odd positions (black squares in the figure) are processed from the contiguous even samples (the white ones). This way, we try to predict each odd sample as a linear combination of the even ones, and thus this step is called prediction step. In the second step, the even values are computed from the contiguous odd ones, and it is called update step. In this manner, we compute successive prediction and update steps. The total number of steps depends on the DWT transform that is being computed. Finally, the odd values calculated in the last prediction step are normalized by a constant factor $(K)$ to achieve the high-frequency wavelet coefficients. The values from the last update step are normalized by $1/K$ to get the low-frequency coefficients.

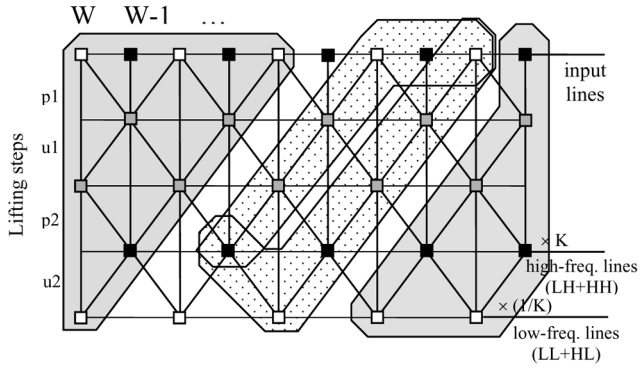


Fig. 5. Overview of the lifting scheme for the B9/7 FWT.

TABLE I
WEIGHTING VALUES FOR PREDICTION AND UPDATE STEPS OF THE LIFTING SCHEME IN THE B9/7 WAVELET TRANSFORM

| | | | |
|---|---|---|---|
| $p_1$ | -1.586134342 | $u_1$ | -0.052980119 |
| $p_2$ | 0.882911076 | $u_2$ | 0.443506852 |
| K for (1,2) normalization | | 1.230174104914 | |
| K' for ($\sqrt{2}$ ,$\sqrt{2}$ ) normalization | | K / $\sqrt{2}$ | |

The normalization constant $K$ depends on the desired features of the transformation (close to orthonormal, preserve the dynamic range of the coefficients, etc.). The lifting scheme depicted in Fig. 5 is for the popular B9/7 transform [25]. The derivation from the filter bank to the weighting factors for every prediction and update step is given in [26], and the numeric values for these weighting factors are shown in Table I.

The main advantage of the use of the lifting scheme instead of convolution is the extra reduction of memory achieved. Let us define $W$ as the total number of weighting factors (prediction and update) for a DWT. Then, the buffer height in the lifting scheme has to be $W+2$, so it can perform the $W$ prediction and update steps needed to compute a low and a high-frequency line in a segmented way, as we will see later. Despite computing $W$ sample lines, we need two additional lines to calculate the first and last values. We will see that these additional lines are read but not modified in the step in which they are read. In general $W+2$ is lower than $2N+1$ (see [26] for details) and therefore we need less lines in the buffers. For example, for B9/7, $2N+1$ is 9 while $W+2$ is only 6. For the sake of clarity, in the rest of this section, we will consider that the number of sample values in each decomposition level is even, and so $W$ is (although it is not difficult to extend it to the general case).

In Algorithm 4 (Fig. 6), we describe how to implement the recursive case of Algorithm 1 using the lifting scheme. In this algorithm, when the buffer is empty (case 3.1), we fill it from $W$ to 0 ($W + 1$ is left empty), using a recursive call. Then, we compute the successive prediction and update steps, using only the lines in the buffer. This way, we compute fewer lines in every step, since the rest of lines rely on information that still has not been input. Finally, we get a low frequency line (with the first line of the $\text{LL}_i$ and $\text{HL}_i$ subbands) and a high frequency line (with one line from $\text{LH}_i$ and $\text{HH}_i$). The lines that have been handled in this step are shown in the highlighted area on the left of Fig. 5. Now, in this figure, each square represents an entire line instead of a single sample. The squares at the top of Fig. 5

*3.1)* **if** $buffer_i$ is empty

**for** $j = W \ldots 0$

$buffer_i(j) = 1\text{DFWT}(\text{GetLLlineBwd}(i\text{-}1))$

Successively predict and update the lines in $buffer_i$

provided that the required lines are in *buffer*.

*3.2)* **if not**( more_lines( $i$-1) )

**repeat** twice Shift( $buffer_i$ )

Update and predict the remaining lines in the buffer

*3.3)* **else**

**repeat** twice

Shift( $buffer_i$ )

$buffer_i(0) = 1\text{DFWT}(\text{GetLLlineBwd}(i\text{-}1))$

$buffer_i(1) = (buffer_i(0) + buffer_i(2))\,p_1 + buffer_i(1)$

$buffer_i(2) = (buffer_i(1) + buffer_i(3))u_1 + buffer_i(2)$

$\ldots$

$buffer_i(W) = (buffer_i(W-1) + buffer_i(W+1))u_{W/2} + buffer_i(W)$

For *3.1)*, *3.2)* and *3.3)*

$\{LLline, HLline\} = buffer_i(W) * (1/K)$

$\{LHline, HHline\} = buffer_i(W-1) * K$

Fig. 6. Algorithm 4: Lifting implementation, recursive case.

