Line Based, Reduced Memory, Wavelet Image Compression

Christos Chrysafis
Hewlett-Packard Laboratories
1501 Page Mill Road, Bldg.3U-3
Palo Alto, CA 94304-1126
chrysafi@hpl.hp.com
Tel : 650-857-3382, Fax: 650-857-4691

Antonio Ortega*
Integrated Media Systems Center
University of Southern California
Los Angeles, CA 90089-2564
ortega@sipi.usc.edu
Tel: 213-740-2320, Fax: 213-740-4651

EDICS: IP 1.1 Coding

Abstract

This paper addresses the problem of low memory wavelet image compression. While wavelet or subband coding of images has been shown to be superior to more traditional transform coding techniques, little attention has been paid until recently to the important issue of whether both the wavelet transforms and the subsequent coding can be implemented in low memory without significant loss in performance. We present a complete system to perform low memory wavelet image coding. Our approach is "line-based" in that the images are read line by line and only the minimum required number of lines is kept in memory. The main contributions of our work are two. First, we introduce a line-based approach for the implementation of the wavelet transform, *which yields the same results as a "normal" implementation*, but where, unlike prior work, we address memory issues arising from the need to synchronize encoder and decoder. Second, we propose a novel context-based encoder which requires no global information and stores only a local set of wavelet coefficients. This low memory coder achieves performance comparable to state of the art coders at a fraction of their memory utilization.

1 Introduction

Memory is an important constraint in many image compression applications. In some cases, especially for mass market consumer products such as printers or digital cameras, this is due to the need to maintain low costs. In other cases, even if sufficient memory is available (e.g.,

*This work was supported in part by the National Science Foundation under grant MIP-9502227 (CAREER), the Integrated Media Systems Center, a National Science Foundation Engineering Research Center, the Annenberg Center for Communication at the University of Southern California, the California Trade and Commerce Agency and by Texas Instruments. An early version of the this work was first published in [1]

image encoding/decoding in a PC or workstation), inefficient memory utilization may limit scalability and hinder overall performance. For example, if for a given algorithm doubling of the image size results in doubling of the memory requirements, the practicality of using the algorithm over a wide range of systems may be questionable.

Existing Discrete Cosine Transform (DCT) based compression algorithms such as those defined under the JPEG standard [2] are very efficient in their memory utilization because, if needed, they can operate on individual image blocks and thus the minimum amount of memory they require is low indeed (e.g., a system could conceivably be implemented so as to manipulate a single image block at a time). Wavelet based coders have been shown to outperform DCT based coders in terms of compression efficiency, but their implementations have not yet reached the stage of maturity of DCT based approaches [2]. Memory efficiency is in fact one of the key issues to be addressed before a widespread deployment of wavelet based techniques takes place and it is currently one area of major research activity within the JPEG2000 standardization process [3].

Algorithms such as those in [4, 5, 6, 7, 8, 9, 10, 11], are representative of the state of the art in wavelet coders. All of these algorithms assume that the wavelet transform (WT) for the whole image has been computed so that all the corresponding coefficients are available in the coding process. Global image information[1] is used in various ways including, among others, classification (e.g., [7]), initialization of the probability models used by a quantizer or arithmetic coder [8, 9, 10] or selection of specific decompositions of the signal [6]. It is also worth noting that even if no global information has to be measured, algorithms that provide progressive transmission [4, 5] might require to store the complete set of wavelet coefficients. The above mentioned algorithms were developed with the goal of achieving competitive compression performance and thus memory utilization was not a major consideration in their development. These algorithms are typically required to buffer the whole image at the encoder, so that memory usage increases proportionally to the image size, without such factors as filter length, or the number levels of the wavelet decomposition affecting significantly the memory utilization.

Low memory implementations of wavelet transforms were first addressed in [12], which only considered one-dimensional (1D) transforms and did not consider the synchronization issues that arise when both forward and inverse transform memory requirements are consid-

[1] i.e., information that can only be obtained after the whole image has been transformed. Examples of global information include the maximum and minimum coefficient values in one subband, the energy per subband, histograms of coefficient values in a subband, etc.

ered. Interest in memory issues has recently increased as memory needs for the WT have been found to be the one of the main bottlenecks for wavelet-based image compression. There have been several recent studies of hardware issues in the implementation of the WT [13, 14]. Many of these studies consider an in-depth analysis of the whole wavelet transform system, including architecture level optimizations and memory usage as in [13], but do not consider the joint design of transform and compression algorithm to guarantee low memory operation. In addition, much of this work has focused on video compression implementations, where images can be orders of magnitude smaller than some of those processed in the hard-copy industry, and thus the proposed approaches might not scale well to large image sizes. For example, typical designs consider an on chip memory to handle the filtering operations [15] and such memory can become prohibitively large when images of hundreds of millions of pixels or more have to be processed.

Assume that a particular WT implementation can handle small images efficiently. Obviously there are approaches to use such an implementation for wavelet coding of large images. The most immediate approach is to tile the large image and encode each tile independently of the others, i.e., as if each tile were a separate image. While tiling is a simple approach it can present some serious drawbacks, especially if the tile size is small with respect to the image size. For example as compression is performed independently, blocking artifacts may appear at the boundaries between tiles. Moreover, since it is not easy to allocate bits among the tiles (since the wavelet coefficients of all the tiles are not known and each tile is treated independently) the performance degradation due to tiling may be severe.

An alternative and more efficient approach can be found in the recent work of Cosman and Zeger [16, 17]. Here, the memory utilization of the encoder is left unchanged and a standard algorithm (e.g. [5]) can be used to compress the image. The whole image is buffered at the encoder, but the order of transmission of the bit-stream is altered from that used in a normal implementation so that the memory utilization at the decoder is reduced. The basic idea is to have the decoder receive "local" sets of encoded wavelet coefficients so that the inverse wavelet transform can be started without having to wait for all the coefficients to be decoded. Performance can be further improved by selecting filters so that the number of coefficients required at the decoder remains small (this can be achieved for example by choosing shorter filters for filtering along the vertical direction.) We refer the reader to [16, 17] for further details.

In summary, the recent work on memory efficient wavelet image coding does not consider

a complete coding system but concentrates instead on the WT or the compressor alone, or only considers either encoder or decoder, but not both.

Our proposed approach differs from earlier work in several ways. First, we consider the overall memory utilization and propose a system with reduced memory *at both encoder and decoder*, whereas [12] and [16, 17] addressed only the memory usage at encoder and decoder, respectively. We thus consider the memory needed for synchronization between encoder and decoder (e.g. a minimum memory forward wavelet transform may require a high memory inverse wavelet transform). Second, we consider a complete coding system, i.e., including both WT and quantization plus entropy coding, and propose a novel context-based compression approach to provide high compression performance with reduced memory utilization. With respect to existing coders, the degradation in performance is modest (e.g., less than $0.5dB$ in average with respect to [9]) although we do not support progressive transmission as [4, 5]. In terms of memory utilization our results show reductions of almost *two orders of magnitude* with respect to widely available implementations of wavelet image coders. In fact, our proposed low memory WT implementation has been adopted within the latest version of the JPEG 2000 verification model.

Our proposed system includes a line-based implementation of the WT, where we assume the image data is available one image line at a time. We begin by analyzing the minimum memory requirements to compute the wavelet transform in Section 2. Analyzing the 1D WT allows us to discuss the various ways in which memory is utilized, including filtering but also synchronization between encoder and decoder. We extend these analyses to the two-dimensional (2D) WT and propose an implementation that requires the minimum number of image lines to be stored for a given filter length and number of levels of decomposition. With this approach the memory needs of the encoder and decoder depend only on the width of the image (rather than the total size as in a traditional row column filtering implementation) which significantly enhances the scalability of the system. Moreover, appropriate choices of filters (e.g., short filters for the vertical filtering) can be used as in [16, 17] to further reduce the memory requirements.

