Implementations of the discrete wavelet transform: complexity, memory, and parallelization issues

Antonio Ortega, Wenqing Jiang, Paul Fernandez
Integrated Media Systems Center
Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089-2564
Email: ortega,wqjiang,pfernand@sipi.usc.edu

Christos Chrysafis
Hewlett-Packard Labs
1501 Page Mill Road, Bldg. 3U
Palo Alto, CA 94303-1126
Email: chrysafi@hpl.hp.com

ABSTRACT

The discrete wavelet transform (DWT) has been touted as a very effective tool in many signal processing applications, including compression, denoising and modulation. For example, the forthcoming JPEG 2000 image compression standard will be based on the DWT. However, in order for the DWT to achieve the popularity of other more established techniques (e.g., the DCT in compression) a substantial effort is necessary in order to solve some of the related implementation issues. Specific issues of interest include memory utilization, computation complexity and scalability. In this paper we concentrate on wavelet-based image compression and provide examples, based on our recent work, of how these implementation issues can be addressed in three different environments, namely, memory constrained applications, software-only encoding/decoding, and parallel computing engines. Specifically we will discuss (i) a low memory image coding algorithm that employs a line-based transform, (ii) a technique to exploit the sparseness of non-zero wavelet coefficients in a software-only image decoder, and (iii) parallel implementation techniques that take full advantage of lifting filterbank factorizations.

Keywords: Discrete Wavelet Transform, Low Memory Implementation, Software Implementation, Parallel Architecture, Finite State Machine, Boundary Postprocessing, Split-and-Merge

1. INTRODUCTION

The discrete wavelet transform (DWT) has quickly become a very popular tool in a number of digital signal processing applications. Filterbanks, such as those used to compute the DWT, have been an active research topic since the early 1980s. Their renewed popularity for applications such as compression or recognition, as well as the establishment of the link between the theory of filterbanks and that of wavelets, have contributed to increase the interest in the implementation of these filterbanks. Whether cost (e.g., memory) or speed is the main consideration, one key obstacle to the full blown deployment of wavelet technology has been the relative lack of study of DWT implementation issues, especially as compared to well established transform techniques, such as the discrete cosine transform (DCT) and the Fast Fourier Transform (FFT).

This paper provides an overview of some of our recent research on the implementation of the DWT. Here, our goal is to maximize performance according to an appropriate metric, such as computation time or memory utilization. The relevant parameter of interest will depend on the application chosen, and may not be the same for a digital camera, high-end scientific computation in a network of workstations or the decoding of a wavelet coded video stream in standard PC. We will concentrate on image processing applications, and in particular compression, as the DWT is particularly challenging in these situations, given the potentially large volumes of data to be handled. Given that the soon to be finalized JPEG 2000 standard will use the DWT, compression is likely to be one of the major applications for wavelet technology.

We will consider three case studies to illustrate the difficulties in wavelet transform implementation, as well as some of the potential solutions available. First, in Section 2, we consider the issue of memory efficiency, which is of particular importance when considering applications such as digital cameras. Our results show the lowest memory requirements reported in the literature and are competitive with state of the art algorithms (that have much higher memory requirements) in terms of compression performance. Then, in Section 3, we discuss some implementation issues that are specific of software environments. In particular, when fast software decoding is required in a low-end platform it is useful to consider techniques that minimize the number of computations, in the average case, by performing input-dependent computation. We will show how it is possible to speed up the computation of the inverse DWT (IDWT) by avoiding computations in cases when the wavelet coefficients are zero. Finally, in Section 4 we study the parallel implementation of a wavelet transform. Parallel implementations are likely to be prefered in large scale computations or in hardware platforms (e.g. FPGAs) that provide built-in parallelism. We show that significant performance gains can be achieved by taking advantage of the structure of the wavelet filters. In particular it is shown that for certain lifting factorizations it is possible to reduce significantly the number of times processors have to communicate with each other.

2. LOW MEMORY 2D WAVELET IMAGE CODING

Memory costs are an important consideration in applications such as compression for digital cameras, where they can be a major component of the cost. In these low memory applications, existing algorithms based on the Discrete Cosine Transform (DCT) (e.g., the JPEG standard [1]) have proven to be popular, as they allow block by block processing. Instead, wavelet-based coders, such as those being considered within the JPEG 2000 standardization process [2], often require orders of magnitude more memory than standard JPEG-based techniques. Indeed, improving the memory efficiency is currently one area of major research activity within the JPEG2000 standardization process [2].

Algorithms such as those in [3–10], are representative of the state of the art in wavelet image coders. All of these algorithms assume that the DWT for the whole image has been computed so that all the corresponding coefficients are available in the coding process. Global image information[*] is used in various ways including, among others, classification (e.g., [6]), initialization of the probability models used by a quantizer or arithmetic coder [7–9] or selection of specific decompositions of the signal [5]. It is also worth noting that even if no global information has to be measured, algorithms that provide progressive transmission [3,4] may require storing all wavelet coefficients, since they are coded using a layered coding approach.

In this section, we will first describe an approach, the so-called line-based wavelet transform, which can compute a separable 2D transform with the least amount of memory, and produce the same result as a standard transform. We will then provide experimental evidence that combining such a transform with a low memory coding algorithm produces results comparable to the best in the literature, at a fraction of the memory utilization. While other authors have approached low memory transforms (e.g., [11]), or encoding for low memory decoding (e.g., [12,13]), the work we describe here is unique in that it addresses memory at the encoder, and presents a complete low memory coding system. We refer to [14–16] for more detailed descriptions of the algorithms.

2.1. Line-based Wavelet Transforms

Low memory implementations of 1D wavelet transforms were first addressed in [11]. Our proposed line-based approach essentially extends the work in [11] by considering 2D data *and* taking into account buffer requirements for synchronization between encoder and decoder. Since it is very common to acquire image data line by line (e.g., through a scanner), we will focus our discussion on algorithms that require the smallest number lines to perform the 2D DWT. The straightforward approach to compute the 2D DWT would consist of performing line filtering and then vertical filtering. However, this would require storing the output generated by filtering each input line, and therefore a memory size of the order of the image size is required. Instead, we are interested in minimizing the number of lines that need to be kept in memory. We assume that as soon as these coefficients are computed they can be processed (e.g., compressed) and removed from main memory (e.g., transmitted.)

The obvious alternative to completing row filtering before starting column filtering 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.

2.1.1. 1D DWT: Delay and Synchronization Filters

Let us consider first an implementation of a 1D DWT, where the system receives data sequentially, one pixel at a time. Without loss of generality, consider the case where the longest filter in the filterbank has odd length equal to $L = 2S + 1$. Consider first a single stage of the DWT. At time zero we start receiving data and store it in the filter memory (a shift register). At time S, with appropriate extensions, we have received enough data to fill the entire

[*]i.e., information that can only be obtained after the whole image has been transformed. Examples include the largest coefficient magnitude, the energy or the coefficient histograms for each subband.

