Low-Memory Wavelet Transforms for Wireless Sensor Networks: A Tutorial

Stephan Rein and Martin Reisslein

***Abstract*—The computational and memory resources of wireless sensor nodes are typically very limited, as the employed low-energy microcontrollers provide only hardware support for 16 bit integer operations and have very limited random access memory (RAM). These limitations prevent the application of modern signal processing techniques to pre-process the collected sensor data for energy and bandwidth efficient transmission over sensor networks. This tutorial introduces communication and networking generalists without a background in wavelet signal processing to low-memory wavelet transform techniques. We first explain the one-dimensional wavelet transform (including the lifting scheme for in-place computation), the two-dimensional wavelet transform, as well as the evaluation of wavelet transforms with fixed-point arithmetic. Then, we explain the fractional wavelet filter technique which computes wavelet transforms with 16 bit integers and requires less than 1.5 kByte of RAM for a 256×256 gray scale image. We present case studies illustrating the use of these low-memory wavelet techniques in conjunction with image coding systems to achieve image compression competitive to the JPEG2000 standard on resource-constrained wireless sensor nodes. We make the C-code software for the techniques introduced in this tutorial freely available.**

***Index Terms*—Image sensor, sensor network, wavelet transform.**

I. Introduction

SENSOR networks consist of nodes equipped with sensors, a processing unit, a short-range communication unit with low data rates, and a battery. To allow these systems to economically monitor the environment, the nodes have to be low-cost and energy efficient. For these reasons, the processing unit of the nodes is typically a low-complexity 16-bit microcontroller, which has limited processing power and random access memory (RAM) [1], [2]. A sensor node can be equipped with a small camera to track an object of interest or to monitor the environment; thus, forming a camera sensor network [3], [4]. These imaging-oriented applications, however, exceed the resources of a typical low-cost sensor node. Even if it is possible to store an image on a low-complexity node using cheap, fast, and large flash memory [5]–[7], the image transmission over the network can consume too much bandwidth and energy [8]. Therefore, image processing techniques are required either (i) to compress the image data, e.g., with wavelet based techniques that exploit the similarities in transformed versions of the image, or (ii) to extract the interesting features from the images and transmit only image meta data.

Up to very recently these image processing techniques had memory requirements that exceeded the resources on the low-complexity microcontrollers. Hence, in practice more complex and expensive platforms have been employed, as for example the iMote2 sensor equipped with the Enalab camera [9]. The iMote2 employs a 32 bit Intel XScale processing core with 256 kByte RAM and requires a battery board for system power. Another example for a typical camera sensor node is the *citric* platform [10], which consists of a camera daughter board with the high-performance PXA270 processor connected to a Tmote Sky board (a variant of the telos B node [11]). Both of these platforms are examples of current image processing sensor platforms [12] which are significantly more expensive than a low-complexity sensor node with a 16 bit processor and RAM in the range of 10 kByte [1].

With advanced data and signal processing techniques that only require very small random access memory, low priced camera sensor networks can be built by connecting small cameras and external flash memory to low-complexity sensor nodes, which can make use of the additional hardware by a software update. A modern pre-processing technique to inspect or compress an image is the discrete wavelet transform. The wavelet transform decorrelates the data, allowing for the extraction of interesting features. Also, the wavelet transform decomposition allows for the application of tree coding algorithms to summarize the typical patterns, e.g., the embedded zerotree wavelet (EZW) [13] or the set partitioning in hierarchical trees (SPIHT) scheme [14].

In this tutorial we introduce communications and networking generalists without a background in signal processing to a range of wavelet transform techniques culminating in recently developed signal processing techniques that require only very small memory for wavelet transforms. In particular, the recently developed fractional wavelet filter [15] requires less than 1.5 kByte of RAM to transform an image with 256×256 8-bit pixels using only 16-bit integer arithmetic, as illustrated in Figure 1. Thus, the fractional wavelet filter works well within the limitations of typical low-cost sensor nodes [1], [2].

This tutorial is organized as follows. Section II gives background on the problem of low-memory wavelet transforms in sensor networks. Section III provides a tutorial on the one- and two-dimensional discrete wavelet transform, which can be based on folding computations or on the more advanced *lifting scheme* for in-place computation. Section IV explains how to compute wavelet transforms with fixed-point arithmetic, which

Manuscript received 14 May 2010; revised 17 September 2010.

S. Rein is with the Telecommunication Networks Group, Technical University Berlin (e-mail: rein@tkn.tu-berlin.de).

M. Reisslein is with the School of Electrical, Computer, and Energy Eng., Goldwater Center, MC 5706, Arizona State University, Tempe, AZ 85287-5706 (e-mail: reisslein@asu.edu).

Digital Object Identifier 10.1109/SURV.2011.100110.00059

a) original image b) two-level transform of image

Fig. 1. Two-level wavelet transform computed on a low-cost 16 bit microcontroller using less than 1.5 kByte of memory with the *fractional wavelet filter*. Each one-level transform results in four subbands denoted by LL, LH, HL, and HH. The contrast of each subband was adjusted to fill the entire intensity range.

is the basis for the efficient integer-based computation of the transform on a microcontroller. Section V gives a tutorial on the fractional wavelet filter. Case studies illustrating the usage and performance of the wavelet transform techniques covered in this tutorial are presented in Section VI. In Section VII we summarize this tutorial.

II. Related Work

One major difficulty in applying the discrete two-dimensional wavelet transform to a platform with scarce resources is the need for large random access memory. Implementations on a personal computer (PC) generally keep the entire source and/or destination image in memory; also, horizontal and vertical filters are applied separately. As this is generally not possible on resource-limited platforms, recent research efforts have examined memory-efficient wavelet transform techniques.

Significant research efforts have gone into the implementation of the wavelet transform on field programmable gate arrays (FPGA), see for instance [16]–[19]. The FPGA-platforms are generally designed for one special purpose and are typically inappropriate for a sensor node that has to perform a variety of tasks including communication and analysis of the surrounding area [1], [2], [20]. This tutorial does not cover FPGAs; instead we focus on image wavelet transform techniques for a general microcontroller with very small RAM.

The traditional approach to build a camera sensor node has been to connect a second platform with a more capable processor to the sensor node [12]. Instead, this tutorial considers a sensor node where a small camera is directly connected to the microcontroller through the universal asynchronous receiver/transmitter (UART) interface and the wavelet transform is performed on the microcontroller, which is extended by a directly connected multimedia flash memory card [5].

The multi-hop transmission path from a sensor node to a sink node in wireless sensor networks is exploited in a few studies, e.g., [21], [22], for distributing the wavelet coefficient computations over several nodes. We focus on the computation of the wavelet transform at one given node in this tutorial.

As we demonstrate in the case studies in Section VI, a low-memory wavelet transform can be combined with a low-memory image coding system, e.g., [23], to achieve on a microcontroller compression performance competitive to the current JPEG2000 image compression system. An alternative strategy that compresses images directly on low-complexity hardware is studied in [24]. However, the strategy in [24] employs the general JPEG technique which gives significantly lower compression performance than wavelet-based compression.

We now proceed to briefly review the research on low-memory wavelet transforms leading up to the fractional wavelet filter technique. A line-based version of the wavelet transform has been developed in [25]. The line-based transform substantially reduces the memory requirements compared to the traditional transform approach with a system of buffers that only store a small subset of the wavelet transform coefficients. An efficient computation methodology for the line-based transform using the lifting scheme and improved communication between the buffers has been developed in [26] and implemented in C++ on a personal computer (PC) for demonstration. The line-based approach [25], [26], however, can not run on a sensor node with very small RAM, as it uses in the ideal case 26 kByte of RAM for a six-level transform of an 512×512 image. A fractional wavelet filter performing a step-wise computation of the vertical wavelet coefficients

of a two-dimensional image was developed in [15], [27]. The fractional wavelet filter approach reduces the memory requirements of the image wavelet transform significantly compared to the line-based approach and enables the image wavelet transform on microcontrollers with very small RAM.