In Section 3 we then propose a backward adaptive context-based coding scheme which utilizes *only* a reduced number of coefficients stored at the encoder or decoder at a given time. While this approach precludes the use of any global information we show that competitive performance can be achieved because, as has been shown in [10, 9, 8, 7], there exists significant localization of energy within wavelet subbands, which can be efficiently exploited with

context-based methods. We provide a complete description of our algorithm and highlight the modifications that had to be undertaken with respect to our earlier context-based coding approaches [9] in order to compensate for the lack of global information.

Our experimental results are presented in Section 4, where we include comparisons with several algorithms [9, 5, 11, 2], which all have significantly larger memory requirements. Our results indicate that the low memory approach we propose can achieve excellent compression performance with significantly lower memory utilization. In section 5 we revise the main contributions of our work.

2 Line-based 2D wavelet transform

Image data is usually acquired in a serial manner. For example, a very common way to acquire image data is to scan an image one line at a time. Throughout this paper we will assume our system operates with this line-by-line acquisition. Given this, our objective in this section will be to design a 2D, WT that requires *storing a minimum total number of lines.* The assumption is that images are stored in memory only while they are used to generate output coefficients, and they are released from memory when no longer needed.

Obviously, performing a 1D WT on a single line can be done without significant memory. However, in order to implement the separable 2D transform the next step is to perform column filtering and here memory utilization can become a concern. For example, a completely separable implementation would require that all the lines be filtered before column filtering starts and thus memory sizes of the order of the image size will be required. The obvious alternative is to start column filtering as soon as a sufficient number of lines, as determined by the filter length, has been horizontally filtered. For example for a one level decomposition if we use 9-7 tap filters we only need 9 lines of the image in order to start the column filtering and generate the first line of output wavelet coefficients.

This online computation approach, which is described in more detail in [12] for the 1D case, will form the basis of our wavelet filtering. This will allow us to store in memory only a reduced number of input lines. The memory needs, as will be discussed in what follows, depend not only on the filter length but also on the number of levels of decomposition and the type of decomposition. For example generic *wavelet packet* [18] decompositions require different structures than a *dyadic tree* decomposition and need to be considered separately. In addition, the order in which lines of wavelet coefficients are generated at the analysis filter bank is not the same order the synthesis filterbank expects them and we will thus need

to introduce some additional "line management" functionality to synchronize encoder and decoder.

2.1 One dimensional wavelet transform

Let us consider first an implementation of a 1D WT, where the system receives data sequentially, one pixel at a time. Let $L = 2S + \phi$ be the maximum length of the filters used in the analysis filterbank, which can be either odd or even, for $\phi = 1$ and $\phi = 0$, respectively. In the next sections, without loss of generality, we will only consider the odd length filter case. The even length filter case can be treated in a similar way. For compression efficiency we use symmetric extensions throughout this paper. Similar delay analyses can be found in [12], although syncronization delays were not considered there.

Consider first a single stage of the WT. At time zero we start receiving data and store it in a shift register as seen in Figure 1. At time S we have received enough data to fill the entire input buffer, i.e., we have received $S+1$ samples and after symmetric extension we can have the total of $L = S+1+S$ samples we need for filtering. Thus, the *delay* for generating output coefficients for one level of decomposition is S. Note that it is more efficient to generate two coefficients at a time as shown in Figure 1, i.e., we read two input samples at a time and generate both a low-pass and a high pass coefficient.

Consider now a two-level decomposition, as shown in Figure 2. Let ρ be the sample rate (in samples per second) at the input of the filterbank. Then the sample rate at the output of the first level will be $2^{-1}\rho$. Let us consider the interval between input samples (i.e., $1/\rho$ seconds) as our basic time unit. Each individual filterbank introduces an S sample delay. However the input to the second filterbank arrives at a lower rate $2^{-1}\rho$, due to downsampling. Thus to begin filtering and see the first outputs of the second filterbank we will have to wait (i) S time units for the first output of the first level filterbank to be generated, and then (ii) another $2S$ time units until sufficient samples have been generated at rate $2^{-1}\rho$. Thus, the total delay from input to output of the second level in the system of Figure 2 is the sum of those individual delays, i.e., $S + 2S$ input samples.

This analysis can be easily extended to the the case where an N-level decomposition is used. Assume that the levels in this decomposition are indexed from 0 to $N-1$, where 0 corresponds to the first level, 1 corresponds to the second level, and so on. It will take $2^n S$ time intervals for S samples to be loaded in the n^{th} level filters and this will happen only after outputs have been generated by levels 0 to $n-1$. Thus the total delay from the input

to the output of an N level filterbank will be the sum of all the individual delays for each level, $D_N = S + 2S + 2^2S + 2^3S + \ldots + 2^{N-1}S = \sum_{k=0}^{N-1} 2^k S = (2^N - 1)S$. The delay from the n^{th} level to the output will be the same as the delay from the input to the output of an $N - n$ level decomposition, i.e., $D_{n,N} = D_{N-n} = (2^{N-n} - 1)S$.

The memory needed for *filtering* will be L samples[2] for each level of decomposition, i.e., the total will be $L \cdot N$ if we use a dyadic tree decomposition. In general we will just need an additional memory of size L for each additional 2-channel filter-bank added to our wavelet tree.

In the synthesis filter-bank the delays from input to output are the same as in the analysis filter-bank and are thus a function of the number of levels of decomposition. Note that, referring to Figs. 2 and 3, the synthesis filterbank will not be able to process $x_1^{(0)}$ until it has processed a sufficient number of $x_0^{(1)}, x_1^{(1)}$ coefficients to generate $x_0^{(0)}$. However the analysis bank generates $x_1^{(0)}$ with less delay than $x_0^{(1)}, x_1^{(1)}$. Thus we will need to store a certain number of $x_1^{(0)}$ samples while the $x_0^{(1)}, x_1^{(1)}$ samples are being generated. We will call the required memory to store these samples *synchronization* buffers.

Because $2S$ samples at level 1 are produced before the first sample at level 2 is produced, we will need a synchronization memory of $2S$ samples (see Figure 3). The required memory can be split into two buffers of size S pixels, with one buffer assigned to the analysis filterbank and the other to the synthesis filterbank.

In the more general N-level case the delay for samples to move from level n to level $N-1$ is D_{N-n}. The synchronization buffer for level n is equal to the delay for data to move from level n to level $N-1$ which is also D_{N-n}, thus the total buffer size needed for *synchronization* is $T_N = \sum_{k=0}^{N-1} D_{N-k} = \sum_{k=1}^{N} D_k = (2^N - N - 1)S$. For a five level decomposition the size of the synchronization buffers can be seen in Figure 4.

As a conclusion, for the 1D case when considering both analysis and synthesis banks, our design will have to optimize memory for both *filtering* and *synchronization*. In the above analysis we have kept analysis and synthesis filterbanks symmetric in terms of memory needs. However, synchronization buffers can be easily assigned to either the analysis or the synthesis filterbanks if it is necessary to make one more memory-efficient than the other. In the rest of

[2]Implementations with memory sizes of $S + 1$ samples (or lines) are also possible, but here we assume storage of L lines to facilitate the description of the synchronization problems. More efficient approaches based on lifting implementations or lattice structures, can asymptotically bring the memory needs from L down to $S + 1$. For example the lattice structure used in [14] can help in reduce both complexity and memory. These structures will not be considered here since the memory savings are filter dependent and do not significantly affect our proposed solution. Lifting or lattice structures can be used within our framework to provide additional memory reductions.

the paper we will only consider symmetric systems. In summary, for 1D signals and N levels of decomposition, the memory needs for the n^{th} level will consist of

- a filtering buffer of size L, and
- a synchronization buffer of size $D_{N-n} = (2^{N-n} - 1)S$.