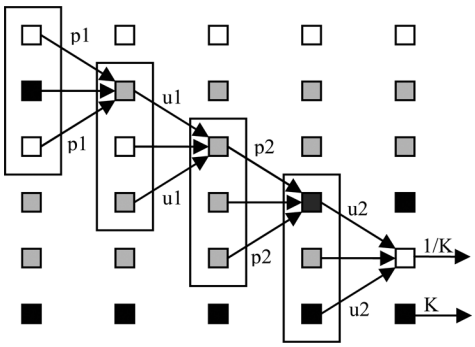


Fig. 7. Line processing in buffer for the lifting scheme.

are lines from the previous level, the grey squares in the middle of the figure are intermediate states of those lines, while those at the bottom are the concatenation of $\text{LH}_i$ and $\text{HH}_i$ lines (black squares), or $\text{LL}_i$ and $\text{HL}_i$ lines (white squares).

At this moment, we can see in Fig. 5 that all the lines in a diagonal (on the hypotenuse of the triangle that defines the left area) are predicted or updated (except the first one). Hence, if we introduce two more lines, and discard other two lines (the line in $W$ was processed and encoded, and $W+1$ was empty), we can compute two more lines in a segmented way (case 3.3). Every time that we introduce two lines in the buffer, the lines are processed as described in Fig. 7 (for B9/7). The first column in this figure indicates the initial state, in which we have two new lines (white and black squares) at top. Then, the new odd line is predicted from its two contiguous even lines (and this square becomes grey because it is an intermediate value). Afterwards, we update the third line in the buffer from the contiguous even lines, and so forth. At the end of this process, we have computed two new lines, which represent four subband lines. The new

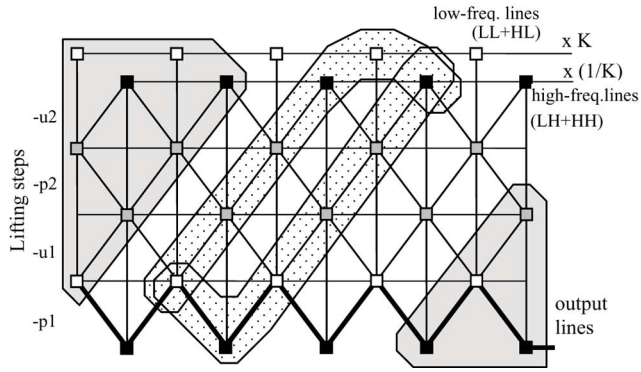


Fig. 8. Overview of the lifting scheme for the B7/9 IWT.

high-frequency line is not released because it is necessary for the following pass. Thus, we normalize and release the new low-frequency line, and the high-frequency line that was computed in the previous pass with a total of four subband lines (two per square). This computation of four subband lines is represented in the middle of Fig. 5 as the pass from the left dotted area, which represents the initial state in Fig. 7 (first column), to the right dotted area, which is the final state in Fig. 7 (last column).

Finally, the highlighted area on the right of Fig. 5 shows the lines that are processed when no more lines can be read from the previous level $i-1$ (case 3.2). In this case, we use the remaining intermediate lines to generate more subband lines while we are cleaning the buffer by shifting it two positions in each call.

Although symmetric extension is not as necessary in the lifting scheme as in a filter bank implementation (to make the transform non-expansive), a simple way to apply symmetric extension in this algorithm is to double the weighting factor when there is only a line available to predict/update another line, in other words, if the predicted/updated line is on an edge in Fig. 5.

For the inverse transform, we have to perform the same operations as in the forward DWT but in the reverse order. Moreover, the sign of the weighting factors have to be changed, and the scaling factors are swapped. These modifications are shown in Fig. 8, where the input data are compound subband lines ($\text{LL}_i+\text{HL}_i$ and $\text{LH}_i+\text{HH}_i$ interleaved), and the output data are low-frequency lines of the previous level $(\text{LL}_{i-1})$. The IWT algorithm is similar to the one given for the forward implementation (Algorithm 4), but considering the same changes that were described in the filter bank implementation. Recall that two lines are computed in every step, and thus the $updatebuffer_i$ variable is used to know if there is a line available in the buffer from the previous pass, or the buffer needs updating to compute two more lines.

