Fast tree-based wavelet image coding with efficient use of memory

Jose Oliver*, Manuel P. Malumbres

Department of Computer Engineering (DISCA),
Universidad Politécnica de Valencia
Camino de Vera 17, 46022 Valencia, Spain

ABSTRACT

In this paper, we present a new wavelet image encoder that focuses on overcoming the main drawbacks of other wavelet based-image coders; their high memory requirements and complexity. In our proposal, we tackle these problems in both parts of the wavelet coder, both the DWT algorithm and the coding system. For the DWT, we propose the use of a line-based algorithm, in which we get rid of the wavelet coefficients as soon as they have been calculated. Some previous line-based proposals cannot be easily implemented, and here we describe a recursive algorithm that is implemented more easily. For the coding system, we use a tree-based coder. These coders have evidenced good Rate/Distortion (R/D) performance. However, it has not been clearly shown their excellent possibilities for fast processing of coefficients. Previous tree-based proposals need the entire image in memory to work and so they cannot be used in our system. Other wavelet encoders achieve good R/D performance, but exhibit high complexity. Our proposal is simpler and therefore faster. Numerical results show that the compression performance of our tree-based encoder is equal or better than state-of-the-art coders, such as SPIHT and JPEG 2000, except for highly detailed images. However, the real benefits of our proposal is shown in the amount of memory required, which is reduced drastically (in the order of 25 times less memory than SPIHT and more than 35 times less than JPEG 2000), and in its lower execution time (about four times lower than SPIHT, and more than 20 times lower than JPEG 2000). These results show that our encoder is a good candidate for many embedded systems and other memory-constrained environments, such as digital cameras and PDAs.

Keywords: Lossy image coding, wavelet coding, tree-based coding, efficient memory usage, fast processing.

1. INTRODUCTION

Wavelet-based image encoders have aroused great interest in the last years due to their nice features, such as natural multiresolution and high compactness of the coefficients, which leads to high compression efficiency. Its compression efficiency caused the new standard JPEG 2000 [6] to use the DWT as its native transform. However, one of the main drawbacks of current wavelet encoders is their high complexity and high memory usage, most of all compared with DCT-based encoders, such as the original JPEG [5] standard.

The main reason why wavelet-based encoders are memory-intensive is that the regular wavelet transform requires a lot of memory to be computed. In addition, in many wavelet encoders, such as [10] [6], the subsequent coding process uses some extra lists or complex data structures in order to hold the state of the coding process. Another usual problem of wavelet-based encoders is their higher complexity. The wavelet coefficients are not encoded using simple run-length coding with Huffman coding (or Variable Length Coding) like in the baseline mode of the standard JPEG, but more complex bit-plane coding with several contexts, adaptive arithmetic coding, and rate-distortion optimization algorithms are usually applied.

One of the first efficient wavelet image coders reported in the literature is the EZW [11]. It is based on the construction of coefficient-trees and successive-approximations, which can be implemented as bit-plane processing. Due to its successive-approximation nature, it is SNR scalable, although at the expensive of sacrificing spatial scalability. SPIHT [10] is an advanced version of this algorithm, where coefficient-trees are processed in a more efficient way. Both

* {joliver, mperez}@disca.upv.es

algorithms need computation of coefficient-trees and perform different iterations focusing on a different bit plane per iteration, which involves high computational complexity. In [14], a tree-based wavelet encoder is introduced which uses a rate/distortion optimization algorithm based on the combination of space and frequency quantization (SFQ). Space quantization is applied by means of a tree pruning algorithm, which modifies the shape of the trees by pruning theirs branches, while a scalar quantization is used for frequency quantization. On the one hand, a joint optimization of both quantization parameters achieves higher rate/distortion performance than SPIHT, but on the other hand, due to the iterative tree pruning stage, the SFQ encoder is about five times slower than other tree-based algorithms like SPIHT.

In the JPEG 2000 standard [6], the proposed EBCOT algorithm [17] does not use coefficient-trees, but it performs bit-plane processing with three passes per bit plane. It also uses a new R/D optimization algorithm along with large number of contexts in order to overcome the inconvenience of not using coefficient-trees. JPEG 2000 attains both spatial and SNR scalability by means of post-processing the encoded image, hence the final bit stream is very versatile at the cost of higher computational complexity. However, the best R/D performance in wavelet coding is not achieved by JPEG 2000 but rather by high-order context modeling techniques [16] [15]. In this kind of coder, not only contiguous coefficients are taken into account for context formation but also other coefficients (including some in other subbands) are evaluated. Despite their excellent compression performance, two main drawbacks arise from these coders. First, training is used to calculate the contexts for coding. Second, computation for high-order context modeling, in which these coders lie, is CPU time intensive; especially the texture pattern extraction from neighboring samples and the linear combination of them, which results in an increase of the encoder complexity.