While this brief review has focused on line-based low-memory wavelet transform so far, we note for completeness that an alternate approach is based on transforming image blocks, see e.g., [28], [29]. These block-based approaches have similar memory requirements as [25] and are often used in conjunction with block-based wavelet image codecs. We focus in this tutorial exclusively on the low-memory wavelet transform, and more specifically on line-based approaches that readily meet the flash memory constraint [5]–[7] of reading contiguous 512 Byte blocks through line-by-line image data access. We note that low-memory wavelet-based image coders are studied in [23], [30].

III. Tutorial on Image Wavelet Transform

In this section the general computation of the image wavelet transform is described. The section does not describe the basics and foundations of the wavelet transform, as there is extensive literature on wavelet theory, see for instance [31]–[33]. The computational aspects of the wavelet transform, however, are rarely discussed, as most of the published literature assumes the usage of wavelet toolboxes on personal computers.

This section is organized as follows. In Subsection III-A the *convolution* operation is applied to compute the one-dimensional wavelet transform. The coefficients of the *Daubechies* 9/7 wavelet are provided. To achieve a multi-level transform the *pyramidal algorithm* is applied. Furthermore, the advanced *lifting scheme* is outlined, which computes the transform in place. Subsection III-C details how the one-dimensional transform is applied to perform the two-dimensional image wavelet transform.

A. Wavelet Transform for One Dimension Using Convolution

1) Haar-Wavelet: To simplify the readability, we refer to the *fast dyadic wavelet transform* as the wavelet transform. Other transforms are not covered in this tutorial. A wavelet transform can be computed by convolution filter operations. Convolution is an elementary signal processing operation, where a flipped filter vector is shifted step-wise over a signal vector while for each position the scalar product between the overlapping values is computed. The boundaries of a signal can be linearly (zero-padding), circularly, or symmetrically extended. For the wavelet transform we use here a symmetrical extension, as it is reasonable to obtain a smooth signal change. If $\mathbf{h} = [h_0, h_1, h_2]$ denotes a filter vector of length $L = 3$ and $\mathbf{s} = [s_0, s_1, s_2, s_3]$ denotes a signal vector of length $N = 4$, the symmetrical convolution $\mathrm{conv}(\mathbf{s}, \mathbf{h})$ can be illustrated as

$$\begin{array}{|cccccccc|} \hline h_2 & h_1 & h_0 & \rightarrow & & & & \\ s_2 & s_1 & s_0 & s_1 & s_2 & s_3 & s_2 & s_1 \\ \hline \end{array} \quad (1)$$

The vector $\mathbf{h}$ is shifted over the signal vector $\mathbf{s}$ and for each position the scalar product of the overlapping values is computed, giving a result vector of length $N + L - 1$.

A one-dimensional wavelet transform is typically performed by separately applying two different filters, namely a lowpass and a highpass filter to the original signal, as illustrated in Figure 2a). The lowpass and highpass filter are also referred to as *scaling filter* and *wavelet filter*, respectively; we employ the term *wavelet filter* for both filter types. The filtered signals are sampled down by leaving out each second value such that the odd indexed values $(1, 3, 5, \ldots)$ are kept from the lowpass filtering (i.e., the first value is kept, the second one discarded, and so on) and the even indexed values $(2, 4, 6, \ldots)$ are kept from the highpass filtering (i.e., the first value is discarded, the second one is kept, and so on). The thus obtained values are the wavelet coefficients and are referred to as *approximations* and *details*. As observed in Fig. 2a), the aggregate number of approximation and detail coefficients equals the signal dimension.

To reconstruct the original signal the coefficients have to be sampled up by inserting zeros between each second value. Upsampling of the vector $\mathbf{s} = [1, 2, 3, 4]$ thus results in $[1, 0, 2, 0, 3, 0, 4, 0]$. The up-sampled versions of the approximations and details are filtered by the synthesis lowpass and highpass filters. (The analysis and synthesis wavelet filters are employed for the forward and backward transform, respectively.) Both filtered arrays of values are summed up. Such a wavelet transform can be performed multiple times to achieve a multi-level transform, as illustrated in Figure 2b). This scheme, where only the approximations are further processed, is called the *pyramidal algorithm*.

We now apply the *Haar-Wavelet* filter to the example signal $\mathbf{s} = [4, 9, 7, 3, 2, 0, 6, 5]$. The Haar-filter coefficients for the lowpass are given as $\mathbf{l} = [0.5, 0.5]$, and for the highpass as $\mathbf{h} = [1, -1]$. The detail coefficients for the first level are computed by shifting the flipped version of the highpass filter over the signal, resulting in

$$\mathbf{d} = \mathrm{conv}(\mathbf{s}, \mathbf{h}) = [4, 5, -2, -4, -1, -2, 6, -1, -5]. \quad (2)$$

After down sampling the details are given as $\mathbf{d}_d = [5, -4, -2, -1]$. All the other coefficients are computed similarly. Noting that each second convolution value is discarded, the filter coefficients can be shifted by two sample steps instead of one step, as illustrated for the first two values:

$$\begin{array}{|cccccccc|} \hline -1 & 1 & \rightarrow & & & & & \\ 4 & 9 & 7 & 3 & 2 & 0 & 6 & 5 \\ \hline \end{array} \quad \begin{array}{|cccccccc|} \hline & & -1 & 1 & \rightarrow & & & \\ 4 & 9 & 7 & 3 & 2 & 0 & 6 & 5 \\ \hline \end{array} \quad (3)$$

These convolution values are computed as $-1 \cdot 4 + 1 \cdot 9 = 5$ and $-1 \cdot 7 + 1 \cdot 3 = -4$. The three-level wavelet transform for this signal is given as:

4	9	7	3	2	0	6	5	signal
6.5	5	1	5.5	5	−4	−2	−1	level 1
5.75	3.25	−1	4.5	level 2				
4.5	-2.5	level 3						
4.5	−2.5	−1	4.5	5	−4	−2	−1	result

Note that the result vector contains all the detail coefficients of the previous levels, which are not further processed.

Generally, instead of the filters $\mathbf{l} = [0.5, 0.5]$ and $\mathbf{h} = [1, -1]$ which we used for ease of illustration, the normalized

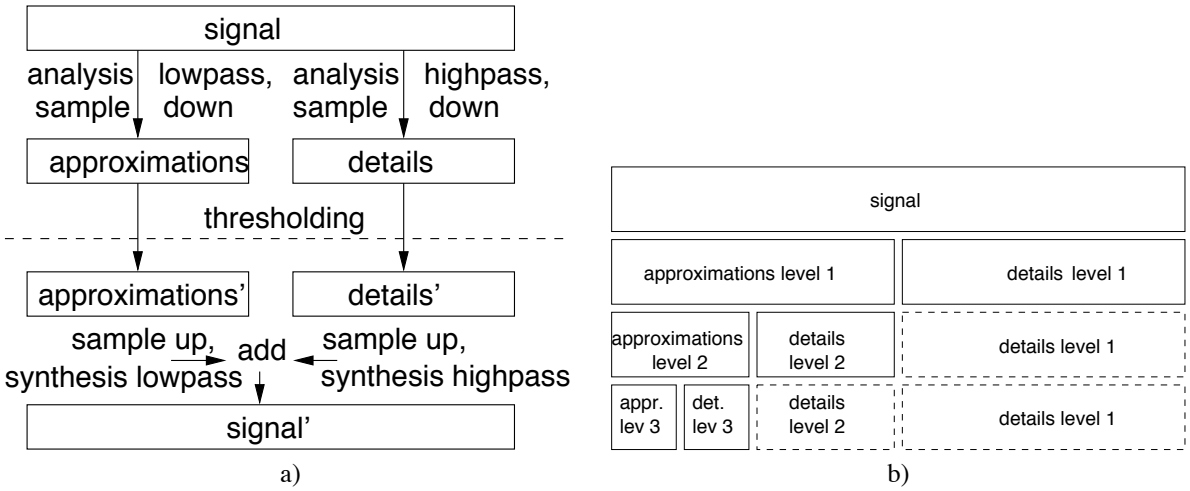


Fig. 2. Figure a) shows a one-level forward (analysis) wavelet transform, and the corresponding backward (synthesis) wavelet transform that reconstructs the original signal. The thresholding operation deletes very small coefficients and is a general data compression technique. These small coefficients are expected to not contain meaningful information. Figure b) illustrates a 3-level forward wavelet transform. The approximations of a given level form the input for the next level. The details are not further processed and are copied to the next transform levels.