### *D. Some Theoretical Considerations*

The main advantage of line-based algorithms is their lower memory requirements compared with the regular wavelet transform. In the filter bank implementation, every buffer contains $2N+1$ lines, so it needs to store $(2N+1) \times BufferWidth$ coefficients. If the image width is $w$ and height is $h$, the first-level buffer width is $w$ coefficients, and the width is halved at every level. So the memory requirements for all the buffers are $(2N+$

$1) \times w + (2N+1) \times w/2 + \cdots + (2N+1) \times w/2^{nlevel}$ coefficients, which is asymptotically (as *nlevel* approaches infinity) $2\times(2N+1)\times w$ coefficients. This is even better when the lifting scheme is used, because only $2 \times (W+2) \times w$ coefficients are needed (the relationship between the $2N+1$ and $W$ is shown in [26], and asymptotically $2N+1$ is twice $W$). We can compare these memory requirements with the regular wavelet transform, which requires $height \times width$ coefficients. Since for efficient filter banks $2 \times (W+2) < 2 \times (2N+1) \ll height$ (i.e., twice the buffer height is considerably lower than the image height), a line-based approach uses much less memory than the regular one. Since the buffer size depends on the image width, and the required amount of memory in a line-based approach is independent of the image height, the best case for this algorithm is transforming wide images with few lines (if this is not the case, the image can be rotated if convenient and possible). Note that this analysis is exactly the same, independently of the use of an iterative or recursive line-based algorithm.

If we analyze the algorithm complexity as the number of coefficients that need to be computed, obviously the result is the same in the regular algorithm and in the line-based approaches. Thus, for the first decomposition level, $h \times w$ coefficients are computed in both cases (although in a different order in line-based algorithms), then for a second level, $(1/2) \times (1/2) \times (h \times w) = (h \times w)/4$ coefficients are computed, and in general for a $n$-level $(h \times w)/4^{n1}$ coefficients are calculated. For both the regular and the proposed algorithms, the total amount of coefficients to be computed can be expressed as $\sum_{i=1}^{nlevel}(h \times w/4^{i-1})$, which asymptotically is $\Theta((4/3)h \times w)$. However, reduction of memory has another beneficial side effect in execution time when our algorithm is implemented in a cache-based system. The subband buffers are more likely to fit in cache memory than the whole image, and that is why the execution time is substantially reduced. Moreover, we have replaced the column access of the regular DWT, which clearly affects the cache performance, with a more sophisticated access arranged in line buffers. In addition, we expect that the lifting scheme version will reduce the computational cost of the algorithm since it performs fewer floating-point operations.

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 [14]. Other simpler solutions are: 1) to save every level in secondary storage separately so that it can be read in a different order, or 2) if the wavelet transform is used for image compression, to keep the compressed coefficients in memory. For the sake of simplicity, we will use the last option for the coder that is introduced in Section III, although any of them can be used.

Energy consumption is critical when running the proposed algorithms in mobile devices (such as PDAs) in order to increase the operation time. In [19], the authors propose a general energy model for studying different 2-D DWT architectures. This model is completely based on memory access operations, since in this kind of memory data-intensive algorithms, such as the 2-D DWT, the energy dissipation due to data storage and transfers forms the dominant component (up to 80%) of the total power budget. It is worth of mention that a transfer to/from an on-chip memory consumes 4–10 times more power than one addition, while an off-chip access requires 10–100 times more power than an on-chip access. So, they propose the following expressions:

$$
\begin{aligned}
E_{\mathrm{MEM}} &= \sum_i E_{\mathrm{ON\_CHIP}} + \sum_i E_{\mathrm{OFF\_CHIP}} \\
E_{\mathrm{ON\_CHIP}} &= N\_\mathrm{ACCESSES}_{\mathrm{ON\_CHIP}} \\
&\quad \cdot F\left(N\_\mathrm{words}, N\_\mathrm{bits}, N\_\mathrm{ports}\right) \\
E_{\mathrm{OFF\_CHIP}} &= N\_\mathrm{ACCESSES}_{\mathrm{OFF\_CHIP}} \\
&\quad \cdot \Big[ N\_\mathrm{bits} \cdot N\_\mathrm{ports} \cdot 10^{-11} \cdot V_{\mathrm{DD}}^2 \\
&\quad + F\left(N\_\mathrm{words}, N\_\mathrm{bits}, N\_\mathrm{ports}\right) \Big]
\end{aligned}
$$

where $F$ is a function that depends on technology (memory size in words, bits per memory word and number of memory ports, among other parameters) and $V_{\mathrm{DD}}$ is the supply voltage. Although the number of memory access between the regular 2-D DWT transform and the line-based versions are similar, the on-chip memory access ratio of line-based proposal will be significantly high compared with the traditional 2-D DWT (notice that first-level cache is placed on-chip, as it happens in most current conventional processors). So, taking into account the previous considerations, we could say that line-based algorithms will require less energy than a regular convolution 2-D DWT transform.

## III. Run-Length Coding of the Wavelet Coefficients

In the proposed wavelet transform, once a subband line is calculated, it has to be encoded as soon as possible in order to release memory and reduce memory consumption. However, entropy coders need to exploit local similarity in the image to be efficient, and therefore better compression performance can be achieved if we group subband lines in an encoder buffer. These buffers store the lines released by the DWT and group them before the coding stage. When we consider that there are enough lines in a buffer to perform an efficient compression, the Run-Length algorithm (Algorithm 5, Fig. 9) is called, passing the encoder buffer (*Buffer*) and the level of the buffer $(L)$ as parameters.