Therefore the total memory size needed for N levels of decomposition in a symmetric system is: $T_{total,N} = (2^N - N - 1)S + NL$. Note that as the number of levels becomes large synchronization buffers become a major concern.

2.2 Two dimensional wavelet transform

Let us now generalize our memory analysis to two dimensions. As a simplifying assumption we assume that horizontal filtering is performed in the usual manner, i.e., our memory budget allows us to store complete lines of output coefficients after horizontal filtering. Thus, after each line is received all the corresponding filter outputs are generated and stored in memory, requiring a memory of size X for each line, where X is the width of the image. Thus we can now apply the above analysis to the vertical filtering operation, except that the input to the WT is now comprised of *lines* of output coefficients generated by horizontal filtering and thus the memory sizes shown above have to be adjusted to account for line buffering requirements.

The exact memory requirements depend on the structure of decomposition. Figs. 5(a) and (b) depict the common *dyadic tree* decomposition and a "hybrid" decomposition, which have the same memory requirements. Let us concentrate on the decomposition of Figure 5(a). In order to implement a one level decomposition vertically we need to buffer L lines[3]. At the second level of the decomposition again we will need to buffer L lines, but the length of each line will be $X/2$ coefficients, because in the dyadic composition the second level decomposition is only applied to the low pass coefficients generated by the first level. Thus the width of our image is reduced by two each time we move up one level in the decomposition, and, correspondingly, the memory needs are reduced by half each time. For N levels we will need $\sum_{k=0}^{N-1} 2^{-k} L = 2(1 - 2^{-N})L$ "equivalent" image lines for filtering (refer to Figure 6). As N grows the required number of lines tends to $2L$, and the corresponding memory becomes $2LX$ i.e., asymptotically we only need a number of lines equal to twice the filter length. The above analysis is valid for both encoder and decoder as long as we use the decompositions in Fig. 5.

[3]As indicated before, implementations with only $S + 1$ lines are possible. They are not considered here since the memory savings in this case come at the cost of a more complicated system.

As in the 1D case, synchronization issues have to be addressed because coefficients from the first level of decomposition are generated by the analysis bank before coefficients from higher levels, while the synthesis filter-bank requires the higher level coefficients before it can use the coefficients from the first level. For example in a two-level decomposition we will need to store the $\mathrm{HL}_0, \mathrm{LH}_0, \mathrm{HH}_0$ bands[4] since these bands become available before the $\mathrm{HL}_1, \mathrm{LH}_1, \mathrm{HH}_1$ bands, and the synthesis filterbank has to start processing data from the second level before it can process data from the first level.

Thus, as in the 1D case, in an N-level decomposition the synchronization delay required for data at level n is $D_{N-n} = (2^{N-n} - 1)S$ lines[5], where the width of each line is the width of a subband at a particular level, e.g., the width of one line at level n will be $2^{-n-1}X$, due to down-sampling and the use of a dyadic decomposition. Because 4 bands are generated at each level, but only one (i.e. the LL band) is decomposed further, we will need synchronization buffers for the remaining three subbands. Thus the synchronization buffers for level n will have a total size of $3 \cdot (2^{N-n} - 1)S \cdot 2^{-n-1}X$ pixels and the total memory needed for synchronization will be:

$$T_N^{(2d)} = 3 \sum_{k=0}^{N-1} (2^{N-k} - 1)SX2^{-k-1} = (2 \cdot 2^N + 2^{-N} - 3)XS \text{ pixels} \quad (1)$$

From (1) we see that the size of the delay buffer increases exponentially with the number of levels, while as discussed before the memory needed for filtering is upper bounded by $2LX$ pixels. Thus, as the number of decomposition levels grows, the filtering requirements remain relatively modest, while the size of the synchronization buffers tends to grow fast. However, memory-efficient implementations are still possible because the filtering buffers hold data that is accessed multiple times, while the synchronization buffers are only delay lines (FIFO queues). This is a key distinction because, as will be described later, the data in the synchronization buffers will only be used by the decoder and therefore it can be *quantized and entropy coded* so that the actual memory requirements are much lower.

In summary, for an N-level dyadic decomposition, we will need

- filtering buffers for up to $2LX$ pixels, and
- synchronization buffers for $T_N^{(2d)} = (2 \cdot 2^N + 2^{-N} - 3)XS$ pixels.

The decomposition of Fig. 5(b) can be implemented with the same memory as that of Fig. 5(a). For the case of Fig. 5(b), we perform five levels of horizontal decomposition when we

[4]Index 0 corresponds to the first level of decomposition

[5]Note that we express this delay in terms of number of input lines, instead of pixels

first read a line and we skip horizontal filtering in all the subsequent decomposition levels in Figure 6. This decomposition gives better compression results for images having a significant part of their energy in the high frequencies. For a specific wavelet packet decompositions the exact structure of a line-based implementation would have to be determined on a case-by-case basis, and thus the above formulas do not apply. However, special decompositions such as the one in Figure 5(b), having the same memory requirements as the simple "dyadic tree" decomposition, can provide some of the gains of a wavelet packet decomposition, while being memory efficient.

2.3 Example

To illustrate our WT algorithm let us consider an example, with $N = 5$ levels of decomposition. Refer to Figure 6. After filtering, data from decomposition level n are passed on to decomposition level $n + 1$. Assume we are at level 2 and we are receiving data from level 1. Each time we receive a line we perform horizontal filtering and we store the data into a circular buffer that can hold L lines. After we have received $S + 1$ lines we can perform vertical symmetric extension inside the buffer. We will end up with L lines that have already gone through a horizontal decomposition and we are then ready to perform vertical filtering. We can generate 2 output lines at once since we have all the necessary input lines. One line will have vertical low pass information and the other will have vertical high pass information. Moreover, half the coefficients in each line will contain horizontal low pass information and the other half will contain horizontal high pass information. As output we have four lines $\mathrm{LL}_2, \mathrm{LH}_2, \mathrm{HL}_2, \mathrm{HH}_2$ of length half the length of our vertical buffer at this level. The LL_2 line needs to go to the next decomposition level, i.e., level 3, while the lines $\mathrm{LH}_2, \mathrm{HL}_2, \mathrm{HH}_2$ need to go through the synchronization buffer FIFO_2. The width of each input line for level n is $2^{-n}X$, while the width for each output line is $2^{-n-1}X$. This process continues for each level of decomposition, up to the last level (level $N - 1$).

2.4 Speed advantages

The WT implementation described in the previous sections provides significant memory savings, as compared to a "naive" row column filtering implementation, which will require a memory size of the order of the image size. This memory efficiency is advantageous also in terms of computation speed. Obviously, the number additions and multiplications is exactly the same in our line-based implementation as in the row column filtering implementation.

However in image compression applications we frequently deal with large images such that the whole image does not fit into the processor cache memory. If the cache memory is insufficient all the image pixels will have to be loaded into cache several times in the course of the WT computation. For example, in a software environment the operating system and the processor are responsible for loading and unloading certain parts of the whole image into cache, and a memory inefficient approach will result in increased memory management overhead. In the naive approach, output coefficients generated by horizontal filtering will have to be removed from the cache, then reloaded when column filtering is performed. Instead, in our proposed system, the enforced "locality" of the filtering operations makes it more likely that strips of the image get loaded into the cache only once [6]. This fact alone reduces the traffic through the buses of the processor and cuts the bandwidth for memory access by orders of magnitude. Cache memory is around 5 times faster than main memory, so we expect speed ups of around 5 times by using our approach. In software implementations these were indeed our observations in simulations. Also the larger the size of the image the greater the speed advantages offered by our algorithm. No optimization was performed in our code and yet to the best of our knowledge our algorithm provides the fastest currently available software implementation.