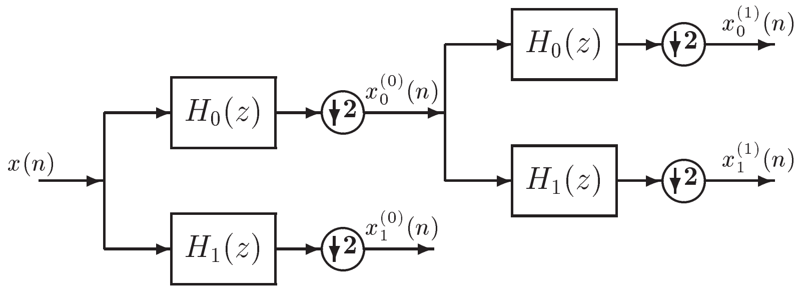

Figure 1. Cascade of two levels of wavelet decomposition. The delay for the first level is S, the delay for the second level is $2S$ and the total delay for both levels is $S + 2S$. The time units for the delay are considered in the input sampling rate.

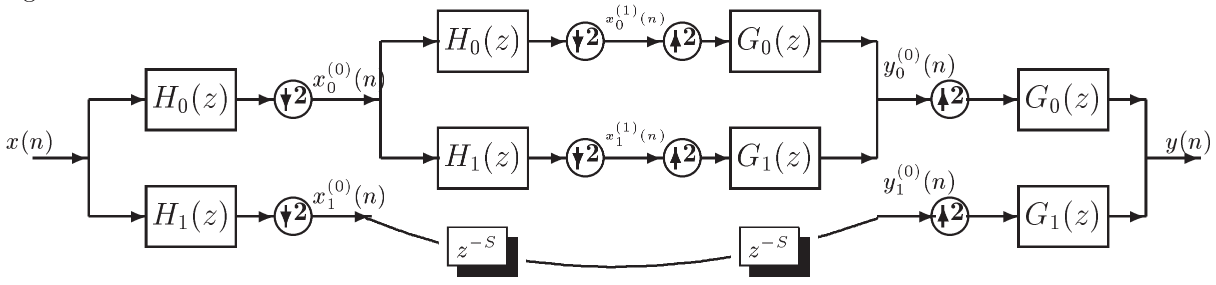

Figure 2. When both synthesis and analysis filter banks are considered two syncronization buffers, z^{-S}, are needed to delay the $x^{(0)}(n)$ samples.

input buffer ($S+1$ samples received, and the S samples duplicated to generate the symmetric extension.) Thus, the *delay* for generating output coefficients for one level of decomposition is S.

Consider the two-level decomposition of Figure 1. 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 Figure 1 is $S + 2S$ input samples.

This analysis can be easily extended to the case of an N-level decomposition. Let the decomposition levels be indexed from 0 to $N-1$, where 0 corresponds to the first level. 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 input to output of an N level filterbank will be $D_N = S + 2S + 2^2S + 2^3S + \ldots + 2^{N-1}S = \sum_{k=0}^{N-1} 2^k S = (2^N - 1)S$. 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[†] 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.

[†]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 $S+1$.

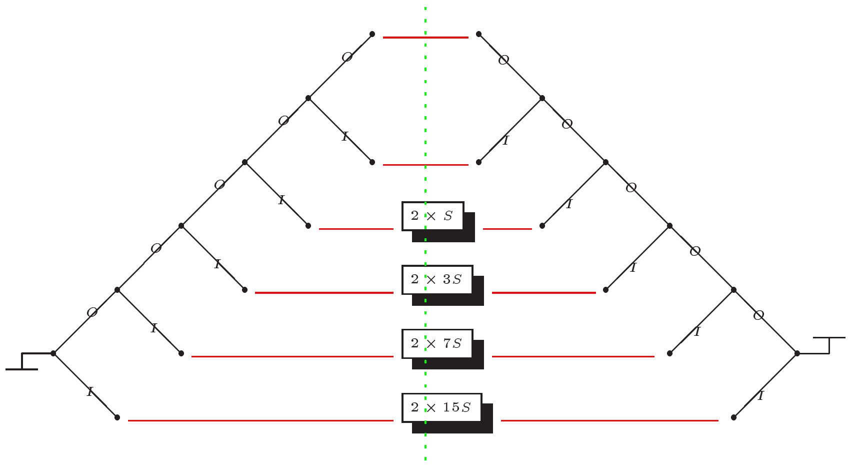

Figure 3. One dimensional analysis and synthesis decomposition trees. As shown the memory needed for the synchronization buffers increases exponentially with the number of levels.

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. 1 and 2, 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 2). The required memory can be split into two buffers of size S, 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 3.

In summary, memory is needed for both *filtering* and *synchronization*, with buffers of size L, and $D_{N-n} = (2^{N-n} - 1)S$, respectively, needed for the n^{th} level of decomposition. Therefore the total memory size needed for N levels of decomposition in a symmetric system is: $T_{total,N} = (2^N - N - 1)S + NL$, where it can be seen that synchronization buffers can become very large as the number of decomposition levels grows.

2.1.2. 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 DWT 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. We consider here the dyadic decomposition where only LL bands at each level of decomposition are further decomposed. In order to implement a one level decomposition vertically we need to buffer L lines. 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, since in the dyadic composition the second level

[‡]Here we assumed the synchronization buffers are split equally between analysis and synthesis, but synchronization buffers can also be easily assigned to either the analysis or the synthesis filterbanks if it is necessary to make one more memory-efficient than the other.

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. 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.

As in the 1D case, synchronization buffers are needed. For example in a two-level decomposition we will need to store the $\mathrm{HL}_0, \mathrm{LH}_0, \mathrm{HH}_0$ bands[§] 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. In an N-level decomposition the synchronization delay required for data at level n is $D_{N-n} = (2^{N-n}-1)S$ lines[¶], 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. 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 $3 \cdot (2^{N-n}-1)S \cdot 2^{-n-1}X$ coefficients and the total memory needed for synchronization will be:

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

From (1) we see that the size of the synchronization 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 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. This is the approach we use in our system to limit the memory required for synchronization.

In summary, for an N-level dyadic decomposition, we will need filtering buffers for up to $2LX$ pixels, and synchronization buffers for $T_N^{(2d)} = (2 \cdot 2^N + 2^{-N} - 3)XS$ coefficients.

2.2. Low memory DWT compression results

In order to take advantage of this low-memory line-based approach, and achieve a complete low memory encoding/decoding, it is necessary to define a compression algorithm that can compress and transmit coefficients as soon as these are generated. Therefore coding should be performed on the fly, without the need for any global information.