Since the encoder does not know the whole image, but only the lines that are in the buffers at that moment, it cannot use global image information. Moreover, we aim at fast execution, and hence no R/D optimization or bit-plane coding can be made. In Section III-A, we propose a RLW encoder that fulfills the aforementioned features.

### A. Fast Run-Length Coding

In the proposed algorithm, which is formally described in Algorithm 5, the quantization process is performed by two strategies: one coarser and another finer. The finer one consists in applying a scalar uniform quantization to every

**function** RLW_Code_Subband( *Buffer*, *L* )
Scan *Buffer* in vertical order (i.e., in columns)
**for each** $c_{i,j}$ in *Buffer*
$nbits_{i,j} = \lceil \log_2(|c_{i,j}|) \rceil$
**if** $nbits_{i,j} \le rplanes$
**increase** $run\_length_L$
**else**
**if** $run\_length_L < enter\_run\_mode$
**repeat** $run\_length_L$ **times**
**arithmetic_output** *LOWER*
**else**
**arithmetic_output** *RUN*
$rbits = \lceil \log_2(run\_length_L) \rceil$
**arithmetic_output** *rbits*
**output** $\text{bit}_{rbits-1}(run\_length_L) \ldots \text{bit}_1(run\_length_L)$
$run\_length_L = 0$
**arithmetic_output** $nbits_{i,j}$
**output** $\text{bit}_{nbits_{(i,j)}-1}(|c_{i,j}|) \ldots \text{bit}_{rplane+1}(|c_{i,j}|)$
**output** $\text{sign}(c_{i,j})$
**end of function**
*Note*: $\text{bit}_n(c)$ is a function that returns the $n^{\text{th}}$ bit of $c$

Fig. 9. Algorithm 5: Run-length coding of the wavelet coefficients.

coefficient using a quantization parameter. We can use the same quantization parameter for all the wavelet coefficients for a $(\sqrt{2}, \sqrt{2})$ normalization, since the DWT then results in a nearly orthonormal transformation. Actually, we can avoid the multiplication that this quantization implies because it can be combined with the normalization step in the lifting scheme computation. The coarser quantization is based on removing bit planes from the least significant part of the coefficients. We define *rplanes* as the number of less significant bits to be removed, and we call significant coefficient those coefficients $c_{i,j}$ that are different from zero after discarding the *rplanes* bits, in other words, if $c_{i,j} \le 2^{rplanes}$.

The wavelet coefficients are encoded as follows. The coefficients in the buffer are scanned column by column (to exploit their locality). For each coefficient, if it is not significant, a run-length count of insignificant symbols at this level is increased $(run\_length_L)$. However, if it is significant, the run is broken and we have to encode the count of insignificant symbols and the significant coefficient.

A significant coefficient is represented by a symbol that indicates the number of bits required to represent that coefficient. We use arithmetic coding to store that symbol in an efficient way. Since coefficients in the same subband have similar magnitude, an adaptive arithmetic encoder is able to represent this information very efficiently. In addition, we can improve the compression performance of the arithmetic coding using two contexts depending on the significance of the left and upper neighbors (which have already been coded). Once we have encoded the amount of bits required by a significant coefficient, we still need to store its significant bits and the sign. For these bits, we use raw coding (i.e., with no compression) to speed up the execution time of the algorithm, with only a small loss in performance.

The count of insignificant symbols $(run\_length_L)$ is encoded with a special symbol, which is called *RUN* symbol. After encoding a *RUN* symbol, the run-length count is stored in a similar way as the significant coefficients. First, the number of bits needed to represent the run value is encoded (with adaptive arithmetic coding and a different context), afterwards, the bits are raw encoded.

Compression performance can be increased if a specific symbol is used for each insignificant coefficient, since an arithmetic encoder stores more efficiently many likely symbols than a lower amount of less likely symbols (i.e., different run-length counts). Notice that we do not use run-length coding to improve the compression performance but to reduce the complexity of the algorithm, most of all at high compression ratios, where many insignificant coefficients in a run are coded with only a symbol. Nevertheless, for short run-lengths, we improve the coding performance if a *LOWER* symbol is encoded for each insignificant coefficient instead of a RUN symbol for the entire sequence. The threshold value to enter in run-length mode and start using run-length symbols is defined by the *enter_run_mode* parameter in Algorithm 5.