TABLE I
FILTER COEFFICIENTS OF THE DAUBECHIES 9/7 WAVELET. THESE COEFFICIENTS ARE EMPLOYED IN THIS ARTICLE AS THEY GIVE STATE-OF-THE-ART IMAGE COMPRESSION. THEY ARE PART OF THE JPEG2000 IMAGE COMPRESSION STANDARD. COMPUTATIONALLY, THE 9/7 WAVELET IS NOT AN IDEAL CHOICE, AS IT REQUIRES FLOATING POINT COMPUTATIONS AND A RELATIVELY LARGE NUMBER OF COEFFICIENTS. A COMPUTATIONAL SCHEME THAT APPLIES A 9/7 IMAGE TRANSFORM ON A LIMITED PLATFORM WITH LESS THAN 1.5 KBYTE OF RAM IS INTRODUCED IN THIS TUTORIAL.

j	analysis lowpass l_j	analysis highpass h_j	synthesis lowpass l_j	synthesis highpass h_j
-4	0.037828			0.037828
-3	-0.023849	0.064539	-0.064539	0.023849
-2	-0.110624	-0.040689	-0.040689	-0.110624
-1	0.377403	-0.418092	0.418092	-0.377403
0	0.852699	0.788486	0.788486	0.852699
1	0.377403	-0.418092	0.418092	-0.377403
2	-0.110624	-0.040689	-0.040689	-0.110624
3	-0.023849	0.064539	-0.064539	0.023849
4	0.037828			0.037828

filter coefficients $\mathbf{l} = [\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}]$ and $\mathbf{h} = [\frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}]$ are used. The normalized coefficients make the transform *orthonormal*; thus, conserving the signal energy.

In Section III-C, the two-dimensional wavelet transform is outlined, which in contrast to the one-dimensional transform, only computes one level for each input line. In the next subsection, a more advanced wavelet, the Daubechies 9/7 Wavelet, will be discussed. The Daubechies 9/7 Wavelet has more coefficients while the principles of the Haar-Wavelet continue to hold.

2) Daubechies 9/7 Wavelet: The *biorthogonal Daubechies* 9/7 wavelet [31] (also called FBI-fingerprint wavelet or Cohen-Daubechies-Feveau wavelet) is used in many wavelet compression algorithms, including the *embedded zerotree wavelet* (EZW) [13], the *set partitioning in hierarchical trees* (SPIHT) algorithm [14], [33], and the JPEG2000 compression standard for lossy compression [34]. It is given by its filter coefficients in Table I [33]. We note that for fulfilling the requirements for perfect signal reconstruction, the synthesis lowpass is generally generated by the flipped version of the analysis highpass, whereby the sign is changed in an alternating way. Similarly, the synthesis highpass is generated from the analysis lowpass.

A wavelet transform with the filter coefficients in Table I is computed as in the case of the Haar-wavelet by low- and highpass filtering of the input signal. The input signal can be a single line $\mathbf{s} = [s_0, s_1, \ldots, s_{N-1}]$ of an image with the dimension N. The two resulting signals are then down-sampled, that is, each second value is discarded to form the *approximations* and *details*, which are two sets of $N/2$ wavelet coefficients.

More specifically, the approximations a_i are computed as

$$a_i = \text{convp}(\mathbf{s}, \mathbf{l}, i) = \sum_{j=-4}^{4} s_{i+j} \cdot l_j, \qquad i = 0, 1, \ldots, N-1 \tag{4}$$

where we introduce the notation $\text{convp}(\mathbf{s}, \mathbf{l}, i)$ to denote the wavelet coefficient at position i obtained by convolving signal $\mathbf{s}$ with filter $\mathbf{l}$. Note that the filter coefficients l_j are the analysis lowpass filter coefficients given in Table I. Analogously, using the highpass analysis filter coefficients h_j in Table I we obtain the details d_i as

$$d_i = \text{convp}(\mathbf{s}, \mathbf{h}, i) = \sum_{j=-3}^{3} s_{i+j} \cdot h_j, \qquad i = 0, 1, \ldots, N-1. \tag{5}$$

Note that due to the symmetry of the lowpass and the highpass filters, the sign of the filter index j (which runs from -4 to $+4$) in the subscript of the signal s in (4) and (5) is arbitrary (i.e., $+j$ could be replaced by $-j$).

Downsampling to $[a_0, a_2, \ldots, a_{N-2}]$ gives $N/2$ approximations and downsampling to $[d_1, d_3, \ldots, d_{N-1}]$ gives $N/2$ details. The down-sampling can be incorporated into the convolution by only computing the coefficients remaining after the downsampling. In particular, to obtain the approximations the center of the lowpass filter shifts over the even signal samples $s_0, s_2, \ldots, s_{N/2-2}$ and to obtain the details the center of the highpass filter moves over the odd signal samples $s_1, s_3, \ldots, s_{N/2-1}$. Intuitively, by aligning the center of the

lowpass filter with the even signal samples and the center of the highpass filter with the odd samples, each filter "takes in" a different half of the input signal. Taken together, both filters "take in" the complete input signal.

In order to avoid border effects we generally perform a point-symmetrical extension at the signal boundaries, i.e., the signal $\mathbf{s} = [s_0, s_1, \ldots, s_{N-1}]$ is extended for the highpass filtering to $[s_3, s_2, s_1, s_0, s_1, \ldots, s_{N-2}, s_{N-1}, s_{N-2}, s_{N-3}, s_{N-4}]$. Thus, as an illustration of the extension, the detail coefficient $d_1 = \text{convp}(\mathbf{s}, \mathbf{h}, 1)$ is obtained by aligning the center of the high pass filter with signal sample s_1:

$$\begin{array}{ccccccc} h_{-3} & h_{-2} & h_{-1} & h_0 & h_1 & h_2 & h_3 \\ s_2 & s_1 & s_0 & s_1 & s_2 & s_3 & s_4. \end{array}$$

Re-numbering the approximations $[a_0, a_2, \ldots, a_{N-2}]$ to $[a_0, a_1, \ldots, a_{N/2-1}]$ and the details $[d_1, d_3, \ldots, d_{N-1}]$ to $[d_0, d_1, \ldots, d_{N/2-1}]$ gives the approximations and details with consecutive indices. Incorporating the downsampling and re-numbering, the approximations are given as

$$a_i = \text{convp}(\mathbf{s}, \mathbf{l}, 2i) = \sum_{j=-4}^{4} s_{2i+j} \cdot l_j, \qquad i = 0, 1, \ldots, \frac{N}{2} - 1 \tag{6}$$

and the details as

$$d_i = \text{convp}(\mathbf{s}, \mathbf{h}, 2i+1) = \sum_{j=-3}^{3} s_{2i+1+j} \cdot h_j, \qquad i = 0, 1, \ldots, \frac{N}{2} - 1. \tag{7}$$

To achieve multiple transform levels this process can be repeated.

B. Wavelet Transform for One Dimension with the Lifting Scheme

In this section the *lifting scheme* [35], [36] is described, which computes the wavelet transform in-place. For a general introduction on the theory behind the lifting scheme see [37].