It should be noted that if providing an embedded bit stream is required it will be necessary 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 [17]). In either case, an embedded output requires more buffering than our proposed approach.

In our approach we use context modeling and classification to provide the probability model to be used by an arithmetic coder. Processing is based on the lines of wavelet coefficients produced by the DWT. Each band is encoded *separately*, and no cross-band information, or global information is used. We have demonstrated in [14–16] 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 [8,4,10,3,9]. Note that using backward adaptive algorithms is particularly useful in our case since other approaches, e.g., those based on explicit classification, would require us to store more wavelet coefficients. The overall memory requirements for context modeling are modest compared to the memory needed for filtering.

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

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

	Rate	SPIHT[4]	C/B[8]	JPEG-AR	VM2.0[10]	Line Based [15,16]
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 1. Comparison between our method [15,16] and [4,8,1,10] 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 [10] the numbers in parenthesis correspond to tiles of size 128 × 128.

Image Size	compressed	SPIHT[4]	C/B[8]	JPEG[1]	VM2.0[10]	Line Based [15,16]
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 2. Memory usage for the algorithms in [4,8,1,10], for tree different image sizes 5.2, 16.9 and 33.9 Mbytes. All images were compressed at 1b/p. Results 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 .

Table 1 shows PSNR comparisons with the algorithms in [4,8,10] and also JPEG [1] with arithmetic coding[‖]. Our results are competitive at a fraction of the complexity and memory utilization. Table 2 presents the exact memory usage for all algorithms [8,4,10,1], 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 in Table 2, where the memory needs increase in proportion to the image size. While it is possible to tile the images in order to reduce the memory needs, this comes at the cost of lower performance. For example, in Table 1 the results corresponding to [10] are

[‖]We use the higher performance arithmetic coding based JPEG (JPEG-AR) instead of baseline JPEG. Memory requirements are similar for both JPEG algorithms and a fair performance is possible since all wavelet coding schemes considered also use arithmetic coding.

based on using tiles of size 128 × 128, resulting in memory needs slightly above those of our algorithm, but with PSNR performance lower than JPEG-AR.

3. INVERSE WAVELET TRANSFORMS OF QUANTIZED DATA

Line-based approaches such as those described above have also advantages in terms of computation speed. While the number of operations is the same, a low memory algorithm is more likely to allow processing to be performed with more efficient use of the processor memory cache (i.e., each set of pixels may only have to be read once into the cache). Thus the above described system is also the fastest when operating with large images. If a wavelet based encoder is to be implemented on a software platform further speed-ups are possible. In particular this section focuses on a fast algorithm for computing the Inverse Discrete Wavelet Transform (IDWT), which takes advantage of the sparseness of the wavelet coefficients after quantization. The key idea here is to design an IDWT algorithm that is optimized for the average case, i.e., such that the computation time will depend on the input, but lower on average than for a fixed computation version of the algorithm. Our results show average case reductions in computation time of about 20% to 50% for PSNRs in the range 35dB to 29dB.

3.1. Motivation

Efficient implementations of the DCT have been widely studied. A recent trend, due to the wide use of software-based image coding and decoding schemes, has been to use input dependent implementations. For example to minimize the complexity of the inverse DCT can be reduced by not processing zero transform coefficients (e.g., [18,19]). The same idea can be applied to wavelet coding, where, as shown in Fig. 4, after quantization only a few wavelet coefficients are non-zero. As an example, in the HL subband of the first level of decomposition about 50% of zero streams have length 256, i.e., half the lines are all zero.

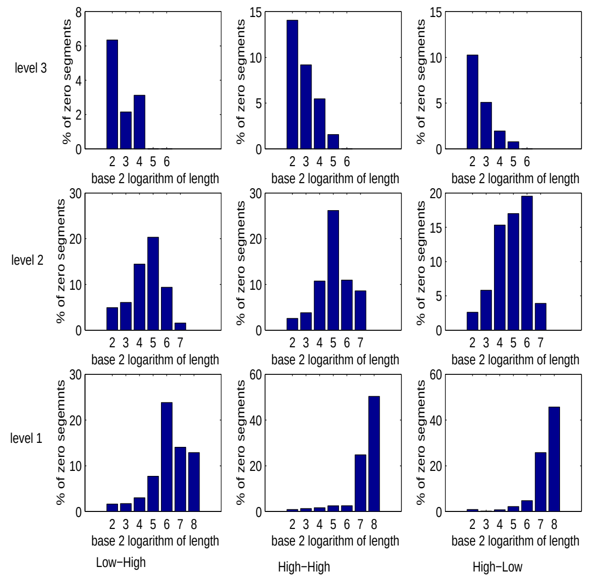

Figure 4. Histograms of zero streams. The x-axis is the logarithm of the length of zero streams. The y-axis is the percentage of each of the zero streams (Number of zero streams of a given length divided by the total possible number of such non-overlapping streams.) Results are for a 512x512 image using 3 levels of decomposition and quantization (the corresponding PSNR is 32dB.)

Related work in [20] has taken advantage of this sparseness in the context of progressively coded subbands. Here each level of resolution consists of a bitplane. Multiplications are avoided and the IDWT operation becomes a set of additions and right shifts. The reconstruction is done coefficient by coefficient. Instead, our work [21] is not based on bitplane coding and assumes that the quantized wavelet coefficients are known with full resolution.

3.2. Basic Zero Testing Algorithm

The basic approach to speed up the computation is to locate zero-valued wavelet coefficients and to skip the filtering operations on those. This is not straightforward for several reasons. First, while it may be possible to test each coefficient (and thus skip operation on all those coefficients that are zero) the cost of testing in this case would be too large and therefore there will be no savings in overall complexity (the additional testing cost would be more than the filtering savings.) Second, filtering is a "running", rather than block-based, operation and thus from zero positions we will need to deduce which filtering operations can be skipped given that a running convolution is computed.

Thus we propose a systematic way of testing for streams of zero coefficients. A significance map of the coefficients is formed when decoding, where each coefficient is represented by one bit, that is set to zero if the corresponding coefficient is zero. In our experiments, since column-filtering is done first, the bitmap is formed column-wise. Zero testing is then carried out on streams of coefficients before filtering. The zero testing is done in a tree-like manner - the top of the tree representing testing for zero streams of a certain length and progressively lower levels of the tree corresponding to testing for zero steams of progressively shorter lengths. A method to optimize the testing, which uses the statistics of typical image coefficients, is also proposed.