In this paper we are going to introduce a fast tree-based wavelet encoder with low use of memory. Many proposals that have been done using tree-based wavelet encoders [11] [10] [14] have evidenced the good Rate/Distortion (R/D) performance that this strategy yields. However, it has not been clearly shown their excellent possibilities for fast processing of coefficients. Moreover, these tree-based encoders need the entire image in memory to work, since they apply the regular wavelet transform in Mallat decomposition [7] in first place, and perform the coding stage in second place. Therefore, they cannot be used in our system because we aim at low memory usage, and hence we need a wavelet encoder that is able to encode the wavelet coefficients as soon as they are calculated in a line-based fashion [2]. For this reason, we have to deal with both memory requirement and complexity in both stages of a wavelet-based encoder, i.e., in the wavelet transform computation and in the following stage in which the wavelet coefficients have to be efficiently encoded.

The reminder of this paper is arranged as follows. In Section 2, we will tackle the problem of how to reduce the memory requirements in the wavelet transform, with fast computation of the wavelet coefficients. In Section 3 we will introduce a simple wavelet encoder, in which every wavelet coefficient is encoded independently with an arithmetic encoder, and which can be applied along with the algorithm proposed in Section 2. This simple encoder will be used as a starting point for the proposed tree-based algorithm, which is further detailed in Section 4, and which is faster and more efficient than that presented in Section 3. Finally, in Section 5, some experimental results are given, so that our proposal can be compared with the state-of-the-art coders SPIHT and JPEG 2000 in terms of R/D performance, complexity and memory requirements.

2. A RECURSIVE WAVELET TRANSFORM ALGORITHM USING A LINE-BASED APPROACH

One of the desirable features of the proposed image coder is to have low memory consumption. As our proposal is a wavelet based coder, the first bottleneck that appears in the efficient use of memory is the computation of the DWT. Our encoder could only have low memory consumption if the DWT is performed in an efficient way.

In the regular DWT, Mallat decomposition is performed [7]. In this decomposition, an image is transformed first line by line, and then row by row, at every decomposition level. Therefore, it must be kept entirely in memory. In this section we propose a different wavelet transform that uses a line-based strategy, in which the key idea for saving memory is to get rid of the wavelet coefficients as soon as they have been calculated.

This idea was first used in [13], aiming to reduce the memory requirements of the 1D DWT. Later, one of the first approaches to reduce memory consumption in image processing was done in [4]. In this algorithm, in order to reduce the memory requirements, the encoder reorders the output bit-stream so that some wavelet coefficients from several subbands representing the same area are placed together, and this way, the decoder can compute only 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. This line-based algorithm was further refined in

[2], where reduction of memory is dealt in both the forward and inverse transform (in [4] it is done only in the decoder) and other issues related to the order of the data are solved so that it allows efficient use of memory in the encoder and the decoder. However, in this 2D version, the authors do not propose a direct algorithm to implement their proposal, and it cannot be easily implemented due to some unclear aspects. In this section, we present a simple recursive algorithm that can be implemented straightforwardly in any programming language, and which we will use in our image coder.

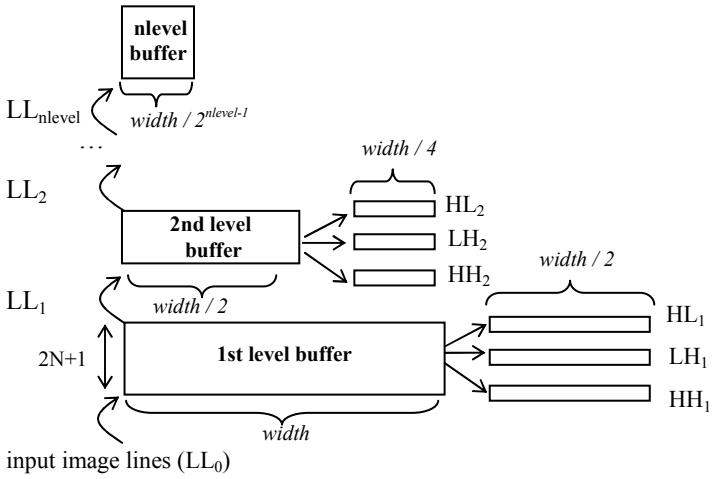

Figure 1. Overview of a line-based forward wavelet transform

2.1. A Line-Based Approach for Image Coding

In a line-based strategy, we only keep in memory those image lines that we are dealing with, while the wavelet coefficients that already have been calculated are output to the coding stage, and the rest of image lines are not input until they are strictly needed.