1) Lifting Scheme for the Haar-Wavelet: In Section III-A1 the Haar-wavelet transform is computed over sets of two consecutive signal samples. To calculate the details, the filter vector $[1, -1]$ is shifted over the signal:

$$\begin{array}{|cccccccc|} \hline -1 & 1 & \rightarrow & & & & & \\ +4 & 9 & 7 & 3 & 2 & 0 & 6 & 5 \\ \hline \end{array} \tag{8}$$

We now focus on the first set of signal samples. The detail coefficient is calculated as

$$d = s_1 - s_0 = 9 - 4 = 5, \tag{9}$$

and the approximation coefficient is calculated as

$$a = \frac{s_0 + s_1}{2} = \frac{4 + 9}{2} = 6.5. \tag{10}$$

This computation can be performed in place if we compute

$$\begin{aligned} s_1 &= s_1 - s_0 = 9 - 4 = 5 \\ s_0 &= s_0 + \frac{s_1}{2} = 4 + \frac{5}{2} = 6.5. \end{aligned} \tag{11}$$

In summary, the in-place computation steps are:

$$\boxed{4}\boxed{9} \rightarrow \boxed{4}\boxed{5} \rightarrow \boxed{6.5}\boxed{5} \tag{12}$$

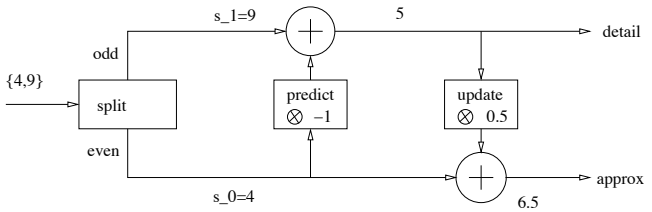


Fig. 3. Basic lifting scheme, i.e., a forward transform with order zero moment. The *split* operation separates the signal in even and odd samples. The predict and update operations simply multiply the input.

This forward transform can be performed using the so-called *lifting scheme*, as illustrated in Figure 3. The lifting scheme first conducts a *lazy wavelet* transform, that is, it separates the signal in even and odd samples. Then, the detail coefficient is predicted using its left neighbor sample. The even sample predicts the odd coefficient. Note that the sample indices start with zero, e.g., $\mathbf{s} = [s_0, s_1, s_2, \ldots]$; thus, the even set is given by $\mathbf{s}_e = [s_0, s_2, s_4, \ldots]$ and the odd set is given by $\mathbf{s}_o = [s_1, s_3, s_5, \ldots]$. The *update* stage ensures that the coarser approximation signal has the same average as the original signal.

For the inverse transform the stages have to be reversed, as illustrated in Figure 4. The original samples are recovered by first reversing the update stage:

$$s_0 = 6.5 - 5/2 = 4. \tag{13}$$

The odd sample is recovered as

$$s_1 = 5 - (-1 \cdot 4) = 9. \tag{14}$$

Both operations can again be computed in place:

$$\boxed{6.5}\boxed{5} \rightarrow \boxed{4}\boxed{5} \rightarrow \boxed{4}\boxed{9} \tag{15}$$

The merge operation merges the even and the odd samples.

2) Linear Interpolation Wavelet: The predict filter of the lifting scheme provides polynomial cancelation, while the update filter preserves the moments. (A moment here refers to the wavelet transform in that the wavelet coefficients are part of a polynomial representation of the input signal.) This means for the Haar wavelet that the predict filter eliminates the relationship between two samples (zero order correlation). The update filter preserves the sample average of the coefficients. The predict and update filters both have order one, as they use either one past or one future sample.

If correlations over more than two samples shall be addressed, as it is, for instance, needed for image compression, higher filter orders can be employed. The *linear* wavelet transform uses filters of order two. To explain the linear wavelet transform, let $s_{2k}, k = 0, 1, \ldots$, denote the even samples, and $s_{2k+1}, k = 0, 1, \ldots$, denote the odd samples. The detail coefficient d_k is then computed as

$$d_k = s_{2k+1} - \frac{s_{2k} + s_{2k+2}}{2}, \; k = 0, 1, \ldots \tag{16}$$

This is a type of prediction where the detail gives the difference to the linear approximation using the left and the right sample. The approximation coefficient a_k is given as

$$a_k = s_{2k} + \frac{d_{k-1} + d_k}{4}, \; k = 1, 2, \ldots \tag{17}$$

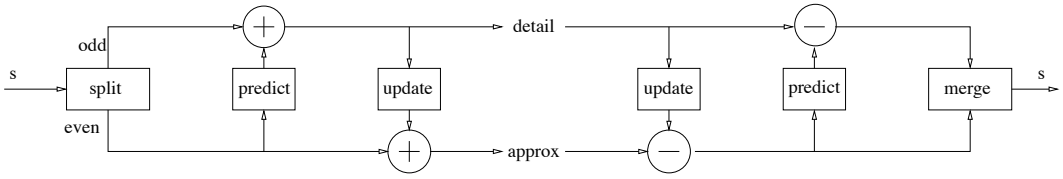


Fig. 4. Forward and inverse transform using the lifting scheme. The inverse transform implements the scheme backwards thus recovering the original samples.

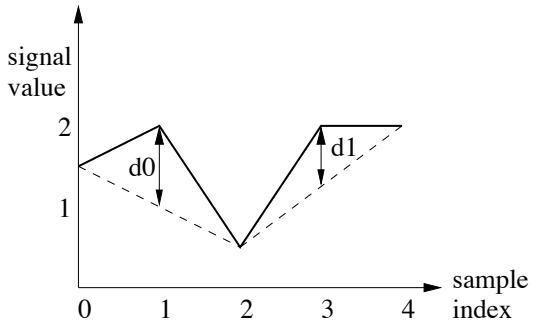


Fig. 5. Numerical Example of Linear Wavelet Transform. The bold line gives the signal samples and the dotted line the prediction. Even samples are used to calculate the details as the difference between the prediction and the actual values.

These steps are reversible, analogous to the Haar-wavelet.

For the example signal $\mathbf{s} = [s_0, s_1, s_2, s_3, s_4] = [1.5, 2, 0.5, 2, 2]$, the details d_0 and d_1 are computed as
$d_0 = s_1 - \frac{s_0+s_2}{2} = 2 - \frac{(1.5+0.5)}{2} = 1$ and
$d_1 = s_3 - \frac{s_2+s_4}{2} = 2 - \frac{(0.5+2)}{2} = 0.75$,
as illustrated in Figure 5. The approximation a_1 is computed as
$a_1 = s_2 + \frac{d_0+d_1}{4} = 0.5 + \frac{(1.75)}{4} = 0.9375$.
The illustrated linear transform corresponds to the biorthogonal (2,2) wavelet transform [35], [38].

Now, we examine the predict and update operations from a digital filter perspective. The predict operation uses a current and a future sample; therefore, the z-transform of its filter is given as $p(z) = -\frac{1}{2}(z^0 + z^1)$. The update filter uses a current and a past sample: $u(z) = \frac{1}{4}(z^0 + z^{-1})$. For signal samples $s_n,\ n = 0, 1, 2, \ldots$, these equations can be written as $p_n = -0.5(s_n + s_{n+1})$ and $u_n = \frac{1}{4}(s_n + s_{n-1})$. In the next section, the lifting scheme for the 9/7 wavelet is reviewed, which similarly employs update and predict operations.

3) Lifting Scheme for the 9/7 Wavelet: Figure 6 illustrates the lifting scheme structure for the 9/7 wavelet. From Section III-A2 we know that the lowpass analysis filter has nine coefficients and the lowpass synthesis filter has seven coefficients. From these coefficients the parameters α, β, γ, δ, and ζ can be derived through a factoring algorithm [35] giving

$$\begin{aligned} \alpha &= -1.5861343420693648 \\ \beta &= -0.0529801185718856 \\ \gamma &= 0.8829110755411875 \\ \delta &= 0.4435068520511142 \\ \zeta &= 1.1496043988602418. \end{aligned} \tag{18}$$

These parameters form the filters that are applied in the so-called *update* and *predict* steps:

$$\begin{aligned} P_\alpha &= [\alpha, \alpha] \\ U_\beta &= [\beta, \beta] \\ P_\gamma &= [\gamma, \gamma] \\ U_\delta &= [\delta, \delta]. \end{aligned} \tag{19}$$

Similarly, the lifting operations can be partitioned to reconstruct the original signal, as illustrated in Figure 7.