A column of wavelet coefficients is processed by first zero testing the corresponding column in the bit-map and then performing filtering if necessary. Figure 5 illustrates the testing method for a stream of 32 coefficients. Essentially, we try to determine which sets of coefficients are zero, first testing the 32 coefficients, then each subset of 16 coefficients, and so on. Filtering is then performed except for those sets of coefficients that have been found to be zero.

The key question, in order to minimize the overall computation, is to determine what are the appropriate root and leaf sizes to be used in a tree such as that of Figure 5. A deeper tree search would have larger overhead of zero testing coefficient steams. This overhead will be offset if there are a significant number of streams of zero coefficients of shorter length. We can find what the best root and leaf sizes are by considering the statistics of the data. Consider filtering a column of N coefficients. We could test if all N coefficients are zero and then perform filtering if needed. This would correspond to a one-level tree search. The overall complexity of this IDWT operation is given by:

$$C = c_z + p_1 N\{Lc_m + (L-1)c_a\}. \tag{2}$$

where, c_z is the complexity of a single zero test, c_a is the complexity of an addition, c_m is the complexity of a multiplication, L is the filter length, and p_1 is the probability that not all N coefficients are zero. This probability can be obtained by averaging measurements over some typical images. Since the best choices for root and leaf size tend to be similar, and to avoid the overhead of having to switch sizes in each band, in our experiments we selected the same sizes for all bands and levels of decomposition.

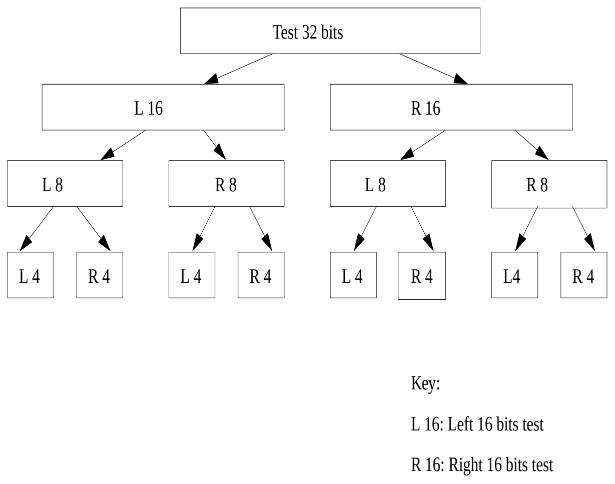

Figure 5. Zero Testing Algorithm

Note that row and column filtering have to be treated differently, In column filtering we have information about the locations of zeros. However, the number of zeros decreases after column and their exact can only be calculated from existing bitmaps. Most of the gain is thus achieved in the first (column) filtering.

3.3. Results and Discussion

We present results using Daubechies orthogonal filters of length 8 with three levels of decomposition. The algorithm was trained on three images: Barbara, Goldhill and Lena, and then tested on three other images (Boat, Creek and Lake) in addition to the training images. Fig. 6 represents the percentage savings in computation time of the IDWT compared to the baseline IDWT implementation. Note that we include the time to compute the bitmap when determining the complexity for the Inverse DWT with zero testing.

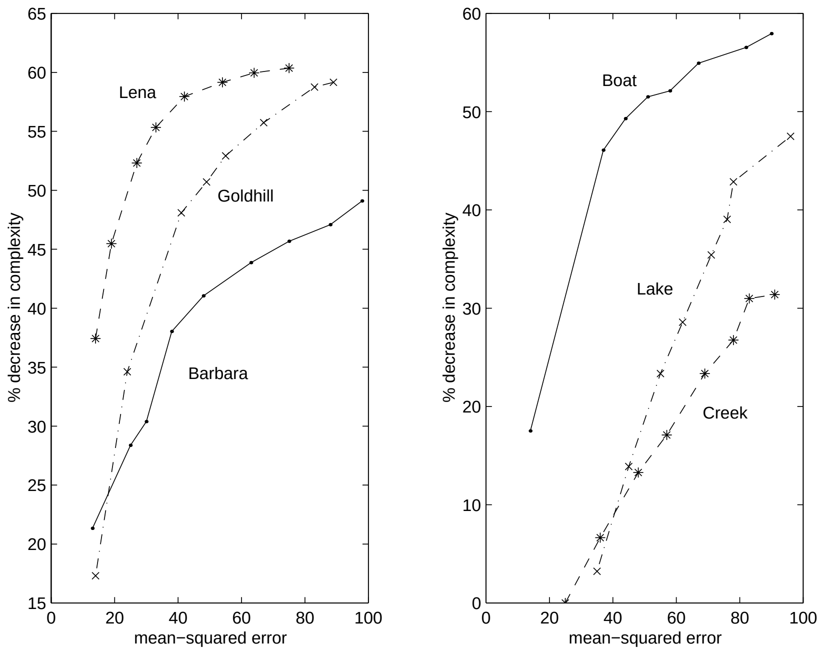

Figure 6. Complexity Savings vs Distortion

Our results show a marked improvement over the baseline case. The algorithm also performed well with the three images it was not trained on. The complexity saving begins to level off after certain distortions when most insignificant coefficients are zero.

4. PARALLEL IMPLEMENTATIONS OF DWT USING LIFTING FACTORIZATIONS

Parallel implementations of the DWT are particularly relevant in applications where speed is important, data volume can be significant and cost may not be the main consideration. Important examples include a number of DWT-based image processing applications, such as satellite imagery compression and analysis in remote sensing [22,23], fast image retrieval and browsing in large databases and real-time pattern recognition and autonomous tracking [24]. Platforms for parallel implementation include Massively Parallel Processors (MPP), such as Intel's Paragon and MasPar's MP-1/2, or combinations of cheap multiple Processing Elements (PE), forming a Network of Workstations (NOW) [25] or a Local Area Multicomputer (LAM) [26]. Examples of analysis of the parallel implementation of the DWT include [22,27–29]. In our work [30,31] we specifically address the boundary issues that arise in the computation of the wavelet transform, when data is split to be sent to more than one processor.