3 Low memory entropy coding

3.1 Desired characteristics of a low memory compression scheme

In order to use our low-memory WT for an image compression application, and still keep the memory needs low, we need to ensure that wavelet coefficients are compressed soon after they have been generated. Wavelet coefficients are generated line by line in an interleaved fashion (i.e., as mentioned earlier, filtering operations generated both high-pass and low-pass data for a given input line), and therefore it will be useful to be able to encode the data in the order it is generated. Obviously, buffering all the coefficients from a certain band before they are coded increases memory requirements and should be avoided. Thus a low-memory encoder should be able to code data as soon as it becomes available, buffering up only a few lines before encoding (rather than entire subbands) and avoiding having to go through the wavelet coefficients more than once.

It should be noted that if providing an embedded bit stream is required it will be necessary

[6] Of course, depending of the size of the cache relative to the image size, we might need to load them more than once.

to perform several passes through the data and thus the memory requirements will be larger. An embedded coder will typically send the most significant bits of *all* the wavelet coefficients, whereas a memory efficient approach would tend to transmit coefficients (down to maximum level of significance) as they are produced. If an embedded bit-stream is desired then it will be necessary to store all the wavelet coefficients (so that the most significant bits of all coefficients can be sent first). Alternatively, with the appropriate bit-stream syntax, it may be possible to generate an embedded bit-stream by first storing a compressed image and then reordering the bit-stream before transmission (as in [19]). In either case, an embedded output requires more buffering than our proposed approach.

As indicated earlier, to reduce the synchronization memory requirements, it will be preferable to store compressed data in those buffers *after* compression. This is clearly advantageous since wavelet coefficients are usually kept in floating point format (32 bits) while after compression they can be stored with about one bit per coefficient on average. Thus one should keep in compressed form as much data as possible and avoid buffering uncompressed wavelet data.

3.2 Line based entropy coder

We now describe our proposed low-memory coding strategy, based on context modeling and classification along with arithmetic coding and probability estimation. Processing is based on the lines of wavelet coefficients produced by the WT. Each band is encoded *separately*, i.e. we do not use *any* kind of information from one band in order to encode another. Moreover no global information is required. Our purpose is to demonstrate that a line-based transform combined with a line-based coder can be competitive in terms of compression performance, at a fraction of the memory requirements of a more general algorithm like [9, 5, 11, 4, 10].

In the rest of the paper we will assume that all subband data are quantized with the same *dead-zone quantizer*, that is the step size δ of the quantizer is the same for all subbands. The quantization operation is a mapping from a wavelet coefficient α to the index $v = \lfloor \frac{\alpha}{\delta} \rfloor$. The inverse quantization is a mapping from the index v to an estimate $\hat{\alpha}$ of the original wavelet coefficient α.

$$\hat{\alpha} = \begin{cases} (v + 1/2) \cdot \delta & \text{if} \quad v > 0 \\ (v - 1/2) \cdot \delta & \text{if} \quad v < 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

We assume appropriate normalization of the filter coefficients, as discussed in [9], in order to compensate for the fact that the biorthogonal filter-banks we use here (to take advantage

of the symmetry properties) do not have norm one. This normalization allows us to use the same quantization step size for each band.

3.2.1 Introduction to Context Modeling

Context modeling is used in various forms in many compression algorithms. It has been the central component of lossless compression algorithms like [20, 21, 22], and is also widely applied in lossy compression environments [8, 9, 10]. We define the context information for a coefficient as the information we can obtain from the neighboring previously quantized and encoded coefficients[7]. Obviously context information is only useful if the data exhibits some sort of correlation, as is the case when considering wavelet coefficients obtained from filtering natural images. Consider for example a one dimensional signal $x_0, x_1, x_2, \ldots$. Context information corresponding to a coefficient x_n can be any function of the form $\sigma_n = f(x_{n-1}, x_{n-2}, x_{n-3}, \ldots)$. Usually we only consider a small finite window, that is, the number of arguments for the function $f()$ is small. The context information σ_n can be useful as a general form of prediction for x_n, where our objective is not to predict the value of x_n itself, but rather to characterize the distribution of x_n given σ_n. For example, if $\sigma_n = |x_{n-1}| + |x_{n-2}| + |x_{n-3}|$, i.e, is the sum of the magnitudes of three previously quantized coefficients, then small values of σ_n may correspond to a probability function for x_n that is highly peaked around zero (i.e. another small coefficient is likely), while larger values of σ_n may correspond to an almost uniform distribution for x_n (i.e. large magnitude wavelet coefficients will tend to appear in clusters). Note, that if each value of σ_n had to be assigned a different probability model, the system would quickly become impractical (excessive number of models as compared to the amount of data would result in context dilution.) Thus, in practice, a finite number of *classes* or *probability models* $\mathcal{N}$ is used so that a given x_n is assigned to one of these $\mathcal{N}$ classes depending on the context. Since the possible values of σ_n may be very large, selecting the $\mathcal{N}$ classes is similar to quantizing σ_n into $\mathcal{N}$ discrete values.

3.2.2 Context Modeling and low memory

We now describe a low memory context modeling approach that is well suited for our line-based transform, refer to Fig. 7. For each subband we keep only one line of context information, where the context contains both sign and magnitude information from previous lines. As a context we will use a *weighted average* of all previous lines of wavelet coefficients. This

[7] We assume that only causal contexts are used to avoid having to send any side information to the decoder.

approach is a good compromise that allow us to maintain low memory while at the same time being able to include in the context more information than that corresponding to the previous line. Let $\alpha_i, i = 0, \ldots K$ be the *quantized* wavelet coefficients in one line of a certain subband, where K is the width of a line. Let $c_i, i = 0, \ldots, K$ be the context information for magnitude in this subband. For each subband we keep one line of context information. We start at the top of the subband with all c_i equal to zero and update each c_i after we encode a pixel of value α_i as follows:

$$c_i = \begin{cases} |\alpha_i| & \alpha_i \neq 0 \\ c_i/2 & \text{otherwise} \end{cases} \tag{3}$$

The scanning within a subband is, left to right, top to bottom. If a coefficient is found to be non-zero (at the given quantization level), we keep its absolute value as the context information in the current position in a line. However if the current coefficient is zero we divide the previous context information by 2, so as to lower (but not set to zero) the neighborhood magnitude. The factor of 2 is chosen for simplicity of implementation, and there was no particular optimization involved. The above way of forming context information is equivalent to looking to several previous lines at a time and using the information from the nearest nonzero coefficient in the vertical direction. The advantage of our approach is that we are doing it in a computationally much more efficient way since we accumulate all context information in one line. Context information c_i is kept in fixed point format, so we expect to have $c_i = 0$ in areas having many zeros, i.e., a non zero coefficient does not "propagate" more than few lines or samples in the context information. Apart from the context related to the magnitude of past encoded coefficients we also keep context information for the sign of the coefficients from the previous line, a coefficient can either be positive "+", negative "-", or zero "0". Note that the sign we store here is that of the actual coefficient in the line in that position. We need 2 bits for each coefficient in order to keep this information, the context information for the sign is:

$$s_i = \text{sign}\{\alpha_i\} = \begin{cases} 0 & \text{if } \alpha_i = 0 \\ 1 & \text{if } \alpha_i > 0 \\ -1 & \text{if } \alpha_i < 0 \end{cases} \tag{4}$$

The total memory needed for storing the context information for all subbands is three equivalent image lines. We need 3 lines of length $X/2$ for the first level of decomposition, 3 lines of length $X/8$ for the second level, and in general 3 lines of length 2^{-n-1} for the n^{th} level. The total buffer size is $T_C = \sum_{i=0}^{N-1} 3 \cdot 2^{-n-1} X = 3(1 - 2^{-N})X$, which tends to $3X$ as N grows.