### B. Tradeoff Between R/D Performance and Speed and Memory Requirements

The proposed algorithm can be tuned depending on the final application requirements. Thus, some parameters can be adjusted to improve the compression performance at the cost of slightly higher memory requirements or execution time. This way, the number of lines in every encoder buffer can be 8 for a good R/D performance, but compression efficiency can be improved with 16 lines, increasing the memory requirements. Another parameter that can be tuned is the *enter_run_mode* variable of Algorithm 5. When this parameter is increased, larger run-lengths are encoded by successive *LOWER* symbols, which results slower but a bit more efficient in R/D performance. Another tradeoff between compression efficiency and complexity is the use of arithmetic coding (with contexts) instead of raw coding to encode the sign of the coefficients, because there is a dependence among the sign of the wavelet coefficients inside a subband. In general, each of these improvements may increase the PSNR of an image encoded at 1 bpp in about 0.1 dB, while the two latter improvements increase the execution time in about 20% each one.

## IV. Global Experimental Results and Coding Performance

In order to compare the regular wavelet transform and our proposals, we have implemented them (using the well-known Daubechies 9/7 [25], [26] ) with standard ANSI C language, on a regular PC computer with 256 KB L2 cache. These implementations are available at [21].

For these tests, we have used the standard Lena ($512 \times 512$) and Woman ($2048 \times 2560$) images. With six decomposition levels, the regular WT (both filter bank and lifting implementations) needs 1030 KB for Lena and 20510 KB for Woman, while the filter bank implementation requires 41 KB for Lena and 162 KB for Woman, i.e., it uses 25 and 127 times less memory. Our proposal using the lifting scheme still needs less memory, requiring 26 KB for Lena and 103 KB for Woman,

TABLE II
MEMORY REQUIREMENT (KB) COMPARISON AMONG OUR PROPOSALS AND THE USUAL ALGORITHM USING DIFFERENT IMAGE SIZES

| Image size (megapixel) | Regular WT | Proposed convolution | Proposed lifting |
|---|---|---|---|
| 20 (4096 x 5120) | 81,980 | 324 | 205 |
| 16 (3712 x 4480) | 65,013 | 293 | 186 |
| 12 (3200 x 3968) | 49,647 | 253 | 160 |
| 8 (2560 x 3328) | 33,319 | 202 | 128 |
| 5 (2048 x 2560) | 20,510 | 162 | 103 |
| 4 (1856 x 2240) | 16,266 | 147 | 93 |
| 3 (1600 x 1984) | 12,423 | 127 | 80 |
| 2 (1280 x 1664) | 8,340 | 101 | 64 |
| 1.25 (1024 x 1280) | 5,125 | 81 | 51 |
| VGA (512 x 640) | 1,288 | 41 | 26 |

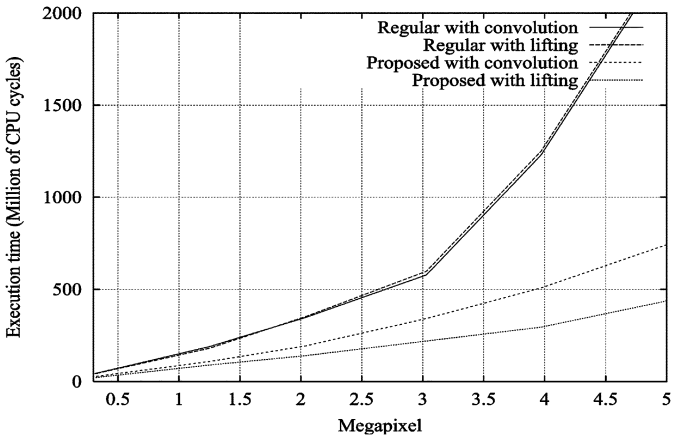


Fig. 10. Execution time comparison (excluding I/O time).

which means that it only requires 60% of memory with respect to the filter bank algorithm. In addition, Table II shows that our proposals are much more scalable than the usual DWT.

In Fig. 10, we present an execution time comparison between our proposals and the regular DWT (convolution and lifting). It shows that, while our algorithms display linear behavior, the regular DWTs approach to an exponential curve. This behavior is mainly due to the ability of our algorithms to fit in cache for all the image sizes (162 KB and 103 KB for the 5-megapixel image with convolution and the lifting scheme, respectively). On the contrary, the usual wavelet transform rapidly exceeds the cache limits (e.g., it needs 1287 KB for the VGA resolution). In the regular implementation, an interesting point is that, despite requiring fewer operations, the lifting scheme exhibits almost the same execution time as the filter bank implementation, since memory access is the real bottleneck in the regular implementation. Thus, the relieved computing time is spent in the coefficient reorganization required by the lifting scheme. However, in the new proposal, this figure shows that the lifting scheme implementation is about 40% faster than the convolution algorithm, not only because fewer floating-point operations are performed, but mainly due to the reduced memory access. Anyway, both proposals are faster than the usual wavelet transform, due to their better use of the cache memory.