Note that boundary issues are also encountered in standard filtering using the FFT and can be easily handled with appropriate data overlapping. However, the DWT consists of the *recursive* application of a filtering operation followed by downsampling, and thus the amount of overlap required will grow exponentially with the number of

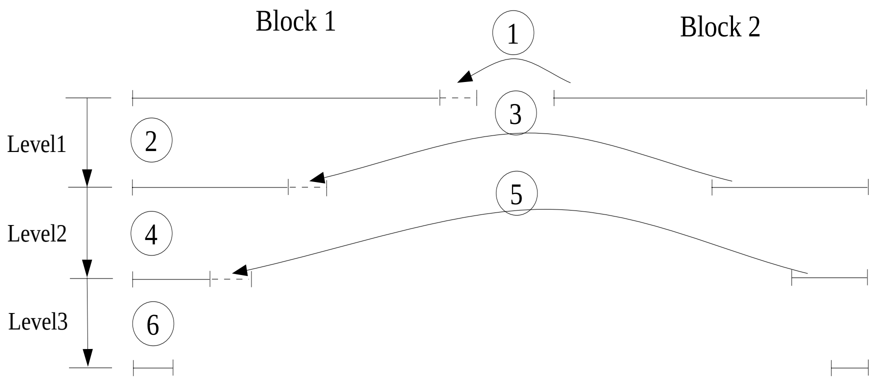

Figure 7. An example DWT dataflow chart using *Boundary Preprocessing* in a two-processor parallel system. Processors 1 and 2 are allocated with input data block 1 and 2 respectively. Solid lines: completely transformed data; Dashed lines: boundary samples from the neighboring block. Operations 1,3,5: communicate boundary data samples to neighboring blocks; Operations 2,4,6: transform for current level.

levels of decomposition. Thus, if initial data overlap is used, only one communication is needed, but the amount of overhead in terms of memory can be significant.

An alternative approach would be to compute each level of the wavelet decomposition and then exchange data between processors as illustrated in Fig. 7. This *Boundary Preprocessing* approach is used in most existing parallel architectures [27,29]. In this case the information that has to be exchanged is reduced, and is a function of the filter length, but the number of exchanges is now equal to the number of levels of decomposition. Obviously this communication overhead adversely affects the speedup of parallel systems, specially those with large communication latencies, such as NOWs and LAMs [32].

To reduce this communication overhead, two alternative approaches have been proposed in the literature. First, it is possible to use the *overlap* technique, as discussed above, where sufficient input data samples are given to each processor so that no communication is needed [33]. This approach may be undesirable due to the increase in mempory requirements and the size of the overlap. A second approach is the *tiling* method [28], which approximates, at each processor, unavailable boundary data samples by symmetric or periodic extensions. While this approach completely eliminates interprocessor communication, without requiring much additional memory, it results in incorrect wavelet coefficients along block boundaries, and this can lead to significant performance degradation in coding applications [34], as well as in pattern recognition and image analysis applications.

4.1. The Split-and-Merge Parallel Architecture

We have introduced [30,31] a new technique, *Boundary Postprocessing*, for the DWT computation near block boundaries. Using this technique, the DWT can be computed correctly, while the interprocessor communication overhead is significantly reduced.

4.1.1. DWT as a Finite State Machine

The basic idea is to model the DWT as a *Finite State Machine (FSM)*, which updates/transforms each raw input sample (initial state) progressively into a wavelet coefficient (final state) as long as there are enough neighboring samples present. Obviously, data samples near block boundaries can only be updated to intermediate states due to lack of enough neighboring samples, and thus we propose that partially updated samples can be saved and communicated between processors. The proposed FSM model is based on exploiting filterbank factorizations such as the ones in [35–37]. In particular, we use the lifting factorization [38] that permits the in-place computation property needed to introduce the FSM approach. Note that filterbank factorizations have been motivated traditionally by the reduction of memory and number of operations, whereas here we demonstrate that they can also contribute to a reduction in the communication overhead in a parallel computation.

Using the Euclidean algorithm, Daubechies and Sweldens [38] have shown that, for any FIR wavelet filters, the polyphase matrix $\mathbf{P_s}(z)$ (subscript s stands for the synthesis) has a factorization form as

$$\mathbf{P_s}(z) \quad = \quad \prod_{i=1}^{m} \begin{bmatrix} 1 & -s_i(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -t_i(z) & 1 \end{bmatrix} \begin{bmatrix} K & 0 \\ 0 & 1/K \end{bmatrix} \tag{3}$$

and the corresponding analysis polyphase matrix $\mathbf{P}_a(z)$ as

$$\mathbf{P_a}(z) \quad = \quad \begin{bmatrix} 1/K & 0 \\ 0 & K \end{bmatrix} \prod_{i=m}^{1} \begin{bmatrix} 1 & 0 \\ t_i(z) & 1 \end{bmatrix} \begin{bmatrix} 1 & s_i(z) \\ 0 & 1 \end{bmatrix} \tag{4}$$

where $(s_i(z), t_i(z))$ are Laurent polynomials and $m \leq \lfloor L/2 \rfloor$ (L is the filter length) is determined by the specific factorization from. The Perfect Reconstruction (PR) property can be easily verified as $\mathbf{P}_s(z)\mathbf{P}_a(z) = \mathbf{I}$ where $\mathbf{I}$ is the identity matrix. It has been shown that such a lifting-factorization based DWT algorithm is, asymptotically for long filters, twice as fast as a direct implementation of the filterbank (Theorem 8 in Daubechies and Sweldens[38]). For example, for the popular $(9,7)$ filters [39], a one-level decomposition using the direct approach requires 11.5 mult/add operations per output sample, while the cost of the lifting-based algorithm is 7.

The elementary matrices (upper and lower triangular) in (4) can be further classified into *prediction/lifting, updating/dual lifting* operations [40]. Since their computation requirements are the same, without loss of generality, we use $l^i(z)$ to represent either $s_i(z)$ or $t_i(z)$ and let $\mathbf{e}^i(z)$ be the corresponding elementary matrix. That is

$$\mathbf{e}^i(z) \equiv \begin{bmatrix} 1 & l^i(z) \\ 0 & 1 \end{bmatrix} \qquad \text{or} \qquad \mathbf{e}^i(z) \equiv \begin{bmatrix} 1 & 0 \\ l^i(z) & 1 \end{bmatrix}.$$

The inverses of $\mathbf{e}^i(z)$ are the matrix inverses, denoted as $\mathbf{e}^{-i}(z)$.

Let us consider the input $\mathbf{X}(z)$ as a column vector, define the intermediate states in the process of transformation, $\{\mathbf{X}^i(z), i = 0, 1, \cdots, 2m+1\}$, where $\mathbf{X}^i(z)$ is the result of applying the operation $\mathbf{e}^{i-1}(z)$ to $\mathbf{X}^{i-1}(z)$, and where the initial input is $\mathbf{X}^0(z) = \mathbf{X}(z)$. Obviously, the forward transform starts from the raw input data samples, the initial state $\mathbf{X}^0(z) = \mathbf{X}(z)$, and, using these elementary matrices $\mathbf{e}^i(z)$, progressively updates the input into the wavelet transform coefficients, the final state $\mathbf{Y}(z) = \mathbf{X}^{2m+1}(z)$. The inverse transform reverses this process to reconstruct the input. One can see that, because of the in-place computation property, every time we generate $\mathbf{X}^i(z)$, we only need to store this set of values, i.e., we do not need to know any of the other $\mathbf{X}^j(z)$, for $j < i$, in order to compute the output. Thus, it is clear that the filtering operation can be seen as a *Finite State Machine* (FSM) as depicted in Fig. 8, where each elementary matrix $\mathbf{e}^i(z)$ updates the FSM state $\mathbf{X}^i(z)$ to the next higher level $\mathbf{X}^{i+1}(z)$.