For the first decomposition level, the algorithm directly receives image lines, one by one. On every input line, a one-level 1D 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 buffer associated to the first decomposition level. This buffer must be able to keep 2N+1 lines, where 2N+1 is the number of taps for the largest analysis filter bank. We only consider odd filter lengths because they have higher compression efficiency, however this analysis could be extended to even filters as well.

When 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 low-pass filter and then the high-pass filter. The result of this operation is the first line of the HL_1, LH_1 and HH_1 wavelet subbands, and the first line of the LL_1 subband. At this moment, we can encode and release the first line of the wavelet subbands. However, the first line of the LL_1 subband does not need to be encoded, but it is needed as incoming data for the following decomposition level.

On the other hand, once the lines in the first level buffer have been used, this buffer is shifted twice (using a rotation operation) so that two lines are discarded and another two image lines are input at the other end. Once the buffer is updated, the process can be repeated and more lines are obtained.

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 in the very same way as we have described for the first level. In this manner, the lines of the second wavelet subbands are achieved, and the low-frequency lines from LL_2 are passed to the third level.

As it is depicted in Figure 1, where an overview of this line-based forward wavelet transform is shown, this process can be repeated until the desired decomposition level (*nlevel*) is reached. At this moment, we compute lines from the last LL subband (LL_{nlevel}), which is part of the final result in a typical wavelet dyadic decomposition.

Although this algorithm might seem quite simple, a major problem arises when it is implemented. This 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, according to their level. Handling several buffers with different delay and rhythm becomes a hard task. In the next subsection we describe a recursive algorithm that clearly specifies how to perform this communication between buffers.

2.2. A Recursive Wavelet Transform with Buffer Synchronization

In [9], we presented a general-purpose recursive algorithm that solves the problems addressed in [2]. Since it is efficient and has low memory requirements, we will use it in our image coder. In this subsection, we are going to describe this forward wavelet transform. The reader is referred to [9] for more details on this algorithm.

In order to solve the synchronization problems, our line-based algorithm defines a recursive function that obtains low-frequency subband (LL) lines at a specific level (determined by a function parameter), using the subband lines of the contiguous level. This recursive function is called GetLLlineBwd(*level*). The first time that this function is called at a certain level, it returns the first line of the LL subband at that level, the following time it returns the second line, etc. If there are no more lines at the specified level, it returns the EOL tag. While the n^{th} line of the LL_{level} subband is computed and returned, the corresponding n^{th} lines of the HL, LH and HH subbands at that level are also computed, compressed and released.

The whole wavelet transform is computed as follows. The wavelet transform starts requesting LL lines to the last level (*nlevel*) using the function call GetLLlineBwd(n*level*). As we have shown in Figure 1, the *nlevel* buffer must be filled with lines from the *nlevel*-1 level before it can generate lines. In order to get these lines, the function recursively call itself until the level 0 is reached. At this point, it no longer needs to call itself since it can return an image line that can be read directly from the input/output system. Notice that although we are calculating a forward wavelet transform, we do it by means of a backward recursion, since it goes from *nlevel* to 0.

Let us see how the GetLLlineBwd(*level*) function works with more detail. Before computing the wavelet lines, we have to prepare the buffer from which those lines are computed. The first time that this recursive function is called at every level, it has its buffer (called $buffer_{level}$) empty. Therefore, its upper half (from N to 2N) is recursively filled with lines from the previous level. Recall that once a line is received, it must be transformed using a 1D FWT before it is stored. Once the upper half is full, the lower half is filled using symmetric extension (the N+1 line is copied into the N-1 position, …, the 2N is copied into the 0 position). On the other hand, if the buffer is not empty, it simply has to be updated. In order to update it, it is shifted 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) using a recursive call. This operation is repeated twice. However, if there are no more lines in the previous level, this recursive call will return *End Of Line* (EOL). That points out that we are about to finish the computation at this level, but we still need to continue filling the buffer. We fill it using symmetric extension again.

Once the buffer is filled or updated, both high-pass and low-pass filter banks are applied to every column in the buffer. As result of the convolution, we get a line of every wavelet subband at this level, and a LL line. The wavelet coefficients are compressed using the coder that we will detail in the next section, and this function call returns the LL line.

Every recursive function needs at least one base case to stop backtracking. This function has two base cases. The first case is when all the lines at this level have been read. It can be detected by keeping an account of the number of lines read and the maximum number of lines that can be read at every level. In this case, the function returns EOL. Another way to detect this base case is propagating the EOL tag. The second base case is reached when the level parameter reaches 0 and then no further recursive call is need since an image line can be read directly from the image source (e.g., from an input file).