3.2.3 Classification and Encoding

In the rest of the document we will use a function $\texttt{encode}\{v \big| \zeta\}$, to represent encoding a number v given a discrete probability model ζ. $\texttt{encode}\{\}$ will use arithmetic coding and $\texttt{encode-raw}\{v^{[\xi]}\}$ will represent the function of sending the ξ least significant bits of v, without any form of entropy coding. For the cases where we are encoding a single bit we use the function $\texttt{encode-bit}\{\mu \big| \zeta\}$, in order to emphasize that we are encoding a single bit.

Based on the context information c_i (see Figure 7) we classify each new coefficient α_i into a class γ_i, among 15 possible classes, as follows:

$$\beta_i = c_i + 2c_{i-1} + c_{i+1} + (c_{i-3}||c_{i-2}||c_{i+2}||c_{i+3}) \tag{5}$$

$$\gamma_i = \begin{cases} 14 & \text{if} \quad \beta_i > 2^{14} - 1 \\ 0 & \text{if} \quad \beta_i = 0 \\ 1 + \lfloor \log_2 \beta_i \rfloor & \text{otherwise} \end{cases} \tag{6}$$

Where $||$ stands for logical "or",

$$\alpha||\beta = \begin{cases} 0 & \text{if} \quad \alpha = 0 \text{ and } \beta = 0 \\ 1 & \text{otherwise} \end{cases}$$

The motivation behind this scheme is to keep the contribution of $c_{i-3}, c_{i-2}, c_{i+2}, c_{i+3}$ to the context formation to a minimum. The selection of the $1 + \lfloor \log_2 . \rfloor$ operator, is mostly for simplicity since it represents the number of significant bits. Moreover using a logarithmic rule for quantization into classes also accounts for the fact that in typical images neighborhoods with small context magnitude (β_i small) are more likely than those with high magnitude. Thus a logarithmic rule allows us to have more classes at low magnitude than at high magnitude.

Class $\gamma_i = 0$ corresponds to a coefficient where all its neighbors are zero, so that we expect $|\alpha_i|$ to be very close to zero, for values of γ_i further away from zero the distribution of $|\alpha_i|$ is much less peaked around zero. Up to 15 classes can occur, but in practice the number of classes that are used depends up on the bit rate. If we happen to have large enough wavelet coefficients, or high bit rate, all 15 classes might be used, but if we are encoding at low bit rates only a portion of the 15 classes might occur. After classification we encode a bit denoting if our current coefficient α_i is significant or not, the encoding of this bit is conditioned upon the class γ_i, sending information denoting if our current coefficient α_i is significant or not corresponds to $\texttt{encode-bit}\{|\alpha_i| > 0 \big| \gamma_i\}$. If a given coefficient is found to be significant, we also encode a parameter l_i:

$$l_i = \lfloor \log_2 |\alpha_i| \rfloor \tag{7}$$

This parameter denotes the extra number of LSBs[8] needed to represent the magnitude of $|\alpha_i|$, given that $|\alpha_i| > 0$. We follow by a raw encoding of the l_i LSBs of $|\alpha_i|$. As an example, assume[9] $|\alpha_i| = 0000\overline{1}\underline{1101}$, $l_i = 4$. If the decoder knows the value of l_i it can then find the highest order nonzero bit of α_i, and with the transmission of l_i more bits, $|\alpha_i|$ will have been completely transmitted.

The sign of α_i is subsequently encoded using a separate context modeling, based on the sign of the two nearest neighbors. Each neighbor can be either zero, positive or negative so that we have a total of nine different combinations. By using a technique know as *sign flipping* [20] we can reduce the number of classes from nine to five. In sign flipping we take advantage of certain symmetries present in the context formation, as seen in Table 1. Let us consider an example. Assume that both neighbors g_o, g_1 are positive and let $p, 1-p$ be the probability that a new coefficient x has the same sign or not, respectively, as its neighbors. We can assume that the probability of "same sign" will be the same if both g_o, g_1 are negative and thus we only characterize the probability of having a sign change and assume these are roughly the same regardless of the sign g_o, g_1 have, as long as their sign is the same.

3.2.4 State information/ Arithmetic Coder

We use an arithmetic coder that operates with different probability models where the probability model depends on the class γ_i. For each class, we keep track of symbol occurrences so as to update the probability model. The models needed for each band are:

- Five binary models for sign encoding
- 15 binary models for encoding significance information,where each model corresponds to a class γ
- One model $\mathcal{B}$ of up to 14 symbols for encoding the number l_i from (7). This model will be used as an M-ary model with $0 < M \leq 14$ by considering the first M symbols. As explained in section 3.2.5, l_i will always be bounded by a number M for each line.

Therefore we will need $5+15+14 = 34$ words per subband in order to store the probability model information. As can be deduced from the foregoing discussion, context modeling does not represent a significant memory overhead to our system as compared to the memory needed for filtering.

[8] Least significant bits

[9] The over-lined bit is the highest order nonzero bit, while the underlined bits are the additional l_i LSBs that will be sent to the decoder.

3.2.5 Algorithm

After wavelet transform and quantization, with the same dead-zone quantizer for all the coefficients, the algorithm for encoding a line corresponding to one subband can be described as follows:

1. For each new line `encode-raw`$\{L^{[15]}\}$, where $L = \lfloor \log_2(1 + \max_i |\alpha_i|) \rfloor$ is the number of bits needed to represent the largest coefficient in a row line. If $L > 0$ go to step 2 else return to step 1. This allow us to determine the maximum value for l_i and to skip lines that are all zero.

2. Classify each new wavelet coefficient α_i into a class γ_i according to equations (5) and (6).

3. `encode-bit`$\{|\alpha_i| > 0 \big| \gamma_i\}$ (Encode whether α_i is zero or not, by using the corresponding model γ_i) and update the statistics for model γ_i.

4. if $|\alpha_i| > 0$ $\Big\{$

   - `encode`$\{l_i \big| \mathcal{B}^{(M)}\}$ where $l_i = \lfloor \log_2 |\alpha_i| \rfloor$, $M = \lceil \log_2 L \rceil$, $\mathcal{B}^{(M)}$ means that we are using model $\mathcal{B}$ as an $M-$ary model, since we know that $l_i < M$.
   - form a class w_i for the encoding of the sign according to table 1
   - `encode-bit`$\{\alpha_i > 0 \big| w_i\}$ and update the statistics for model w_i.
   - `encode-raw`$\{|\alpha_i|^{[l_i]}\}$, that is we encode the l_i LSBs of $|\alpha_i|$.

   $\Big\}$ else go to step 5

5. update the context information according to equation (3)

6. if not end of line go to the next pixel (step 2)

7. if not end of image go to the next line (step 1)

4 Experimental Results and Memory Analysis

In Table 2 we present PSNR results for five different images along with comparisons with algorithms in [5, 9, 11] and also JPEG [2] with arithmetic coding[10]. Our results are not always

[10] In order to provide a fair comparison with a DCT based technique we select the arithmetic coding based JPEG (JPEG-AR) rather than baseline JPEG. The memory requirements are very similar and the compression performance better for JPEG-AR. All the wavelet coders we consider use arithmetic coding.

the best but are competitive at a fraction of the complexity and memory utilization. It is worth repeating that our algorithm is one pass and there are no rate distortion optimization decisions made at any stage. The memory requirements depend upon the filter length in our filter-bank, the number of levels in the decomposition, the type of decomposition itself, and the width of the image. The height of the image does not affect the memory needs. The transform can be implemented in only $2LX$ pixels of memory independently of the number of levels of decomposition, the encoder and decoder are responsible for handling the synchronization buffers. Even though the synchronization buffers grow exponentially with the number of decomposition levels, we can bring their size down by orders of magnitude if we keep them in compressed form. The memory needed for context modeling is $3X$ pixels. The overall memory needs for our algorithm are the lowest reported in the literature for wavelet coders.