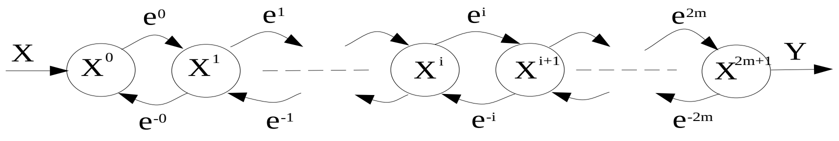

Figure 8. State transition diagram of DWT as a Finite State Machine.

4.1.2. Boundary Postprocessing

In the FSM model, the DWT is computed as each input sample updates itself with the help of samples in its neighborhood (weighted by the *prediction/updating* filter coefficients at each stage). This is illustrated by Fig. 9. If there are enough neighboring samples available, then this state transition process will continue until each sample reaches its final state (a wavelet coefficient). However, if not enough neighboring samples are available, then only partial updating is possible which will leave samples in intermediate states. This is exactly the situation for samples near block boundaries when the DWT is computed on a block-by-block basis. Note that in Fig. 9 only the central wavelet coefficient is computed while those on the sides of the block cannot be computed. However these partially updated coefficients are maintained as "state information". As long as necessary state information in each block is preserved, the boundary transform can be completed after independent transformations of each block. This is done by communicating this state information across blocks so that postprocessing operations are initiated to complete the transform. We thus call this boundary transform technique *Boundary Postprocessing* in contrast to the *Boundary Preprocessing* approach which communicates raw data samples before the start of transform of each block [11,41,14,15,42,27,29].

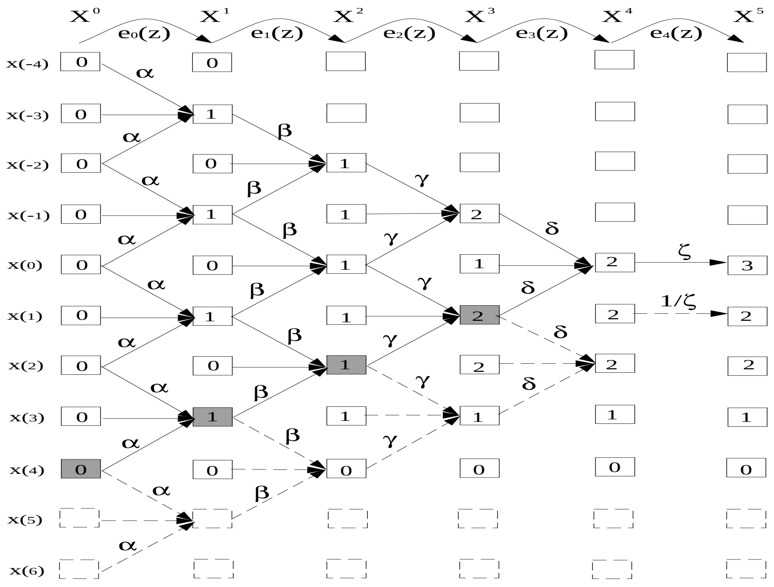

Figure 9. Illustration of DWT as a FSM using the $(9,7)$ wavelet filters. Solid lines represent operations performed for the transform of input pair $(x(0), x(1)$ while dashed lines represent operations to be performed later for input pair $(x(2), x(3)$. Along each line is the multiplication factor with default value 1. The operation at each end node is a summation. Shaded boxes represent state information on one side of the input vector $\mathbf{X}$.

What makes this *Boundary Postprocessing* technique attractive is that it can be generalized to any arbitrary number of decomposition levels. After one level of decomposition, half of the samples (the high frequency subband) will remain unchanged while the other half (the low frequency subband) starts over another round of state transitions exactly the same as in the previous level of decomposition. This process continues until the transform reaches the deepest level of decomposition. Each block can then be independently transformed up to the required level of decomposition. The state information is communicated after and postprocessing is initiated to complete the transform for boundary samples. In Fig.10 we show an example dataflow chart of a three-level wavelet decomposition using the *Boundary Postprocessing* technique. Application of the proposed *Boundary Postprocessing* technique results in a new parallel DWT architecture, *Split-and-Merge*, shown in Fig.11. As one can see, for 1D wavelet decompositions, only one interprocessor data exchange is needed for any J-level wavelet decompositions. Compared to existing approaches which require J communications [22,27–29], the interprocessor communication overhead is significantly reduced.

In Fig.11 the proposed parallel DWT architecture is shown. The striped data partition scheme, as described by Fridman and Manolakos [27], is used to allocate the input data sequence uniformly onto P available processors. Each processor computes its own allocated data up to the required wavelet decomposition level J. This stage is called *Split*. The output from this stage consists of two parts: (i) completely transformed coefficients and (ii) the state information (partially updated boundary samples). In the second stage, *Merge*, a one-way communication is initiated and the state information is transfered to the neighboring processors. The state information from the neighboring processor is then combined together with its own corresponding state information to complete the whole DWT transform.

4.2. Experimental Results

We provide some experimental results for a case when implementation takes place in a processor network, in which each processor can communicate to every other processor. A typical example is the LAM/NOW systems where locally connected machines are reconfigured into a parallel system. Though virtually any arbitrary topology can be built upon such a physical processor network, for a parallel system local interprocessor communication is preferred

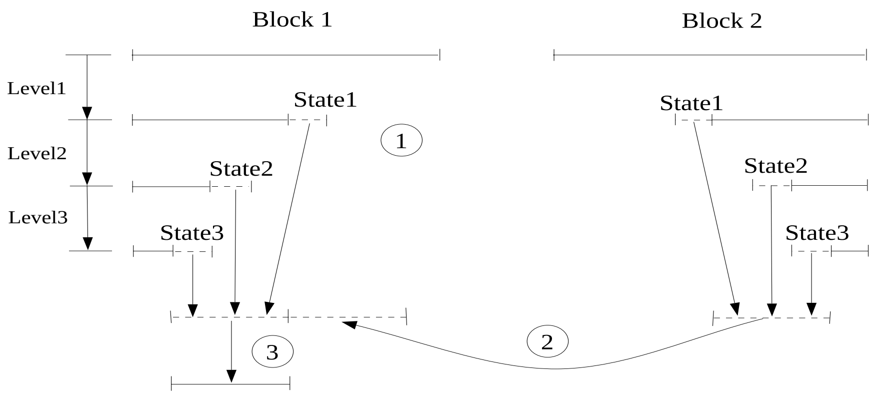

Figure 10. An example dataflow chart of a three-level wavelet decomposition using the proposed *Boundary Postprocessing* technique. Solid lines: completely transformed data; Dashed lines: partially transformed data. Operation 1: each block transforms its own allocated data independently and state information is buffered; Operation 2: state information is communicated to neighboring blocks; Operation 3: complete transform for the boundary data samples.