We can describe the recursive function that we have depicted as follows:

- *function* GetLLlineBwd (*level*)
- First base case: If the input parameter *level* is equal to 0, the function returns an image line, which is read from the I/O system.
- Second base case: If there are no more lines at this level, because all the lines have already been read, the function returns an EOL (End Of Line) tag.
- Recursive case:

1) Prepare the buffer.
   ⇒ If the buffer is empty (it is the first function call at this level), the upper half is filled with LL lines from the previous level. These lines are calculated through recursive calls, decreasing in one the parameter *level*, i.e., using the function call GetLLlineBwd(*level*-1). A 1D DWT is applied on each line before it is introduced in the buffer. Once the upper half is full, the bottom half is filled through symmetric extension.
   ⇒ Otherwise, the buffer has to be updated. The buffer is shifted twice so that two lines are discarded and two new lines are introduced from the previous level. These two new lines are calculated using a recursive call again. If the recursive call returns EOL, the two empty slots are filled using symmetric extension.
2) A low and a high pass filter are applied to each column in the buffer. This way, we get a line from every wavelet subband (LH, HH and HH) and another line from the low-frequency (LL) subband at this level. The wavelet coefficients are passed to the encoder and the function returns the LL line.

Using this recursive function, an *nlevel* DWT can be easily computed calling this function with *nlevel* as parameter, until it returns EOL (that is, until no more LL lines can be read). This way, we also get the LL_{nlevel} subband, which has to be passed to the encoder to be compressed. Thus, the entire wavelet transform is computed using the following function:

- *function* LowMemoryUsageWT(*nlevel*)
- Set all the buffers empty at every level.
- While there are LL lines to be computed at the last level (*nlevel*).
   ⇒ Get an LL line using the function call GetLLlineBwd(*nlevel*). This line is passed to the encoder because it belongs to the LL subband of the wavelet decomposition. The computation of this line causes the computation of all the wavelet lines in the descendant subbands representing the same area that this LL line represents. When we define coefficient trees (like in [10] and [11]), the recursive function ensures that each descendent coefficient is always computed before any of its parent coefficients. This property will be of interest when we define the tree-based encoder.

The inverse wavelet transform is similar but recursion is carried out forward, starting from 0 to *nlevel*. A formal and detailed description of both the FWT and IWT can be found in [9].

Another interesting improvement of this algorithm is the use of the lifting scheme [12] instead of a simply convolution filter. Using the lifting scheme, we can expect faster execution and lower memory consumption, since the number of weighting factors employed in the lifting scheme is usually lower than the number of filter taps needed in the equivalent filter bank. The derivation from a filter bank to the weighting factors for the lifting scheme is given in [3], where it is shown that asymptotically, as the filter length grows, the memory consumption and number of required floating-point operations are halved when the lifting scheme is used.

3. SIMPLE CODING OF THE WAVELET SUBBANDS

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 use an encoder buffer for each subband level in order to group several subband lines released by the DWT and encode them together. When we consider that there are enough lines in a buffer to perform efficient compression, the coding algorithm is called, passing the encoder buffer (*Buffer*) as parameter in the function call.

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 our coding algorithms, the quantization process is performed by two strategies: one coarser and another finer. The finer one consists in applying a scalar uniform quantization to the coefficients, and it is performed along with the DWT. An efficient way to do this uniform quantization is applying it with the filter normalization. On the other hand, the coarser one is based on removing bit planes from the least significant part of the coefficients, and it is performed while our algorithm is applied. Related to this bit plane quantization, we define *rplanes* as the number of less significant bits to

be removed, and we call significant coefficient to those coefficients $c_{i,j}$ that are different to zero after discarding the least significant *rplanes* bits, in other words, if $c_{i,j} \geq 2^{rplanes}$.

The wavelet coefficients are encoded as follows. The coefficients in the buffer are scanned column by column (to exploit their locality). For each coefficient in that subband, if it is not significant, a LOWER symbol is encoded with an adaptive arithmetic encoder, so as to point out that $c_{i,j} < 2^{rplanes}$ and hence that coefficient has been absolutely quantized to zero. However, if it is significant, we have to encode the quantized significant coefficient.

A significant coefficient is encoded by means of a symbol indicating the number of bits required to represent that coefficient. An arithmetic encoder with two contexts is used to efficiently store that symbol depending on the significance of the upper and left coefficients. As coefficients in the same subband have similar magnitude, an adaptive arithmetic encoder is able to represent this information in a very efficient way. However, we still need to encode its significant bits and sign. They are raw encoded to speed up the execution time.