Table II details the computation scheme for the 9/7 lifting scheme. First the original signal is separated into even and odd values. Then follow four update and predict steps where the parameters α and γ are convoluted with the odd part and parameters β and δ are convolved with the even part of the signal. Note that the conv() operation is a convolution without signal extension at the boundaries, as the signal extension is implied by the given data vector.

For example, consider a signal $\mathbf{s} = [3, 4]$ and a filter $\mathbf{h} = [2, 1]$. The convolution with point-symmetrical extension would be computed as $[1 \cdot 4 + 2 \cdot 3,\ 1 \cdot 3 + 2 \cdot 4,\ 1 \cdot 4 + 2 \cdot 3]$. In Table II, the computation is only $[1 \cdot 3 + 2 \cdot 4]$, so there is no boundary computation; and, the result dimension is smaller by one. The signal value in the convolution is extended by one value. Thus, the result of the convolution has exactly half of the original signal dimension N, so it can be added to $\boldsymbol{s_e}$ or $\boldsymbol{s_o}$.

The operator notation in Table II follows the conventions of the C programming language. For instance, the "+=" operator means that the vector on the left-hand side is updated by (assigned) its value plus the result of the right-hand side. For instance, $\boldsymbol{a} \mathbf{+=} \mathbf{7}$ assigns $\boldsymbol{a}$ the value $\boldsymbol{a} \mathbf{+ 7}$, i.e., $\boldsymbol{a} \leftarrow \boldsymbol{a} \mathbf{+ 7}$.

C. Two-dimensional Wavelet Transform

For the one-level wavelet transform we computed all wavelet levels for one input line and stored the result in a second line with the same dimension. For the two-dimensional transform, we perform a one-level transform on all rows of an image, as illustrated in Figure 8. Specifically, for each row the approximations and details are computed by lowpass and highpass filtering. This results in two matrices $\boldsymbol{L}$ and $\boldsymbol{H}$, each with half of the original image dimension. These matrices are similarly low- and highpass filtered to result in the four submatrices $\boldsymbol{LL}$, $\boldsymbol{LH}$, $\boldsymbol{HL}$, $\boldsymbol{HH}$, which are called *subbands*: *LL* is the all-lowpass subband (coarse approximation image), *HL* is the vertical subband, *LH* is the horizontal subband, and *HH* is the diagonal subband. To achieve a second wavelet transform level, only the $\boldsymbol{LL}$-submatrix is processed, as illustrated in Figure 9, and similarly for further levels. The

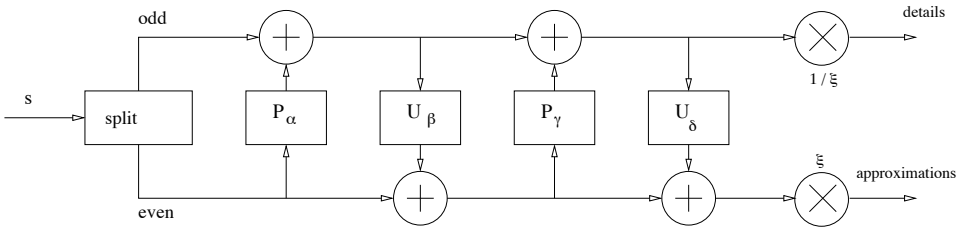


Fig. 6. Lifting scheme structure for the Daubechies 9/7 wavelet. The parameters α, β, γ, and δ refer to predict and update filters that are similar to the Haar wavelet lifting filters. The ζ parameter scales the output.

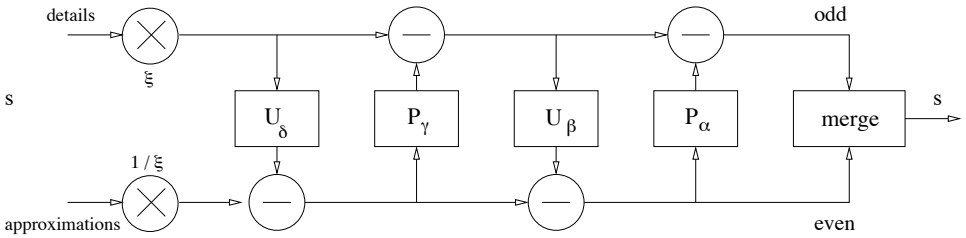


Fig. 7. Inverse wavelet transform for the 9/7 wavelet using the lifting scheme. The predict, update, and scaling steps from the forward transform are undone using the same operators.

TABLE II
COMPUTATION OF THE LIFTING SCHEME FOR THE FORWARD (FIGURE A)) AND BACKWARD (FIGURE B)) WAVELET TRANSFORM.

a) forward:

$$\mathbf{s}_e = [s_0, s_2, s_4, \ldots, s_{N-2}] \quad (20)$$
$$\mathbf{s}_o = [s_1, s_3, s_5, \ldots, s_{N-1}] \quad (21)$$
$$\mathbf{s}_o += \text{conv}([\mathbf{s}_e, s_{N-2}], [\alpha, \alpha]) \quad (22)$$
$$\mathbf{s}_e += \text{conv}([s_1, \mathbf{s}_o], [\beta, \beta]) \quad (23)$$
$$\mathbf{s}_o += \text{conv}([\mathbf{s}_e, s_{N-2}], [\gamma, \gamma]) \quad (24)$$
$$\mathbf{s}_e += \text{conv}([s_1, \mathbf{s}_o], [\delta, \delta]) \quad (25)$$
$$\mathbf{s}_e \cdot = \zeta \quad (26)$$
$$\mathbf{s}_o /= \zeta \quad (27)$$

b) backward:

$$\mathbf{s}_o \cdot = \zeta \quad (28)$$
$$\mathbf{s}_e /= \zeta \quad (29)$$
$$\mathbf{s}_e -= \text{conv}([s_1, \mathbf{s}_o], [\delta, \delta]) \quad (30)$$
$$\mathbf{s}_o -= \text{conv}([\mathbf{s}_e, s_{N-2}], [\gamma, \gamma]) \quad (31)$$
$$\mathbf{s}_e -= \text{conv}([s_1, \mathbf{s}_o], [\beta, \beta]) \quad (32)$$
$$\mathbf{s}_o -= \text{conv}([\mathbf{s}_e, s_{N-2}], [\alpha, \alpha]) \quad (33)$$

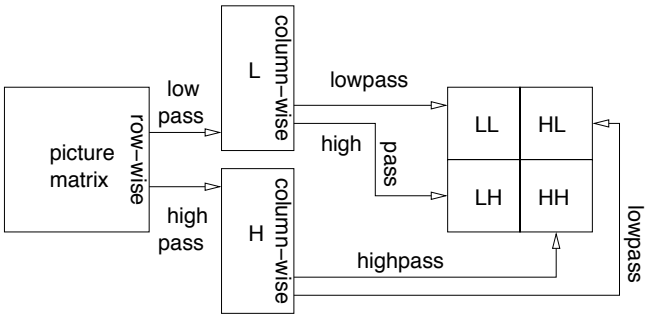


Fig. 8. One-level image wavelet transform. The wavelet-image is partitioned in four regions that are called *subbands*. The image is first filtered row-by-row, resulting in the *L* and *H* matrix, which both have half the dimension of the original image. Then, these matrices are filtered column-by-column, resulting in the four subbands. It is also possible to first filter column-by-column and then line-by-line. The *HL* subband, for instance, denotes that first the highpass and then the lowpass filter was applied. In the literature, the subband *HL* is sometimes denoted by *LH*. Therefore, the position of these subbands in the destination matrix is sometimes interchanged.