In Table 3 we present the exact memory usage for all algorithms [9, 5, 11, 2], as measured in an HP-Kayak workstation running windows NT, the memory needs of our algorithm are much closer to JPEG than any of the above mentioned algorithms. The scaling problems of wavelet coders can be seen clearly by observing the numbers in table 3. In many practical applications images of hundreds of millions of pixels need to be compressed, but in many cases it is impossible to buffer the whole image. Tiling the image causes blocking artifacts and degrades the performance. In table 2 for the column corresponding to [11] we forced the algorithm to work with tiles of size 128×128, this configuration corresponds to memory needs slightly above our current algorithm, but the performance of the wavelet coder in this case falls below that of JPEG-AR.

It is worth pointing out that we do not use a bit plane coder for the refinement bits as for example [11]. Instead we are encoding each quantized wavelet coefficient at once, without the need for multiple passes. The results, as compared to bit plane coders [5, 11, 4], are still very competitive. We compensate for not entropy coding some bits by the use of a higher order arithmetic coder to encode the number l_i of those bits.

5 Contributions

In this paper we have developed a technique for line based wavelet transforms, analyzed the memory needs of the transform and separated the memory needed in two different categories, namely *filtering* memory and *synchronization* memory. We pointed out that the synchronization memory can be assigned to the encoder or the decoder and that it can hold compressed

data. We provided an analysis for the case where both encoder and decoder are symmetric in terms of memory needs and complexity. We described a novel entropy coding algorithm that can work with very low memory in combination with the line-based transform, and showed that its performance can be competitive with state of the art image coders, at a fraction of their memory utilization. The whole system can also be used as a tool for WT implementation in low memory, and in fact line based transforms as implemented in this work have been incorporated into the JPEG 2000 verification model. It is also worth mentioning that we can use compression to deal with the memory growth of the synchronization buffers, even in applications where compression is not the main objective, but where memory is an issue.

To the best of our knowledge, our work is the first to propose a detailed implementation of a low memory wavelet image coder. It offers a significant advantage by making a wavelet coder attractive both in terms of speed and memory needs. Further improvements of our system especially in terms of speed can be achieved by introducing a lattice factorization of the wavelet kernel or by using the lifting steps. This will reduce the computational complexity and complement the memory reductions mentioned in this work.

References

[1] C. Chrysafis and A. Ortega, "Line Based Reduced Memory Wavelet Image Compression," in *Proc. IEEE Data Compression Conference*, (Snowbird, Utah), pp. 308–407, 1998.

[2] W. Pennebaker and J. Mitchell, *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, 1994.

[3] D. Lee, "New work item proposal: JPEG 2000 image coding system." ISO/IEC JTC1/SC29/WG1 N390, 1996.

[4] J. M. Shapiro, "Embedded Image Coding Using Zerotrees of Wavelet Coefficients," *IEEE Trans. Signal Processing*, vol. 41, pp. 3445–3462, December 1993.

[5] A. Said and W. Pearlman, "A New Fast and Efficient Image Coder Based on Set Partitioning on Hierarchical Trees," *IEEE Trans. Circuits and Systems for Video Technology*, vol. 6, pp. 243–250, June 1996.

[6] Z. Xiong, K. Ramchandran, and M. Orchard, "Space-frequency quantization for wavelet image coding," *IEEE Trans. on Image Proc.*, vol. 6, pp. 677–693, May 1997.

[7] R. L. Joshi, H. Jafarkhani, J. H. Kasner, T. R. Fischer, N. Farvardin, M. W. Marcellin, and R. H. Bamberger, "Comparison of different methods of classification in subband coding of images," *IEEE Trans. on Image Proc.*, vol. 6, pp. 1473–1486, Nov. 1997.

[8] Y. Yoo, A. Ortega, and B. Yu, "Image subband coding using progressive classification and adaptive quantization," *IEEE Trans. on Image Proc.*, Jun. 1997. Submitted.

[9] C. Chrysafis and A. Ortega, "Efficient Context-based Entropy Coding for Lossy Wavelet Image Compression," in *Proc. IEEE Data Compression Conference*, (Snowbird, Utah), pp. 241–250, 1997.

[10] S. M. LoPresto, K. Ramchandran, and M. T. Orchard, "Image Coding based on Mixture Modeling of Wavelet Coefficients and a Fast Estimation-Quantization Framework," in *Proc. IEEE Data Compression Conference*, (Snowbird, Utah), pp. 221–230, IEEE Computer Society Press, Los Alamitos, California, 1997.

[11] C. Christopoulos (Editor), "JPEG 2000 Verification Model Version 2.1," *ISO/IEC JTC/SC29/WG1*, June 1998.

[12] M. Vishwanath, "The recursive pyramid algorithm for the discrete wavelet transform ," *IEEE Trans. Signal Processing*, vol. 42, pp. 673–676, March 1994.

[13] C. Chakrabarti and C. Mumford., "Efficient realizations of encoders and decoders based on the 2-D Discrete Wavelet Transform ," *accepted for publication in the IEEE Transactions on VLSI Systems*, 1998.

[14] T. Denk and K. Parhi, "LSI Architectures for Lattice Structure Based Orthonormal Discrete Wavelet Transforms," *IEEE Trans. Circuits and Systems*, vol. 44, pp. 129–132, February 1997.

[15] "Analog Devices", "Low Cost Multiformat Video Codec, ADC601," *Manual*, 1997.

[16] P. Cosman and K. Zeger, "Memory Constrained Wavelet-Based Image Coding," in *Presented at the First Annual UCSD Conference on Wireless Communications*, March 1998.

[17] P. Cosman and K. Zeger, "Memory Constrained Wavelet-Based Image Coding," *Signal Processing Letters*, vol. 5, pp. 221–223, September 1998.

[18] K. Ramchandran and M. Vetterli, "Best Wavelet Packet Bases in a Rate–Distortion Sense," *IEEE Trans. on Image Proc.*, vol. 2, pp. 160–175, April 1993.

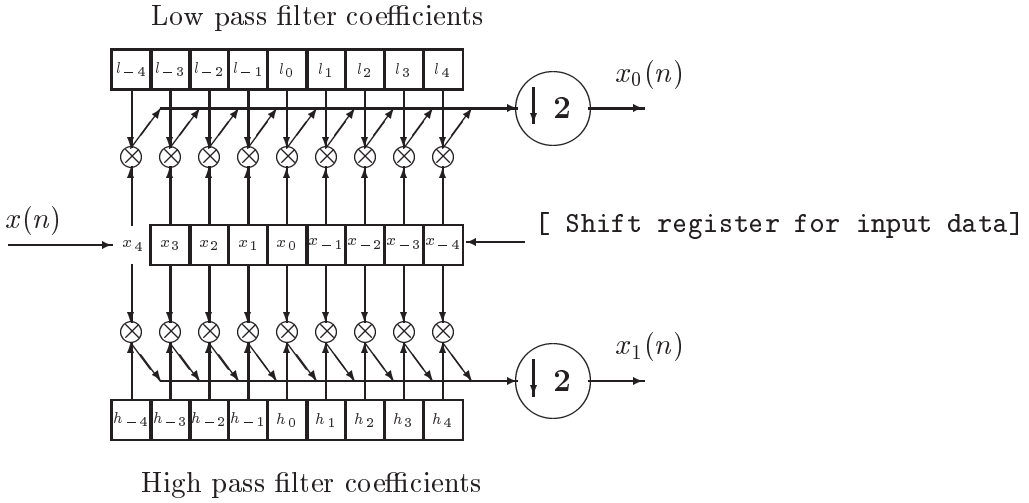


Figure 1: One level decomposition for filters of length $L = 2S+1 = 9$. We need eight memory elements, which are represented by a shift register. The delay between input and output is $S = 4$ samples.