The proposed simple algorithm is described as follows:

- *function* SimpleWaveletCoding(*Buffer*)
- The encoder buffer is scanned column-by-column. For each coefficient:
    - If it is not significant (it is zero after quantization, in other words, it is lower than $2^{rplanes}$), we encode it with a LOWER symbol using adaptive arithmetic coding.
    - Otherwise, the number of significant bits of the quantized coefficient is encoded using an arithmetic encoder. The significant bits and the sign of the coefficient are raw encoded.

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 [2]. Other simpler solutions are: to save every level in secondary storage separately so that it can be read in a different order, and to keep the compressed coefficients in memory, using a different bitstream buffer for each subband, so that once the image has been compressed, the subband levels can be stored or transmitted in the order required by the decoder. For the sake of simplicity, we will use the last option for the coders introduced in this section and in the next one, although any of them could be used.

The encoder presented in this section is simple but has several disadvantages. The lack of dependency between subbands yields to high robustness, but it also causes lower compression performance in the encoding process. The correlation among coefficients and subbands can be exploited in a more efficient way (like in [10] and [11]), as we can see in the next section, which takes this algorithm as a starting point to define a more efficient encoder, with similar characteristics to the one presented in this section.

Another disadvantage of this algorithm is its higher complexity when working at low bit rates. Observe that one arithmetic symbol is always encoded for every coefficient. The cost of this operation is not negligible, most of all if it is repeated many times. At low bit rates, most of the encoded coefficients are represented by the LOWER symbol. An arithmetic encoder is able to take profit from this redundancy and achieves reasonable performance. However, a simple grouping algorithm is needed in order to decrease the total number of encoded symbols, and thus the computational cost of this algorithm. In the next section, we also solve this problem.

4. TREE-BASED CODING OF THE WAVELET COEFFICIENTS

In this section we present a tree-based coder that is an extension of the simpler algorithm that we described in Section 3. Tree-based coders exhibit good R/D performance [10] [14]. However, they have not been widely used in low-complexity coding of wavelet coefficients. Moreover, previous tree-based proposals need the entire image in memory to work [8] [10] [11] [14] and so they cannot be used in our system. As we saw in Section 1, other wavelet encoders, such as the standard JPEG 2000 [6], achieve good R/D performance by means of bit-plane processing, with R/D optimization algorithms, and a large number of contexts, and thus they exhibit high complexity. Our proposal is simpler than JPEG 2000 and therefore faster.

In our algorithm, we will arrange coefficient in trees as in [11] [10]. We consider that a coefficient is a lower-tree root if that coefficient and all its descendants (but not the following ascendant) are lower than $2^{rplanes}$. The set formed by all these coefficients forms a lower-tree. In our proposed encoder, we use the LOWER symbol to encode an entire lower-