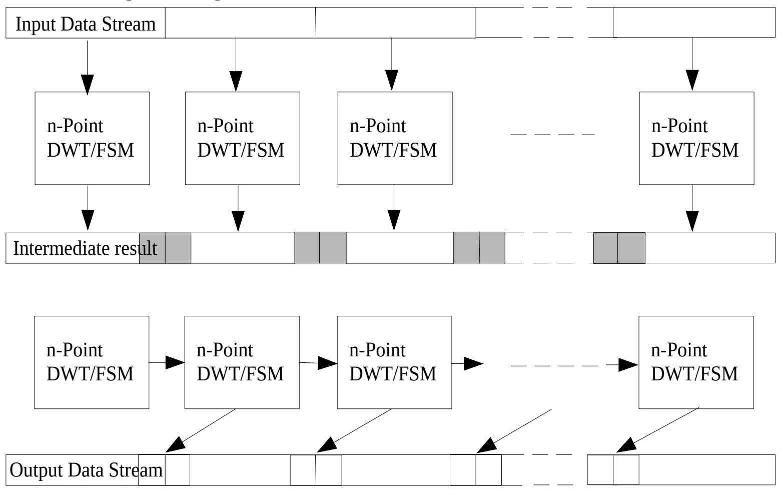

Figure 11. The proposed *Split-and-Merge* parallel DWT architecture. The shaded parts store the state information. In *Split* stage, each processor computes its allocated data independently up to the required decomposition level. In *Merge* stage, a one-way communication is initiated to communicate the state information to neighboring processors. A postprocessing operation is then started to complete the transform for boundary samples.

to reduce the network traffic, and hence the communication overhead. Consequently, we propose to use the strip partition to allocate data to different processors, where processor P_n is allocated with input samples of indices $(x,y), 0 \leq x \leq W-1, nN_c \leq y \leq (n+1)N_c$. The block size is now WxN_c.

In the first stage, *Split*, each processor is allocated with its own strip and transforms up to the required level of decomposition J. Since no segmentation is done in the row direction, state information obviously will only appear along up and down boundaries in each block. This is shown in Fig.12. Next, in the *Merge* stage, only one communication is necessary to transfer/receive the column state information from neighboring processors.

In the simulation, we compare a standard parallel algorithm, one based on lifting but using the standard approach at boundaries, and one based on boundary postprocessing. The Daubechies (7,9) filters are used. The baseline sequential algorithm is chosen to be the fast lifting DWT algorithm [38]. The strip partition strategy is used in the experiment to segment an input 512x512 image into two strips of size 256x512, each of which is loaded into one machine for transform. The parallel platform is LAM 6.1 from Ohio Supercomputer Center [26], which runs over Ethernet connected SUN ULTRA-1 workstations in our lab (CPU clock frequency $133MHz$). Two workstations are used to simulate a parallel system with two processors. The algorithm running time is measured using the $MPI\_Wtime()$ function call from MPI libraries averaging over 50 running instances. The relative speedup is calculated against the

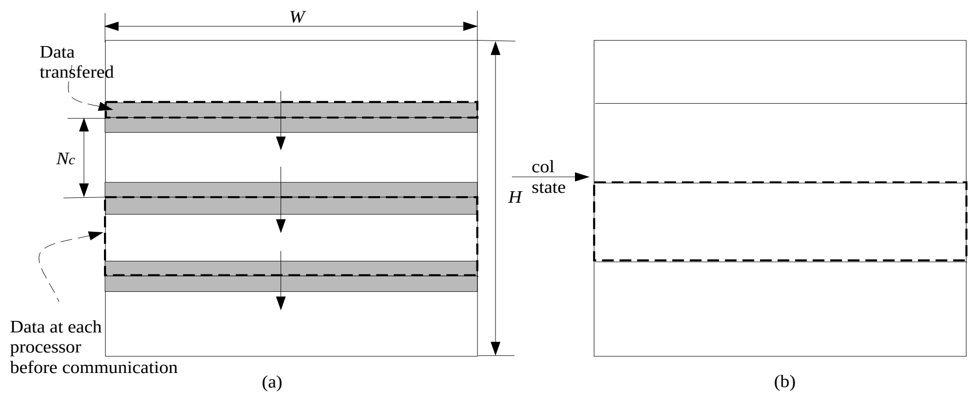

Figure 12. *Merge* operations for strip parallel implementation. (a) transfer row state information from P_i to P_{i+1}; and (b) complete transforms for boundary samples in each processor.

sequential lifting algorithm as $T_{seq}/T_{para} - 1$. The results of DWT running times for different decomposition levels are given in Table.4.2.

As one can see, the simple parallel standard algorithm and the parallel lifting algorithm do not improve that much from the sequential lifting algorithm (relative speedup is only about 10% to 30%) due to communication overhead between the two workstations. However, using the *Boundary Postprocessing* technique, the proposed parallel algorithm provides speedup from 50% to 70% for all five levels of decompositions. It can be concluded that the proposed parallel algorithm can reduce the DWT computation time by significantly reducing the communication overhead.

Table 3. DWT running time of different parallel algorithms (in seconds).

Level	Sequential Lifting	Parallel Standard		Parallel Lifting		Parallel Proposed	
		time	speedup	time	speedup	time	speedup
1	0.3638	0.3115	17%	0.2745	33%	0.2045	78%
2	0.3649	0.3275	11%	0.2899	26%	0.2338	56%
3	0.3952	0.3490	13%	0.2938	34%	0.2369	67%
4	0.4028	0.3513	15%	0.3041	34%	0.2383	69%
5	0.4041	0.3675	9%	0.3165	28%	0.2417	67%

REFERENCES

1. W. Pennebaker and J. Mitchell, *JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, 1994.
2. D. Lee, "New work item proposal: JPEG 2000 image coding system." ISO/IEC JTC1/SC29/WG1 N390, 1996.
3. J. M. Shaprio, "Embedded image coding using zerotrees of wavelet coefficients," *IEEE Trans. on Signal Processing* **41**, pp. 3445–3462, Dec. 1993.
4. A. Said and W. A. Pearlman, "A new fast and efficient image codec based on set partitioning in hierarchical trees," *IEEE Transactions on Circuits and Systems for Video Technology* **6**, pp. 243–250, June 1996.
5. Z. Xiong, K. Ramchandran, and M. Orchard, "Space-frequency quantization for wavelet image coding," *IEEE Trans. on Image Proc.* **6**, pp. 677–693, May 1997.
6. 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.* **6**, pp. 1473–1486, Nov. 1997.
7. Y. Yoo, A. Ortega, and B. Yu, "Image subband coding using progressive classification and adaptive quantization," *IEEE Trans. on Image Proc.* , Jun. 1997. Submitted.
8. C. Chrysafis and A. Ortega, "Efficient Context-based Entropy Coding for Lossy Wavelet Image Compression," in *Proc. DCC'97*, pp. 241–250, (Snowbird, Utah), 1997.
9. 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. DCC'97*, pp. 221–230, (Snowbird, Utah), 1997.