Since the forward and inverse transform are symmetric, further experiments have shown that the IWT has similar results

TABLE III
PSNR (DB) WITH DIFFERENT BIT RATES AND CODERS FOR THE EVALUATED IMAGES. THE NUMBERS IN PARENTHESIS CORRESPOND TO THE DECREASE OF PERFORMANCE IF THE R/D IMPROVEMENTS DISCUSSED IN SECTION III-B ARE NOT APPLIED

| | Lena (512x512) | | | Barbara (512x512) | | |
|---|---|---|---|---|---|---|
| codec\ rate | SPIHT | Jasper/ JP2K | Proposed run length | SPIHT | Jasper/ JP2K | Proposed run length |
| 1 | 40.41 | 40.31 | 40.37 (-0.14) | 36.41 | 37.11 | 36.82 (-0.35) |
| 0.5 | 37.21 | 37.22 | 37.15 (-0.10) | 31.39 | 32.14 | 31.90 (-0.29) |
| 0.25 | 34.11 | 34.04 | 34.03 (-0.08) | 27.58 | 28.34 | 28.12 (-0.22) |
| 0.125 | 31.10 | 30.84 | 30.97 (-0.04) | 24.86 | 25.25 | 25.19 (-0.08) |
| | Woman (2560x2048) | | | Café (2560x2048) | | |
| codec\ rate | SPIHT | Jasper/ JP2K | Proposed run length | SPIHT | Jasper/ JP2K | Proposed run length |
| 1 | 38.28 | 38.43 | 38.49 (-0.21) | 31.74 | 32.04 | 31.89 (-0.26) |
| 0.5 | 33.59 | 33.63 | 33.72 (-0.15) | 26.49 | 26.80 | 26.67 (-0.16) |
| 0.25 | 29.95 | 29.98 | 30.04 (-0.08) | 23.03 | 23.12 | 23.10 (-0.12) |
| 0.125 | 27.33 | 27.33 | 27.40 (-0.04) | 20.67 | 20.74 | 20.67 (-0.06) |

in memory requirement and execution time. Moreover, a comparison with iterative methods (like the one in [14]) shows very similar execution times and memory requirements, since in both implementations the number of floating-point operations and memory access is identical and executed in the same order, and the defined buffers and their size are exactly the same as well.

Next, we compare the proposed RLW coder with the state-of-the-art wavelet coders SPIHT [13] and JPEG 2000 [1]. The results for JPEG 2000 have been obtained using Jasper [27], an official implementation included in the ISO/IEC 15444-5 standard. All of them use Daubechies B9/7 and have been written and compiled with the same level of optimization. In our comparison, we use the standard images Lena and Barbara (monochrome, 8 bpp, $512 \times 512$) and the larger and less blurred images Café and Woman (monochrome, 8 bpp, $2560 \times 2048$, equiv. 5 megapixel), from the JPEG 2000 testbed.

Table III shows a compression comparison for the evaluated images and coders. In general, our proposal performs as well as SPIHT for less detailed images (Lena and Woman) and better than it for more detailed images (Barbara and Café). SPIHT performs worse with complex images because it is based on coefficients trees, and it can establish fewer trees in images with many details. On the contrary, JPEG 2000 is more efficient than our proposal in highly detailed images, since it defines more contexts and uses a Rate/Distortion optimization algorithm. Both JPEG 2000 and our run-length coder exhibit similar results in low detailed images.

As expected, the comparison in which our encoder clearly outperforms both SPIHT and the evaluated implementation of JPEG 2000 (Jasper) is in memory consumption. Table IV shows that for a 5-megapixel image, our proposal requires between 25 and 35 times less memory than SPIHT, and more than 50 times less memory than Jasper/JPEG 2000. However, it is important to point out that other implementations of JPEG 2000 can integrate a line-based approach (an iterative version or the new recursive proposal) to greatly reduce memory consumption. In this table, the last column refers to the case in which the complete bit stream (i.e., the compressed image) is kept in memory while it is generated. Due to the computation order in the proposed wavelet

TABLE IV
TOTAL AMOUNT OF MEMORY REQUIRED (IN KB) TO ENCODE THE WOMAN IMAGE WITH THE COMPARED ALGORITHMS. THE NUMBERS IN PARENTHESIS CORRESPOND TO THE MEMORY THAT IS SAVED IF THE R/D IMPROVEMENTS ARE NOT USED (IT IS APPLIED IN BOTH COLUMNS OF OUR PROPOSED ALGORITHM)

| code\ rate | Compressed Image File | SPIHT | Jasper/ JP2K | Proposed run length | Proposed with bit- stream in memory | |
|---|---|---|---|---|---|---|
| 1 | 640 | 42,888 | 62,768 | 1,197 | 1,837 | (-180) |
| 0.5 | 320 | 35,700 | 62,240 | 1,133 | 1,453 | (-180) |
| 0.25 | 160 | 31,732 | 61,964 | 1,133 | 1,293 | (-180) |
| 0.125 | 80 | 28,880 | 61,964 | 1,117 | 1,197 | (-180) |