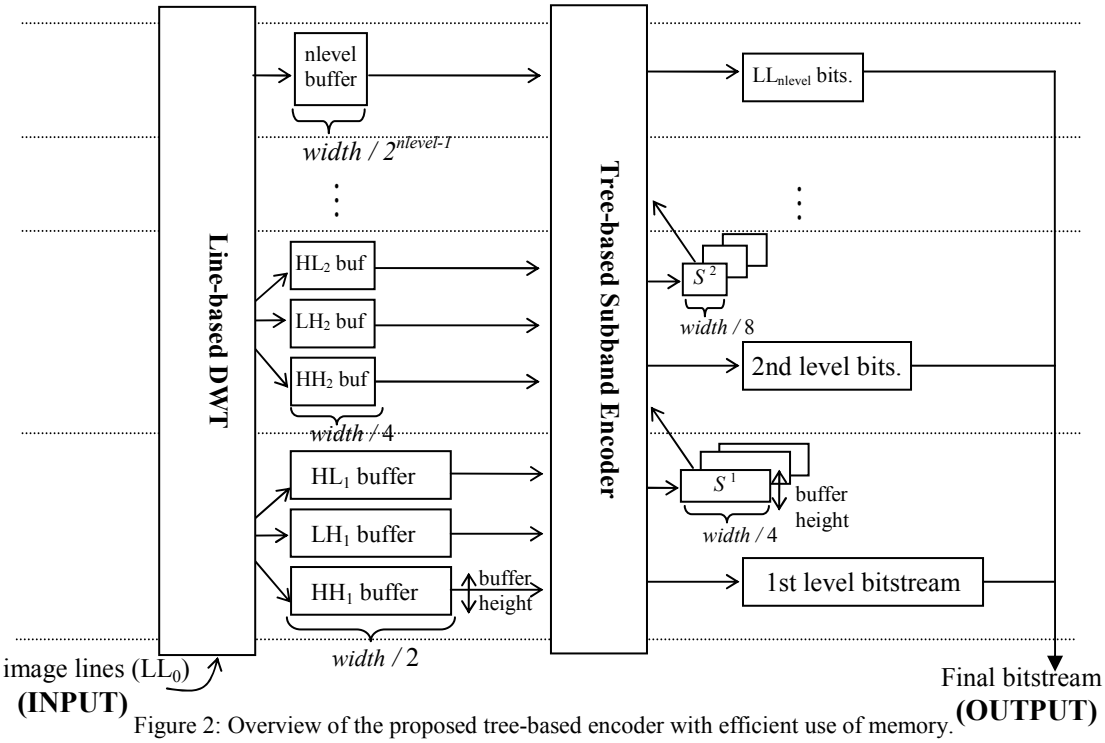

Figure 2: Overview of the proposed tree-based encoder with efficient use of memory.

tree. On the other hand, if a coefficient is lower than $2^{rplanes}$ but it does not belong to a lower-tree, because it has a significant descendant, it is considered as ISOLATED_LOWER.

Figure 2 shows our overall system. When the DWT releases subband lines, they are inserted into an encoder buffer, which is passed to the encoder system. The encoder determines if each 2x2 block of coefficients in the buffer is part of a lower-tree. If the four coefficients in the block are lower than the quantization threshold $2^{rplanes}$, and their descendant offspring are also insignificant, they are part of a lower-tree and do not need to be encoded. In order to know if their offspring are significant, we need to hold a significance map of every encoder buffer (S^L in the figure) because the encoder buffer is overwritten by the wavelet transform once its contents is encoded, and hence the significance for their ascendant coefficients is not automatically held. The width of each significance map is sized half the width of the encoder buffer that it represents, because it points out the significance of each 2x2 block in that buffer. The height is not the half but the same as the buffer height because each buffer at a level i is encoded with double frequency compared to the i+1 level. Moreover, the significance for each block can be held using only a bit, and therefore the memory required for these significance maps is almost negligible when compared with the rest of buffers.

On the other hand, when a 2x2 block is encoded and there is any significant coefficient in the block, or a coefficient in the block has a significant descendant coefficient, we need to encode each coefficient in the block separately. For each coefficient, if that coefficient and all its descendants are insignificant, we use the LOWER symbol to encode the entire tree. Otherwise, if it is insignificant, but the significance map of its four direct descendant coefficients shows that it has a significant descendant, that coefficient is encoded as an ISOLATED_LOWER. Finally, when a coefficient is significant and it has any significant descendant, it is encoded in the same way as the significant coefficients in the algorithm described in Section 3 (i.e., with a symbol indicating the number of bits required, and using raw coding for the significant bits and sign); however, if the coefficient is significant but all their descendants are insignificant, we need to encode that special case of lower tree (in which that coefficient is the root) in a different way, we encode it using a different set of symbols in order to show both the number of bits required by that coefficient and the fact that their descendant coefficients are insignificant.

It is important to see that in this algorithm, trees are built from leaves to roots, spreading the significance of the coefficients. Fortunately, this is exactly the order in which the line-based DWT algorithm described in Section 2 generates the wavelet coefficients. For example, the first two lines that are released by the wavelet transform are the first

two lines of the wavelet subbands at the 0 level; then, the following generated line is the first line of the first level. The coefficients in this new line are the ascendants of those computed in the two first lines. This order holds for the rest of the wavelet transform computation at all the levels, i.e., as we get two lines at a level *i*, we then get its parents at *i*+1 before getting two more lines at *i*.

The algorithm to encode a buffer is described as follows:

- *function* TreeBasedWaveletCoding(*Buffer*)
- The encoder buffer is scanned in 2x2 blocks (in column order). For each block:
    - If the four coefficients in the block are insignificant (i.e., they are zero after quantization), and the offspring of the coefficients in the block are also insignificant (according to their significance map), no symbol is encoded. In this case, we only have to spread the tree. We do it just marking this block as insignificant in the significance map of this level buffer.
    - Otherwise, this block is marked as significant in the map, and its coefficients are encoded as follows: For each coefficient in the block
        - ⇒ if it is insignificant then, if its descendant block is marked as insignificant in the significance map, we encode that entire tree with a symbol LOWER, otherwise we encode that isolated insignificant coefficient with an ISOLATED_LOWER symbol.
        - ⇒ otherwise (the coefficient is significant) the number of significant bits of the quantized coefficient is encoded using an arithmetic encoder. We also need to encode the significance of its descendant block, to this end, we use a different symbol set if the four descendant coefficients are not significant. Finally, the significant bits and the sign of the coefficient are raw encoded.