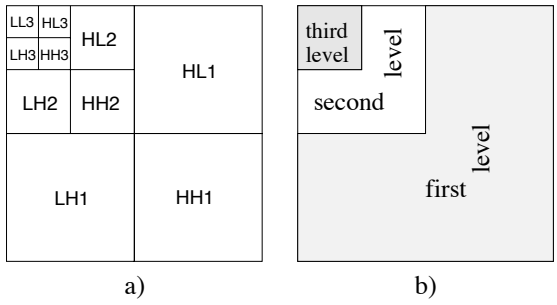


Fig. 9. Three-level image wavelet transform. Figure a) illustrates the subbands of each level. For computation of a level the LL-subband of the previous wavelet level is retrieved. In Figure b) the areas of each level are hatched.

operations can be repeated on each of the LL subbands to obtain the higher level subbands. Note that $\boldsymbol{LL}$ represents a smaller and smoothed version of the original image. $\boldsymbol{LH}$ intensifies the horizontal elements, $\boldsymbol{HL}$ the vertical, and $\boldsymbol{HH}$ the diagonal elements. The reconstruction of the original image can be performed similarly. An example of a two-level wavelet transform was given in Fig1b).

IV. WAVELET TRANSFORM WITH FIXED-POINT NUMBERS

A. Introduction

This section gives a short tutorial on fixed-point arithmetic, which is needed to perform real (floating-point) number calculations with integer numbers. Fixed-point numbers differ from

floating-point numbers in that the decimal point is fixed (the position is known at compile time and the programmer has to adjust the position). Many low-cost microprocessors provide only hardware support for 16 bit integer numbers, and floating point operations have to be implemented by the compiler. The 16 bit fixed-point arithmetic speeds up computations with real numbers at the expense of lower precision and more detailed algorithm analysis (including an estimation of the possible range of the results). In the following subsections we provide the necessary background for fixed-point evaluations of wavelet transforms.

B. Number Representation

A number is represented as a sequence of bits, i.e., a binary word, in a personal computer or microcomputer. A binary word with $M = 16$ bits can be given as $b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}\ldots b_1b_0$. If this word is interpreted as an unsigned integer, its numerical value is computed as

$$B = \sum_{n=0}^{M-1} 2^n b_n. \tag{34}$$

The internal binary word is also called the *mantissa*. The most significant bit is located on the left hand side.

To introduce the fixed-point notation we take an 8 bit number as an example: 01000011. If we interpret this word as an unsigned integer, it would take the value 67. It is also possible to interpret this word as a fixed-point number. For instance, we can imply a binary point as follows: 010000.11. Now, the value of this word is calculated as $2^4+2^{-1}+2^{-2} = 16.750$, or generally as

$$B = \frac{1}{2^{\exp(b)}} \sum_{n=0}^{M-1} 2^n b_n, \tag{35}$$

where $\exp(b)$ (here $\exp(b) = 2$) denotes the number of binary positions that follow the binary point (i.e., note that $\exp(\cdot)$ does not refer to the exponential function). The number range for an unsigned fixed-point number B is $[0,\ \frac{2^M-1}{2^{\exp(b)}}]$. Note that in the internal representation there is no point in the word. The number is declared as an unsigned integer number and the debugger probably will show the value 67. The correct interpretation of the number depending on the context is the programmer's responsibility.

Generally, the numerical value B (as interpreted by the user) of a fixed-point number b is thus given by the integer value of the internal binary word divided by $2^{\exp(b)}$, i.e.,

$$B = b \cdot 2^{-\exp(b)}. \tag{36}$$

We denote b for the integer value of the internal binary word and $\exp(b)$ for the exponent of b. Throughout this tutorial, a capital letter will be used for the number as interpreted by the user, and the correspondent lower case letter for the integer value of the internal binary word.

1) Negative Numbers: For representing negative numbers, the MSB bit of the internal representation is reserved for the sign. If the MSB is zero, the number is positive. If the MSB is one, the number is negative. Generally, the *two's complement* is used for the internal representation of signed numbers. The two's complement representation has the advantage that the number zero has only one representation and that addition and subtraction are same operations as for unsigned numbers. The two's complement represents positive numbers in the same way as in the ordinary unsigned notation with the exception that the MSB must be zero. A negative number is formed by inverting all bits and adding one to the result, e.g., $-2 = \text{inv}(0010) + 1 = 1101 + 1 = 1110$. To obtain the integer number of the binary word, all bits are inverted and one is added. Generally, the interpreted value A of a two's complement is given as

$$A = 2^{-\exp(a)}[-2^{M-1}a_{M-1} + \sum_{n=0}^{M-2} 2^n a_n].$$

An M-bit variable for unsigned integers can represent numbers from 0 to $2^M - 1$. Using the two's complement, the integer range is given as

$$[-2^{M-1},\ 2^{M-1} - 1],$$

and the fixed-point number range is thus given as

$$[-2^{M-1-\exp(a)},\ 2^{M-1-\exp(a)} - 2^{-\exp(a)}].$$

2) Q-format: We use the *Q-format* to denote the fixed-point number format. The Q-format assumes the two's complement representation; therefore, an M-binary word always has $M-1$ bits for the absolute numerical value. The $Qm.n$ format denotes that m bits are used to designate the two's complement integer portion of the number, not including the MSB bit (sign bit), and that n bits are used to designate the two's complement fractional portion of the number, that is, the number of bits to the right of the radix point. Thus, a $Qm.n$ number requires $M = m + n + 1$ bits. (Some tutorials, e.g., [39] include the sign bit in m.) The range of a $Qm.n$ number is $[-2^m,\ 2^m - 2^{-n}]$ and the *resolution* is given by 2^{-n}. The value m in Qm,n is optional; if m is omitted, it is assumed to be zero. The special case of arithmetic with $m = 0$ is commonly referred to fractional arithmetic and operates in the range $[-1, 1]$. In this tutorial we employ the general fixed-point arithmetic since our computations require larger number ranges.

C. Basic Operations in Fixed-Point Arithmetic

This section gives a tutorial on computing basic arithmetic operations with fixed-point arithmetic. In this tutorial we build on the computational strategies of [40]. An alternative computation approach [41] includes fixed-point data-types for compiler optimization. The presented arithmetic works for both signed and unsigned numbers.

Throughout this section, a conversion of a fixed-point number to a different format is performed by multiplying the number with $2^{\exp(\cdot)}$, whereby $\exp(\cdot)$ refers to a given exponent. This operation performs a bit-shift operation. There exist two definitions for the shift operation, the *logical* and the *arithmetic* shift. Logical and arithmetic left shift both shift out the MSB bit, that is, the MSB bit is discarded, and a zero bit is shifted in. However, there is a difference between logical and arithmetic right shift: The logical right shift is

performed on unsigned binary numbers and inserts a zero bit. The arithmetic right shift relevant in this tutorial is performed on signed numbers and inserts a copy of the sign bit (i.e., the MSB bit). Logical and arithmetic shift are both called *literal shift*, as they are shifts on the binary word. A *virtual* shift does not change the binary word but the exponent.

1) Changing the Exponent (Virtual Shift): For some operations the operands are required to have the same exponent. To change the exponent $\mathbf{exp}(a)$ of a number a to the exponent $\mathbf{exp}(b)$, note that

$$a \cdot 2^{-\exp(a)} = a \cdot 2^{\exp(b)-\exp(a)} \cdot 2^{-\exp(b)}. \quad (37)$$

Thus, if $\mathbf{exp}(b) \geq \mathbf{exp}(a)$ we shift the bits of a to the left by $\mathbf{exp}(b) - \mathbf{exp}(a)$ positions. Whereas, if $\mathbf{exp}(b) < \mathbf{exp}(a)$, we shift the bits of a to the right by $\mathbf{exp}(a) - \mathbf{exp}(b)$ positions.