TABLE V
EXECUTION TIME (IN MILLION OF CPU CYCLES) NEEDED TO ENCODE LENA AND WOMAN WITH DIFFERENT IMAGE SIZE. THE NUMBERS IN PARENTHESES CORRESPOND TO TIME REDUCTION ACHIEVED IF NO R/D IMPROVEMENTS ARE APPLIED

| | Woman (2560x2048) | | | Lena (512x512) | | |
|---|---|---|---|---|---|---|
| | SPHIT | Jasper / JP2K | Proposed run length | SPHIT | Jasper / JP2K | Proposed run length |
| 1 | 3,669 | 23,974 | 1,801 (-587) | 147 | 750 | 96 (-28) |
| 0.5 | 2,470 | 23,864 | 1,237 (-377) | 97 | 734 | 63 (-21) |
| 0.25 | 1,939 | 23,616 | 916 (-259) | 73 | 726 | 42 (-11) |
| 0.125 | 1,651 | 23,563 | 729 (-197) | 60 | 717 | 32 (-7) |

transform, the coefficients from different subband levels are interleaved. Thus, instead of a single bit stream, we can generate a different bit stream for every level. These different streams can be kept in memory or saved in secondary storage. The amount of memory required for the bit stream buffers can be reduced if we reverse the order of the coefficients and release memory as soon as we have enough coefficients at all the levels (e.g., one line at level N, two lines at level N-1, four lines at N-2, etc.). In this table, we can estimate that the amount of memory needed for a single process (written in C and running under Windows XP) is about 650 KB, and the remaining memory is required for the DWT transform and the coding algorithm. Moreover, we can save 180 KB if we use 8 lines per buffer instead of 16.

Finally, we compared the complexity of the coders. Since JPEG 2000 has more contexts and uses a Rate/Distortion optimization algorithm, it is more complex than our proposal. SPIHT is also more complex because it performs several image scans focusing on a different bit-plane in every image scan. In addition, in cache-based systems, the proposed DWT improves the cache performance. Table V presents an execution time comparison with two image sizes (Lena and Woman). It shows that our algorithm clearly outperforms Jasper/JPEG 2000, and it is several times faster than SPIHT. Moreover, we can speed it up in about 30% if no compression improvement is performed (in particular, without arithmetic coding for the signs, and with a lower value for the *enter_run_mode* parameter in Algorithm 5).

## V. CONCLUSION

In this paper, we have introduced a recursive line-by-line wavelet transform algorithm, which presents very low memory requirements and reduced execution time. The execution order of the wavelet transform is automatically placed by recursion and this way, the problems about different delay and rhythm among buffers are explicitly solved. A first general description has been further detailed to allow two different implementations (filter bank and lifting algorithms), being the former simpler to implement but the latter more efficient in terms of complexity and memory requirements.

The proposed DWT can be used as part of a compression algorithm such as JPEG 2000, speeding up its execution time and reducing its memory requirements. However, we have described a simple run-length coder to be used along with this recursive wavelet transform, which displays good compression performance and very low complexity requirements.

As a conclusion, the main contribution of the proposed DWT and image coder is their low memory requirements with a straightforward software implementation, which makes them good candidates for many embedded systems and other memory-constrained applications (such as digital cameras and PDAs).

## ACKNOWLEDGMENT

The authors would like to thank E. Oliver for her implementation of the lifting scheme.

## REFERENCES

[1] *JPEG2000 Image Coding System*, ISO/IEC 15444-1, 2000.
[2] S. Mallat, "A theory for multiresolution signal decomposition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 11, no. 7, pp. 674–693, Jul. 1989.
[3] K. R. Rao and P. Yip, *Discrete Cosine Transform, Algorithms, Advantages, Applications*. New York: Academic, 1990.
[4] E. Murata, M. Ikekawa, and I. Kuroda, "Fast 2-D IDCT implementation with multimedia instructions for a software MPEG2 decoder," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process.*, 1998, pp. 3105–3108.
[5] K. Lengwehasatit and A. Ortega, "Scalable variable complexity approximate forward DCT," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 14, no. 11, pp. 1236–1248, Nov. 2004.
[6] G. Uytterhoeven, "Wavelets: software and applications," Ph.D. dissertation, Dept. Computerwetenschappen, Katholieke Universiteit Leuven, Leuven, Belgium, 1999.
[7] W. Sweldens, "The lifting scheme: A custom-design construction of biorthogonal wavelets," *Appl. Comput. Harmon. Anal.*, vol. 3, pp. 186–200, 1996.
[8] W. Sweldens, "The lifting scheme: a construction of second generation wavelets," *SIAM J. Math. Anal.*, pp. 511–546, 1997.
[9] M. Rabbani and R. Joshi, "An overview of the JPEG2000 still image compression standard," *Signal Process.: Image Commun.*, vol. 17, pp. 3–48, Jan. 2002.
[10] M. Vishwanath, "The recursive pyramid algorithm for the discrete wavelet transform," *IEEE Trans. Signal Process.*, vol. 42, no. 3, pp. 673–676, Mar. 1994.
[11] P. Cosman and K. Zeger, "Memory constrained wavelet based image coding," *IEEE Signal Process. Lett.*, vol. 5, no. 9, pp. 221–223, Sep. 1998.
[12] J. M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," *IEEE Trans. Signal Process.*, vol. 41, no. 12, pp. 3445–3462, Dec. 1993.
[13] A. Said and A. Pearlman, "A new, fast, and efficient image codec based on set partitioning in hierarchical trees," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 6, no. 6, pp. 243–250, Jun. 1996.
[14] C. Chrysafis and A. Ortega, "Line-based, reduced memory, wavelet image compression," *IEEE Trans. Image Process.*, vol. 9, no. 3, pp. 378–389, Mar. 2000.
[15] Y. Bao and C. C. J. Kuo, "Design of wavelet-based image codec in memory-constrained environment," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 11, no. 5, pp. 642–650, May 2001.
[16] C. K. Hu, W. M. Yan, and K. L. Chung, "Efficient cache-based spatial combinative lifting algorithm for wavelet transform," *J. Signal Process.*, vol. 84, no. 5, pp. 1689–1699, May 2004.