At the last level (*nlevel*), the tree cannot be propagated upward, and we always encode all the coefficients. Moreover, as in the previous section, we can keep the compressed bit-stream in memory, which allows us to invert the order of the bitstream for the inverse procedure.

The tree-based algorithm that we have introduced in this section is based on the LTW encoder presented in [8]. The main difference between both encoders is that the earlier LTW encoder needs all the wavelet coefficients (i.e., the transformed image) in memory to work, and performs a 2-pass coding process. This new version of the LTW can be used along with a line-based wavelet transform and employs 1-pass coding.

5. PRACTICAL RESULTS

We have implemented the proposed coders in ANSI C language. In this section we will compare it with the state-of-the-art wavelets coders SPIHT [10] and JPEG 2000 [6]. The results for JPEG 2000 have been obtained using Jasper [1], an official implementation included in the ISO/IEC 15444-5 standard. All of them use the same wavelet filter bank (Daubechies' B7/9) and have been written and compiled with the same level of optimization. In our comparison, we will use the standard images Lena and Barbara (monochrome, 8bpp, 512x512) and the larger and less blurred images Café and Woman (monochrome, 8bpp, 2560x2048, equiv. 5-Megapixel), from the JPEG 2000 testbed.

Table 1. PSNR (dB) with different bit rates and coders for the evaluated images.

	Lena (512x512)				**Barbara (512x512)**			
	SPIHT	**Jasper / JP2K**	**Proposed simple**	**Proposed Tree-Based**	**SPIHT**	**Jasper / JP2K**	**Proposed simple**	**Proposed Tree-Based**
1	40.41	40.31	40.26	40.45	36.41	37.11	36.47	36.58
0.5	37.21	37.22	37.05	37.29	31.39	32.14	31.59	31.63
0.25	34.11	34.04	33.93	34.23	27.58	28.34	27.87	27.95
0.125	31.10	30.84	30.87	31.23	24.86	25.25	25.04	25.16
	Woman (2560x2048)				**Café (2560x2048)**			
	SPIHT	**Jasper / JP2K**	**Proposed Simple**	**Proposed Tree-Based**	**SPIHT**	**Jasper / JP2K**	**Proposed simple**	**Proposed Tree-Based**
1	38.28	38.43	38.27	38.46	31.74	32.04	31.64	31.96
0.5	33.59	33.63	33.55	33.77	26.49	26.80	26.53	26.82
0.25	29.95	29.98	29.99	30.13	23.03	23.12	22.98	23.24
0.125	27.33	27.33	27.33	27.52	20.67	20.74	20.59	20.79

Table 1 shows a compression comparison for the evaluated images and coders. In general, our tree-based proposal performs as well as SPIHT does for the less detailed image Lena, and better than SPIHT for the rest of images (Barbara, Woman and Café), which are more complex. On the contrary, JPEG 2000 is more efficient than our proposal in the highly detailed image Barbara, since it defines more contexts and uses an R/D optimization algorithm, but our tree-based encoder is equal or better than JPEG 2000 for the rest of images (in general, the less detailed, the better our algorithm is compared with JPEG 2000, because fewer trees can be established in images with many details). In table 1 we also include the R/D results for the simple algorithm proposed in Section 3, in order to show the benefit achieved by the introduction of trees in our encoder.

The comparison in which our encoders (both the simple and tree-based encoders) clearly outperform both SPIHT and JPEG 2000 is in memory consumption. Table 2 shows that for a 5-Megapixel image, our proposals require between 30 and 40 times less memory than SPIHT, and more than 60 times less memory than Jasper/JPEG 2000. In this table, the last column refers to the case in which the complete bitstream (i.e., the compressed image) is kept in memory while it is generated. Due to the computation order in the proposed wavelet transform, the coefficients from different subband levels are interleaved. Thus, instead of a single bitstream, we generate a different bitstream for every level. These different bitstreams can be kept in memory or saved in secondary storage. In addition, recall that having a different bitstream for each level eases the decompression process, since the order in the inverse transform is just the reverse of the order in the forward one.

This memory comparison has been performed under Windows XP, where the memory estimated for executing a single process (compiled in C) is about 650 KB. Hence, we can consider that the remaining memory is the data memory.

Since JPEG 2000 has more contexts and uses an R/D optimization algorithm, it is more complex than our tree-based 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 makes a more efficient use of it. The last table shows an execution time comparison for a 5-Megapixel image. Our algorithm clearly outperforms Jasper/JPEG 2000, being more than 20 times faster than it, and it is three times faster than SPIHT. On the other hand, in this table, the comparison of the execution time between the algorithms proposed in Section 3 and Section 4 shows that the use of coefficient trees is not only an efficient method of coding coefficients but also a fast way of grouping them. Moreover, these improvements in execution time are more noticeable in lower bit rates, since more coefficients can be encoded with a single symbol.