2) Addition and Subtraction: For addition and subtraction the two numbers a and b have to be converted to the exponent of the result number c. Then,

$$c = a \cdot 2^{-\exp(c)} + b \cdot 2^{-\exp(c)} = (a+b) \cdot 2^{-\exp(c)}. \quad (38)$$

If the exponents of a and b are equal, the two numbers can simply be added. In case of an overflow, the sum of two binary numbers requires one more integer bit in the result, that is, if the numbers are in the form $Qa.b$, the result requires the form $Q(a+1).b$.

3) Multiplication: The product of two binary words a and b can be computed with an integer multiplication as

$$A \cdot B = a \cdot 2^{-\exp(a)} \cdot b \cdot 2^{-\exp(b)} = a \cdot b \cdot 2^{-(\exp(a)+\exp(b))}. \quad (39)$$

The exponent of the result is the sum of the input exponents. If the numbers are in the form $Qa.b$ and $Qc.d$, the result is in the form $Q(a+c).(b+d)$ for unsigned numbers, or in the form $Q(a+c+1).(b+d)$ for signed numbers, whereby $(a+c)$ is the maximum number of integer bits and $(b+d)$ the required number of fractional bits (in order to not lose precision).

Note that in order to compute the product of two numbers A and B with the exponents $\mathbf{exp}(a)$ and $\mathbf{exp}(b)$ and the desired result exponent $\mathbf{exp}(c)$, the product $a \cdot b$ has to be converted from the exponent $\mathbf{exp}(a) + \mathbf{exp}(b)$ to the exponent $\mathbf{exp}(c)$. In the special case of two numbers A and B both with exponent $\mathbf{exp}(a)$, the product is computed as $a \cdot b \cdot 2^{-2\exp(a)}$. In the special case of a fixed-point number in format $Qa.b$ being multiplied with an integer number, the result is a fixed-point number in format $Qa.b$.

Two examples are now given to 1) illustrate the required numbers of integer bits and 2) to explain the required numbers of fractional bits. Two signed input numbers with M bits (recall that M includes the sign bit) require a result with $2M$ bits. Consider an example with two signed $M = 8$ bit input numbers. We consider the extreme ends of the input number range $[-2^7 = -128,\ 2^7 - 1 = 127]$, and evaluate the product $(-128) \cdot (-128) = 16384$. The output now requires 16 bits, because the number range of $M = 15$ bits would only include the interval $[-2^{14} = -16384,\ 2^{14} - 1 = 16383]$. In the unsigned case, $M = 15$ bits are sufficient.

In the second example, the product of two binary numbers is considered:

$$\overbrace{01011.110}^{11.75} \cdot \overbrace{11011.101}^{27.625} = \overbrace{101000100.100110}^{324.59375}$$

Internally, the integer calculation $94 \cdot 221 = 20774$ is performed, resulting in the binary sequence 101000100100110. The $3+3=6$ last digits of this number are fractional digits, as each input number has 3 binary digits. The input number format $Q5.3$ does not influence the calculation, but sets the position of the radix point. Note that the result only requires 9 binary integer digits in this example. In general, $5+5=10$ is the maximum number of required integer digits.

The intermediate results of such operations may exceed the format of the final result. Many 16 bit microcontrollers, such as the dsPIC [42], have a 16×16 bit multiplier that can calculate 32 bit intermediate integer results, which then have to be right shifted to obtain 16 bit results.

4) Division: We consider a division of A by B, where the exponents of A, B, and the result C are equal, i.e., $\mathbf{exp}(a) = \mathbf{exp}(b) = \mathbf{exp}(c)$. The result can thus be evaluated as

$$C = \frac{A}{B} = \frac{a}{b} = \frac{a}{b} \cdot 2^{\exp(a)} \cdot 2^{-\exp(a)} \quad (40)$$

$$= \frac{a \cdot 2^{\exp(a)}}{b} \cdot 2^{-\exp(a)} \quad (41)$$

$$= c \cdot 2^{-\exp(c)}, \quad (42)$$

whereby only $c = a \cdot 2^{\exp(a)}/b$ has to be computed. To minimize rounding errors, we first compute the product $a \cdot 2^{\exp(a)}$ and then the division by b. The special case of a fixed point number A divided by an integer number B requires no conversion, and the result is given in the format of A. The format of a signed division for two operands in format $Qa.b$ and $Qc.d$ is given in format $Q(a+d+1).(c+b)$ [43].

D. Implementation Notes

Building on [40], we implemented the fixed-point arithmetic macros required for low-memory wavelet transforms in the C programming language and make our source code freely available at http://mre.faculty.asu.edu/fwf_code. Note that the data types differ on a personal computer (PC) and a microcontroller in that a different number of bits may be allocated for a data type on each of the systems. For instance, the integer data type defined by `int` allocates 32 bits on a PC while it only allocates 16 bits on the microcontroller. For consistency, we represent fixed-point numbers using a custom-defined datatype INT16 that represents integers with 16 bits. We do not consider code improvements, such as, using assembly language for selected calculations or taking advantage of specific features of the employed micro-controller to ensure that our C source code is compatible with a wide range of embedded C-compilers.

E. Example for One-dimensional Discrete 9/7 Wavelet Transform

We now give an example of a fixed-point wavelet filter implementation. We filter an input line of an image with N

TABLE III
DAUBECHIES 9/7 A) ANALYSIS AND B) SYNTHESIS WAVELET FILTER COEFFICIENTS IN REAL AND $Q15$ DATA FORMAT. THE $Q15$ DATA FORMAT IS A FIXED-POINT REPRESENTATION OF REAL NUMBERS IN THE RANGE OF $[-1,\ 1-2^{-15}]$ WHICH REQUIRES 16 BITS.

a) Analysis filter coefficients

	analysis lowpass l_j		analysis highpass h_j	
j	real	$Q15$	real	$Q15$
0	0.852699	27941	0.788486	25837
±1	0.377403	12367	**−0.418092**	**−13700**
±2	**−0.110624**	**−3625**	**−0.040689**	**−1333**
±3	**−0.023849**	**−781**	0.064539	2115
±4	0.037828	1240	0	0

b) Synthesis filter coefficients

	synthesis lowpass l_j		synthesis highpass h_j	
j	real	$Q15$	real	$Q15$
0	0.788486	25837	0.852699	27941
±1	0.418092	13700	**−0.377403**	**−12367**
±2	**−0.040689**	**−1333**	**−0.110624**	**−3625**
±3	**−0.064539**	**−2115**	0.023849	781
±4	0	0	0.037828	1240

samples using the Daubechies 9/7 one-dimensional forward wavelet transform. The original input line is then reconstructed by an inverse (backward) wavelet transform. Our focus is on the considerations for converting an algorithm from floating-point to fixed-point representation.

1) Overview: A one-level wavelet transform calculates the *approximations* through lowpass-filtering of the signal and the *details* through highpass-filtering of the signal. Each of the two filtered signals is sampled down, that is, each second value is discarded. Thus, there will be $N/2$ approximation coefficients a_i and $N/2$ detail coefficients d_i which are computed as

$$a_i = \text{convp}(\mathbf{s}, \mathbf{l}, 2i) = \sum_{j=-4}^{4} s_{2i+j} \cdot l_j, \quad i = 0, 1, \ldots, \frac{N}{2} - 1 \tag{43}$$

$$d_i = \text{convp}(\mathbf{s}, \mathbf{h}, 2i+1) = \sum_{j=-3}^{3} s_{2i+1+j} \cdot h_j, \quad i = 0, 1, \ldots, \frac{N}{2} - 1, \tag{44}$$

whereby l_j and h_j denote the filter coefficients given in Table IIIa). Note that we extend the signal symmetrically at its boundaries.

For the inverse wavelet transform the approximations and details are sampled up, that is, zeros are inserted between each second value. Then, the synthesis lowpass and highpass filters with the coefficients from Table IIIb) are applied. The two resulting signals of dimension N are added to form the reconstructed signal with dimension N.