[19] D. Taubman, "Embedded independent block-based coding of subband data," `Hewlett Packard`, *ISO/IEC JTC/SC29/WG1N871 Document, Copenhagen*, June 1998.

[20] X. Wu, "Lossless Compression of Continuous-tone Images via Contex Selection, Quantization and Modeling," *IEEE Trans. Image Processing*, vol. 6, pp. 656–664, May 1997.

[21] "ISO/IEC JTC1/SC29/WG1 (ITU-T SG8)", "Coding of Still Pictures, JBIG, JPEG. Lossless, nearly lossless Compression, WD14495," tech. rep., Working Draft, June 1996.

[22] M. Weinberger, G. Seroussi, and G. Sapiro, "Loco-I: A low complexity, context-based, lossless image compression algorithm," in *Proc. IEEE Data Compression Conference*, (Snowbird, Utah), pp. 140–149, IEEE Computer Society Press, Los Alamitos, California, 1996.

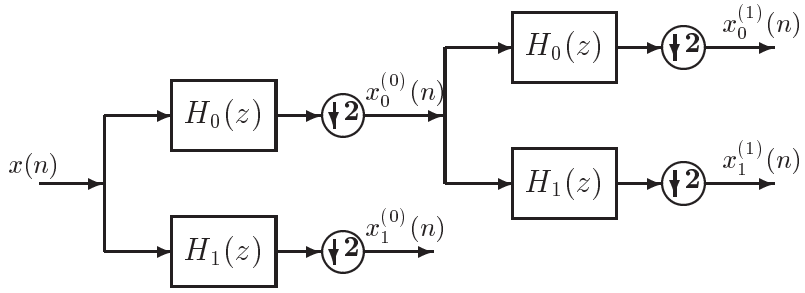


Figure 2: Cascade of two levels of wavelet decomposition. The delay for the first level is S, the delay for the second level is $2S$ and the total delay for both levels is $S + 2S$. The time units for the delay are considered in the input sampling rate.

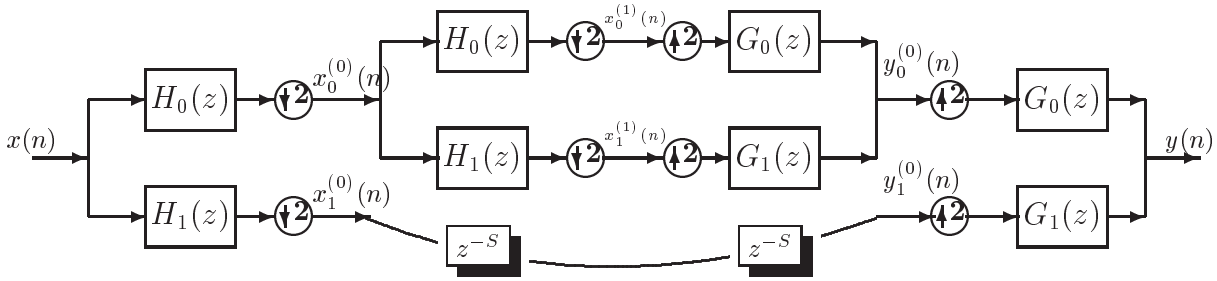


Figure 3: Consideration of both synthesis and analysis filter banks reveals the need for syncronization buffers. Two buffers z^{-S} are needed to form a queue for the $x^{(0)}(n)$ samples.

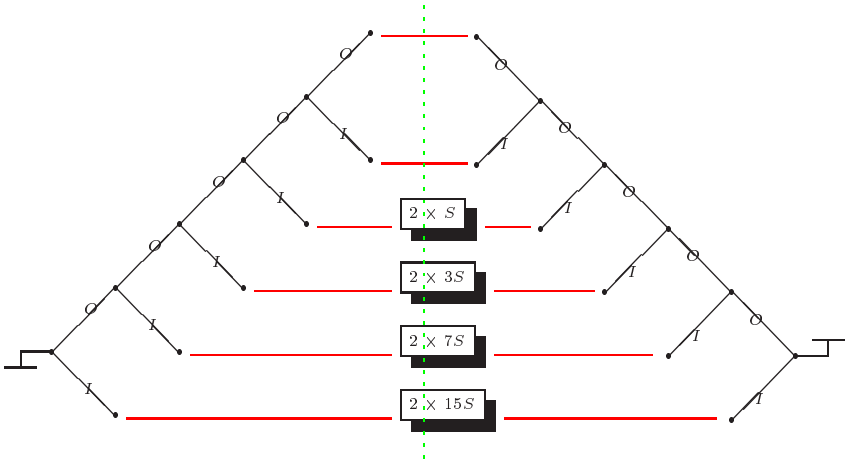
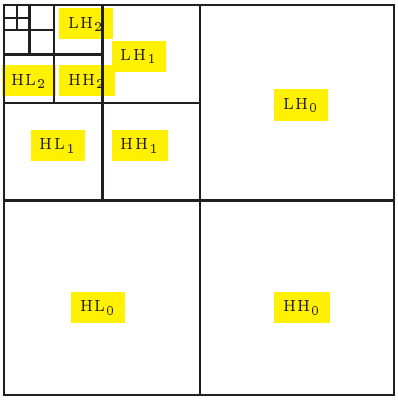
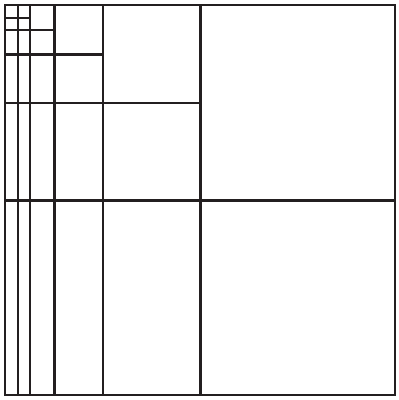


Figure 4: One dimensional analysis and synthesis decomposition trees, the delay involved in the analysis/synthesis filter banks is depicted, the memory needed increases exponentially with the number of levels. The major factor for memory is the syncronization buffers and not the filtering buffers.



(a) Dyadic tree decomposition



(b) "Hybrid" decomposition

Figure 5: (a) Five level dyadic tree decomposition. The multiplications needed for the whole decomposition are $\frac{8}{3}XYL$. (b) Five levels decomposition in the horizontal direction for all lines, followed by a dyadic decomposition in the vertical direction. The memory needs are the same as in the previous case, the multiplications needed are $\frac{10}{3}XYL$

class	sign flip	g_1	g_0
0	NO	0	0
1	NO	+	0
	YES	-	0
2	NO	0	-
	YES	0	+
3	NO	-	-
	YES	+	+
4	NO	+	-
	YES	-	+

		g_0	
	g_1	x	

Table 1: Context formation for sign encoding/decoding. We only use the sign from the two nearest neighbors for context formation. We exploit symmetry by using a sign flipping technique and we thus reduce the number of classes from nine to five. g_0, g_1 are the signs of the wavelet coefficients at the corresponding locations.