For more tests, the reader can download the proposed coders at the authors' web site http://www.disca.upv.es/joliver/treecoder.

Table 2. Total memory required (in KB) to encode a 5-Megapixel image with the compared algorithms. We give two results for our tree-based algorithm. In the first one, every different bitstream at each level is kept in secondary storage, while in the second one it is kept in memory. The memory consumption results for the simple algorithm proposed in Section 3 are roughly the same as those presented in this table.

Codec \ Rate	**Compressed image file**	**SPIHT**	**Jasper / JP2K**	**Proposed Tree-Based**	**Proposed with bit-stream in memory**
1	640	42,888	62,768	1,096	1,736
0.5	320	35,700	62,240	1,032	1,352
0.25	160	31,732	61,964	1,032	1,192
0.125	80	28,880	61,964	1,016	1,096

Table 3. Execution time (in Million of CPU cycles) needed to encode a 5-Megapixel image.

Codec \ Rate	**SPIHT**	**Jasper / JP2K**	**Proposed Simple**	**Proposed Tree-Based**
1	3,669	23,974	1,650	1,067
0.5	2,470	23,864	1,462	760
0.25	1,939	23,616	1,383	591
0.125	1,651	23,563	1,359	503

6. CONCLUSIONS

In this paper, we have presented a complete tree-based wavelet image coder (both wavelet transform and tree-based entropy coding stages) with state-of-the-arte compression performance. The main contribution of this image coder is that it requires much less memory to work and so, it is a good candidate for many embedded systems and other memory-constrained environments (such as digital cameras and PDAs). In addition, it is also several times faster than other evaluated wavelet image coders.

ACKNOWLEDGMENTS

This work was supported in part by the Spanish Ministerio de Ciencia y Tecnología, funded with the CICYT project under grant 20040052

REFERENCES

1. M. Adams, "Jasper software reference manual," ISO 1/SC 29/WG 1 N 2415, October 2002.
2. C. Chrysafis, and A. Ortega, "Line-based, reduced memory, wavelet image compression," *IEEE Transactions on Image Processing*, March 2000.
3. I. Daubechies, W. Sweldens, "Factoring wavelet transforms into lifting steps," *J. Fourier Anal.*, no.3, 1998.
4. P. Cosman, K. Zeger, "Memory constrained wavelet-based image coding," *Proc. UCSD Conf. Wireless Communications*, March 1998.
5. ISO/IEC 10918-1/ITU-T Recommendation T.81, *Digital Compression and Coding of Continuous-Tone Still Image*, 1992.
6. ISO/IEC 15444-1: JPEG 2000 image coding system, 2000.
7. S. Mallat, "A theory for multiresolution signal decomposition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, July 1989.
8. J. Oliver, M. P. Malumbres, "Fast and Efficient Spatial Scalable Image Compression Using Wavelet Lower Trees," *IEEE Data Compression Conference,* Snowbird, UT, March 2003.
9. J.Oliver, M.P.Malumbres, "A fast wavelet transform for image coding with low memory consumption," *24th Picture Coding Symposium*, December 2004.
10. A. Said, A. Pearlman, "A new, fast, and efficient image codec based on set partitioning in hierarchical trees," *IEEE Trans. on Circuits and Systems for Video Technology*, June 1996.
11. J.M. Shapiro, "Embedded Image Coding Using Zerotrees of Wavelet Coefficients," *IEEE Transactions on Signal Processing*, vol. 41, pp. 3445-3462, Dec. 1993.
12. W. Sweldens, "The lifting scheme: a custom-design construction of biorthogonal wavelets," *Appl. Comput. Harmon. Anal.*, 3, pp. 186-200, 1996.
13. M. Vishwanath, "The recursive pyramid algorithm for the discrete wavelet transform," *IEEE Transactions on Signal Processing*, March 1994.
14. Z. Xiong, K. Ramchandran, M. Orchard, "Space-Frequency Quantization for Wavelet Image Coding," *IEEE Transactions on Image Processing*, vol. 46, pp. 677-693, May 1997.
15. X. Wu. "High-order context modelling and embedded conditional entropy coding of wavelet coefficients for image compression," in *Proc. 31st Asilomar Conf. Signals, Systems, Computers,* VOL. 23, pp. 1378-1382, 1998.
16. X. Wu. "Compression of Wavelet Transform Coefficients," The Transform and Data Compression Handbook, pp. 347-378, CRC Press, 2001.
17. D. Taubman, "High Performance Scalable Image Compression with EBCOT," *IEEE Transactions on Image Processing*, vol. 9, pp. 1158-1170, July 2000.