[17] T. Acharya and P. Tsai, *JPEG 2000 Standard for Image Compression: Concepts, Algorithms and VLSI Architectures*. New York: Wiley, 2005, ch. 5.
[18] G. Dillen, B. Georis, J. Legat, and O. Cantineau, "Combined line-based architecture for the 5-3 and 9–7 wavelet transform of JPEG 2000," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 9, pp. 944–950, Sep. 2003.
[19] N. Zervas, G. Anagnostopoulos, V. Spiliotopoulos, Y. Andreopoulos, and C. Goutis, "Evaluation of design alternatives for the 2-D discrete wavelet transform," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 11, no. 12, pp. 1246–1262, Dec. 2001.
[20] W.-H. Chang, Y.-S. Lee, W.-S. Peng, and C.-Y. Lee, "A line-based, memory efficient and programmable architecture for 2-D DWT using lifting scheme," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, 2001, vol. 4, pp. 330–333.
[21] C Implementation of the Proposed Algorithms. [Online]. Available: http://www.disca.upv.es/joliver/wavelet
[22] D. Taubman, "High performance scalable image compression with EBCOT," *IEEE Trans. Image Process.*, vol. 9, no. 7, pp. 1158–1170, Jul. 2000.
[23] M. J. Tsai, J. Villasenor, and F. Chen, "Stack-run image coding," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 6, no. 10, pp. 519–521, Oct. 1996.
[24] W. Berghorn, T. Boskamp, M. Lang, and H. Peitgen, "Fast variable run-length coding for embedded progressive wavelet-based image compression," *IEEE Trans. Image Process.*, vol. 10, no. 12, pp. 1781–1790, Dec. 2001.
[25] M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies, "Image coding using wavelet transform," *IEEE Trans. Image Process.*, vol. 1, no. 2, pp. 205–220, Apr. 1992.
[26] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," *J. Fourier Anal. Appl.*, vol. 4, no. 3, pp. 247–269, 1998.
[27] M. Adams, "Jasper Software Reference Manual (Version 1.600.0)," ISO/IEC JTC 1/SC 29/WG 1 N 2415, 2002.
[28] K. K. Parhi and T. Nishitani, "VLSI architectures for discrete wavelet transforms," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 1, no. 6, pp. 191–202, Jun. 1993.

**Jose Oliver** (M'04) received the M.S. and Ph.D. degrees in computer engineering from the Technical University of Valencia (Universidad Politécnica de Valencia, UPV), Spain, in 1998 and 2006, respectively.

He has been with the Department of Computer Engineering (DISCA) at UPV since 1999, where he holds a Lecturer position gained by means of the "programa cantera" in 2001. Since then, he has taught Computer Networks and Multimedia Networks at the Faculty of Computer Science, and he has served as a referee for several major conferences and journals. His research interests include image and video coding and transmission, and digital signal processing.

**Manuel Perez Malumbres** (M'01) received the B.S. degree in computer science from the University of Oviedo, Spain, in 1986. In 1989, he joined the Computer Engineering Department (DISCA) at the Technical University of Valencia (UPV), Spain, as an Assistant Professor. He received the M.S. and Ph.D. degrees in computer science from UPV in 1991 and 1996, respectively.

In 2000, he became an Associate Professor and founded the Computer Networks Group (GRC), being the group director until June 2005. In September 2005, he moved to Miguel Hernandez University, Elche, Spain, holding the same position. He is the author of more than 90 conference and journal publications and several networking books for undergraduate CS courses. Currently, his research and teaching activities are related to multimedia networking (audio/video coding and network delivery) and wireless technologies (MANETs and sensor networks).

Dr. Perez Malumbres is a technical committee member of the IEEE Multimedia Communications Group (since 2004) and also a member of IASTED Image Processing technical committee (since 2002). He serves as a technical program committee member of several international conferences and journals related with his main research interests.