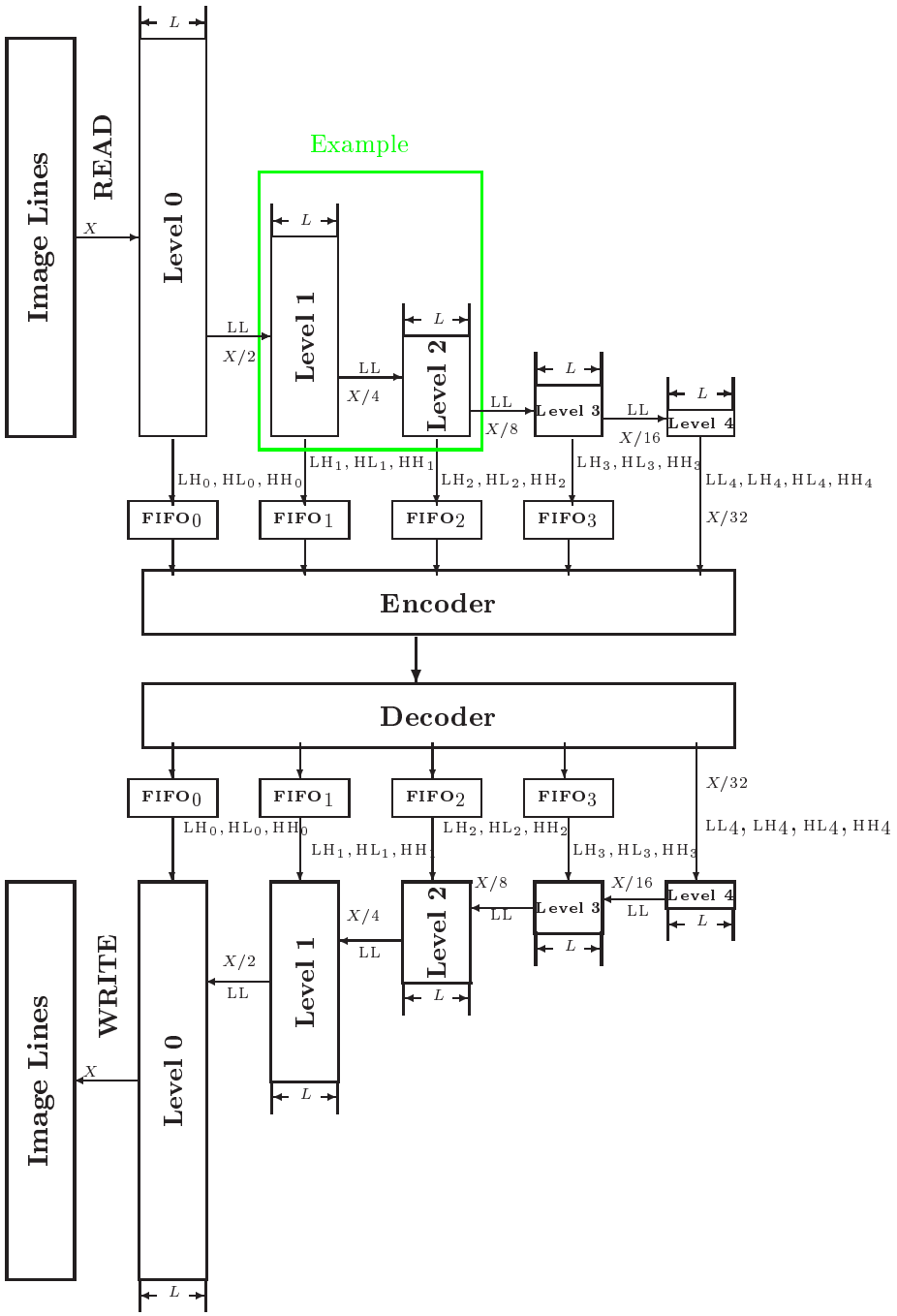


Figure 6: Full system with analysis filter bank, encoder, decoder and synthesis filter bank, we consider 5 levels of decomposition. The FIFO buffers can become part of the encoder and decoder, in order to reduce the total memory size. Encoder and decoder need to be able to work with each band independent of the others. The filtering buffer for level n consists of L lines of length $2^{-n}X$. The data flow is as follows: we read image data, pass through the filtering blocks for the appropriate levels, send data to the encoder and inverse the process in order to reconstruct the image.

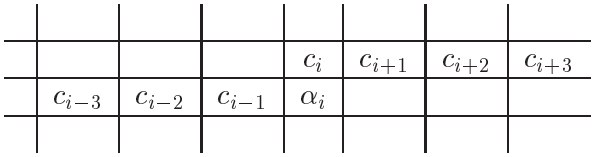


Figure 7: Context information for magnitude encoding. For each subband we keep a line of context information. In this Figure we see the relative position of the wavelet coefficient α_i to be encoded and the context information around it.

	Rate	SPIHT[5]	C/B[9]	JPEG-AR	VM2.0[11]	This work
Lena	0.125	31.10	31.32	28.45	30.93,(27.96)	31.05
512 × 512	0.25	34.13	34.45	31.96	34.03,(31.36)	34.20
	0.50	37.24	37.60	35.51	37.16,(34.75)	37.35
	1.00	40.45	40.86	38.78	40.36,(38.60)	40.47
Barbara	0.125	24.84	25.39	23.69	24.87,(23.27)	25.20
512 × 512	0.25	27.57	28.32	26.42	28.17,(25.38)	28.18
	0.50	31.39	32.29	30.53	31.82,(29.20)	31.87
	1.00	36.41	37.40	35.60	36.96,(33.79)	36.68
Goldhill	0.125	28.47	28.61	27.25	28.48,(26.84)	28.49
512 × 512	0.25	30.55	30.75	29.47	30.58,(29.21)	30.64
	0.50	33.12	33.45	32.12	33.27,(31.88)	33.27
	1.00	36.54	36.95	35.57	36.81,(35.47)	36.66
Bike	0.125	25.82	26.16	24.88	25.75,(21.89)	25.92
2560 × 2048	0.25	29.12	29.43	28.20	29.30,(24.83)	29.17
	0.50	33.00	33.47	32.11	33.28,(29.30)	33.04
	1.00	37.69	38.27	36.39	38.08,(34.39)	37.66
Woman	0.125	27.27	27.67	26.05	27.23,(24.09)	27.51
2560 × 2048	0.25	29.89	30.36	28.83	29.79,(26.12)	30.14
	0.50	33.54	34.12	32.47	33.54,(28.80)	33.74
	1.00	38.24	38.92	37.11	38.30,(32.96)	38.47

Table 2: Comparison between our method and [5, 9, 2, 11] for images: Barbara, Lena, Goldhill, Bike and Woman, the last two images are part of the test images for JPEG2000. We used five levels dyadic decomposition with 9-7 tap filters.(JPEG-AR stands for JPEG compression with the addition of arithmetic coding.) For algorithm [11] the numbers in parenthesis correspond to tiles of size 128×128.

Image Size	compressed	SPIHT[5]	C/B[9]	JPEG[2]	VM2.0[11]	Line Based
5.2M 2560×2048	650K	27M	21M	688K	51M	(1.5M) 850K
16.9M 3312×5120	2.1M	81M	67M	720K	97M	(3.4M) 1.3M
33.9M 6624×5120	4.2M	*	92M	720K	*	(5.5M) 1.3M

Table 3: Memory usage for algorithms [5, 9, 2, 11] for tree different image sizes 5.2, 16.9 and 33.9 Mbytes. All images were compressed at 1b/p. The numbers were obtained using an HP Kayak-XU workstation with a 300MHz Pentium II processor running windows NT, with 128M of memory. The numbers in parenthesis for the line based algorithm correspond to the memory needed for the algorithm plus memory for buffering of the complete bit stream. The numbers were measured for the decoder but for all the above algorithms encoder and decoder are symmetric in terms of memory. The "*" corresponds to cases where the memory needs exceeded the machines limitations .