10. C. Christopoulos (Editor), "JPEG 2000 Verification Model Version 2.1," *ISO/IEC JTC/SC29/WG1* , June 1998.
11. M. Vishwanath, "The recursive pyramid algorithm for the discrete wavelet transform ," *IEEE Trans. Signal Processing* **42**, pp. 673–676, March 1994.
12. P. Cosman and K. Zeger, "Memory Constrained Wavelet-Based Image Coding," in *1s Ann. UCSD Conference on Wireless Comm.*, March 1998.
13. P. Cosman and K. Zeger, "Memory Constrained Wavelet-Based Image Coding," *Signal Processing Letters* **5**, pp. 221–223, September 1998.
14. C. Chrysafis and A. Ortega, "Line Based Reduced Memory Wavelet Image Compression," in *Proc. DCC'98*, pp. 308–407, (Snowbird, Utah), 1998.
15. C. Chrysafis and A. Ortega, "Line based, reduced memory, wavelet image compression," *IEEE Transactions on Image Proc.* , 1999. To appear.
16. C. Chrysafis and A. Ortega, "An Algorithm for Low Memory Wavelet Image Compression," in *Proc. ICIP'99*, (Kobe, Japan), Oct 1999.
17. D. Taubman, "Embedded independent block-based coding of subband data," Hewlett Packard, *ISO/IEC JTC/SC29/WG1N871 Document, Copenhagen* , June 1998.
18. K. Froitzheim and H. Wolf, "A knowledge-based approach to JPEG acceleration," in *Proc. of IS&T/SPIE*, (San Jose), Feb. 1995.
19. K. Lengwehasatit and A. Ortega, "DCT computation with minimal average number of operations," in *Proc. of VCIP'97*, vol. SPIE-3024, pp. 71–82, (San Jose, CA), Feb. 1997.
20. H. Guo, "Mapped inverse discrete wavelet transform for data compression," in *Proc. of ICASSP'98*, (Seattle, WA), May 1998.
21. P. Fernandez and A. Ortega, "An input dependent algorithm for the inverse discrete wavelet transform," in *Proc. of Asilomar '98*, (Pacific Grove, CA), Nov 1998.
22. A. Uhl, "A parallel approach for compressing satellite data with wavelets and wavelet packets using PVM," in *Workshop Paragraph '94, RISC - Linz Report Series No. 94-17*, 1994.
23. T. Bell, "Remote sensing," *IEEE Spectrum* , pp. 25–31, 1995.
24. F. Yu and D. Gregory, "Optical pattern recognition: architectures and techniques," *Proc. of the IEEE* **84**(5), pp. 733–752, 1996.
25. T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW team, "A case for NOW (Networks of Workstations)," *IEEE Micro* , Feb. 1995.
26. *http://www.osc.edu/lam.html.*
27. J. Fridman and E. S. Manolakos, "On the scalability of 2-D discrete wavelet transform alogrithms," *Multidimensional Systems and Signal Processing* (8), pp. 185–217, 1997.
28. L. Yang and M. Misra, "Coarse-grained parallel algorithms for multi-dimensional wavelet transforms," *The Journal of Supercomputing* **12**(1/2), pp. 99–118, 1998.
29. O. Nielsen and M. Hegland, "TRCS9721: A scalable parallel 2D wavelet transform algorithm," tech. rep., The Australian National University, Dec. 1997.
30. W. Jiang and A. Ortega, "Parallel architecture for the discrete wavelet transform based on the lifting factorization," in *SPIE, Parallel and Distributed Methods for Image Processing III*, (Denver, CO), July 1999.
31. W. Jiang and A. Ortega, "Efficient discrete wavelet transform architectures based on filterbank factorizations," in *Proc. of Intl. Conf. on Image Proc., ICIP'99*, (Kobe, Japan), Oct 1999.
32. R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson, "Effects of communication latency, overhead, and bandwidth in a cluster architecture," in *ICSA 24*, pp. 716–731, June 1997.
33. F. Kossentini, "Spatially segmented wavelet transform," tech. rep., UBC, 1998. ISOIEC JTC 1SC29WG1 WG1N868.
34. "Report on core experiment codeff1 "Complexity reduction of SSWT"," tech. rep., Motorola Australia, UBC, 1998. ISOIEC JTC 1SC29WG1 WG1N881.
35. P. P. Vaidyanathan and P.-Q. Hoang, "Lattice structures for optimal design and robust implementation of two-channel perfect-reconstruction QMF banks," *IEEE Trans. Acoust., Speech, Signal Processing* **36**, pp. 81–94, Jan. 1988.
36. P. P. Vaidyanathan, "Multirate digital filters, filter banks, polyphase networks, and applications: A tutorial," *Proceedings of The IEEE* **78**, pp. 56–93, Jan. 1990.
37. T. G. Marshall, "Zero-phase filter bank and wavelet coder matrices: Properties, triangular decompositions, and a fast algorithm," *Multidimensional Systems and Signal Processing* **8**, pp. 71–88, 1997.
38. I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," *J. Fourier Anal. Appl.* **4**(3), pp. 247–269, 1998.
39. M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies, "Image coding using the wavelet transform," *IEEE Trans. Image Proc.* **1**, pp. 205–220, Dec. 1992.
40. W. Sweldens, "The lifting scheme: A new philosophy in biorthogonal wavelet constructions," in *Wavelet Applications in Signal and Image Processing III*, A. F. Laine and M. Unser, eds., pp. 68–79, Proc. SPIE 2569, 1995.
41. C. Chakrabarti and M. Vishwanath, "Efficient realizations of the discrete and continuous wavelet transforms: From single chip implementations to mappings on SIMD array computers," *IEEE Trans. on Signal Proc.* **43**, pp. 759–771, Mar. 1995.
42. G. Lafruit, L. Nachtergaele, J. Bormans, M. Engels, and I. Bolsens, "Optimal memory organization for scalable texture codecs in MPEG-4," *IEEE Trans. on Circuits and Systems for Video Technology* **9**, pp. 218–243, Mar. 1999.