2) Selecting Appropriate Fixed-Point Q-Format: To compute the filter operation with fixed-point arithmetic, the filter coefficients first have to be converted to the Q-format. We determine the appropriate Q-format for the filter coefficients by comparing their number range, which is $[-0.42, 0.85]$, with the Q-format number ranges in Table IV. Clearly, a larger number range implies coarser resolution, i.e., lower precision. Therefore, the Q-format with the smallest number range (finest resolution) that accommodates the number range of the transform coefficients should be selected. We therefore select the $Q0.15$ format. (Selecting a format with a larger number range, e.g., the $Q1.14$ would unnecessarily reduce precision.) Table III gives the integer coefficients that result from multiplying the real filter coefficients by 2^{15}. These integer filter coefficients are employed for the fixed-point filter functions.

TABLE IV
NUMBER RANGE FOR THE TEXAS INSTRUMENTS $Qm.n$ FORMAT. USING 16 BITS, THERE ARE 16 POSSIBILITIES TO PLACE THE RADIX POINT WITH m DENOTING THE NUMBER OF INTEGER BITS AND n THE NUMBER OF FRACTIONAL BITS. ONE OF THE INTEGER BITS IS RESERVED FOR THE NEGATIVE REPRESENTATION, THEREFORE ONLY 15 BITS ARE DENOTED BY THE Q-FORMAT FOR 16 BIT VARIABLES.

$Qm.n$	range		resolution
$Q0.15$	-1	0.999969482421875	0.000031
$Q1.14$	-2	1.99993896484375	0.000061
$Q2.13$	-4	3.99987792968750	0.000122
$Q3.12$	-8	7.99975585937500	0.000244
$Q4.11$	-16	15.9995117187500	0.000488
$Q5.10$	-32	31.999023437500	0.000977
$Q6.9$	-64	63.9980468750000	0.001953
$Q7.8$	-128	127.996093750000	0.003906
$Q8.7$	-256	255.992187500000	0.007812
$Q9.6$	-512	511.984375000000	0.015625
$Q10.5$	-1024	1023.96875000000	0.031250
$Q11.4$	-2048	2047.93750000000	0.062500
$Q12.3$	-4096	4095.87500000000	0.125000
$Q13.2$	-8192	8191.75000000000	0.250000
$Q14.1$	-16384	16383.5000000000	0.500000
$Q15.0$	-32768	32767	1.000000

Similarly, we need to determine an appropriate Q-format for the result coefficients of the wavelet transform, i.e., the approximations and details. Generally, one can first obtain an estimate by calculating bounds on the number range of the result coefficients. Through pre-computing the coefficients on a personal computer for a large set of representative sample images one can next check whether a Q format with a smaller number range and correspondingly higher precision can be used [16].

For 8-bit image samples with a number range $[-128, 127]$, we can bound the range of the transform result coefficients by summing the magnitudes of the filter coefficients, e.g., for the lowpass filter $\sum_{j=-4}^{4} |l_j| \approx 1.953$ and multiplying with the maximum magnitude of the input samples, i.e., $1.953 \cdot 128 \approx 249.98$. Comparing with the ranges in Table IV we find that the $Q8.7$ format has a sufficiently large number range for the result coefficients of the one-dimensional wavelet transform of 8-bit input values.

Next, we pre-compute the wavelet transform result coefficients for our specific input values in the first line of Table V on a personal computer with a high-level computing language, such as *Octave* which we provide with the software for the fractional wavelet filter. (Comparing the pre-computed values with the values obtained with the fixed-point transform will also illustrate the minor inaccuracies introduced by the fixed-point transform.) From the pre-computed transform coeffi-

cients in Line 2 of Table V we observe that the format $\boldsymbol{Q7.8}$ has a sufficiently large number range for this example.

3) Forward (Analysis) Wavelet Transform: To illustrate the computation required for the fixed-point wavelet transform, we compute the first detail wavelet coefficient, i.e., $\boldsymbol{d_0}$. In particular, we compute the scalar product of the symmetrically extended input and the analysis highpass filter with the coefficients from Table III. We align the center of the filter with signal sample $\boldsymbol{s_1 = 58}$:

50	58	31	58	50	44	47
2115	$\boldsymbol{-1333}$	$\boldsymbol{-13700}$	25837	$\boldsymbol{-13700}$	$\boldsymbol{-1333}$	2115

The scalar product is computed as $\boldsymbol{(50+47)\cdot 2115+(58+44)\cdot(-1333)+(31+50)\cdot(-13700)+58\cdot 25837 = 458035}$. Note that this result is in the $\boldsymbol{Q0.15}$ format, as the input values are in the $\boldsymbol{Q15.0}$ format and the filter coefficients in the $\boldsymbol{Q0.15}$ format (cf. Eqn. (39)). The computed number is an intermediate result that can exceed the 32 bits. We transform this intermediate result in the $\boldsymbol{Q0.15}$ format to the $\boldsymbol{Q7.8}$ format by right shifting by 7 bits. In particular, the intermediate result 458035 is right shifted by 7 bits to obtain the number 3578 in the $\boldsymbol{Q7.8}$ format, as given Line 3, 5th column in Table V.

We obtain the first detail coefficient $\boldsymbol{d_0}$ in the $\boldsymbol{Q15.0}$ format by right shifting the binary representation of 3578 by 8 bits (i.e., dividing by $\boldsymbol{2^8}$ and retaining only the integer result), resulting in 13, see Line 4. Note that Line 4 gives the computed result of the wavelet transform using only integer calculations to illustrate the loss in precision if the format $\boldsymbol{Q15.0}$ would be required by the user.

4) Backward (Synthesis) Wavelet Transform: To reconstruct the original values from the approximations and details in Line 3 of Table V in the $\boldsymbol{Q7.8}$ format, the integer wavelet synthesis filter coefficients are applied in a similar way. For instance, for the first reconstructed signal value $\boldsymbol{s_0}$, the approximations and the details are filtered by the synthesis coefficients and the two results are added, recall Figure 2a). We first apply the synthesis lowpass on the zero-padded approximations from Line 3, which are symmetrically extended on the left side:

0	18916	0	15516	0	18916	0
$\boldsymbol{-2115}$	$\boldsymbol{-1333}$	13700	25837	13700	$\boldsymbol{-1333}$	$\boldsymbol{-2115}$

The scalar product is computed as $\boldsymbol{2\cdot(18916\cdot(-1333))+15516\cdot 25837 = 350456836}$. Right shifting this intermediate result by 15 bits gives 10695 in the $\boldsymbol{Q7.8}$ format.

These operations are repeated for the detail coefficients from Line 3, which have to be filtered by the high-pass synthesis filter:

0	$\boldsymbol{-1208}$	0	3578	0	3578	0	$\boldsymbol{-1208}$	0
1240	781	$\boldsymbol{-3625}$	$\boldsymbol{-12367}$	27941	$\boldsymbol{-12367}$	$\boldsymbol{-3625}$	781	1240

The scalar product results in $\boldsymbol{-90385148}$, which is $\boldsymbol{-2758}$ in the $\boldsymbol{Q7.8}$ format. To obtain the final reconstructed signal value, we add the two results $\boldsymbol{10695+(-)2758 = 7937}$. To obtain the original number format we compute a right shift on $\boldsymbol{7937}$ by $\boldsymbol{8}$ bits to obtain the reconstructed signal value $\boldsymbol{31}$.

Note that the reconstructed signal values in Line 6 of Table V obtained through the fixed-point arithmetic forward wavelet transform (Line 1 to Line 3) followed by the fixed-point arithmetic backward wavelet transform (Line 3 to Line 6) are slightly different from the original signal values. Generally, the signal values reconstructed from wavelet coefficients computed with fixed-point arithmetic may lead to slight deviations from the original signal samples, as evaluated in Section VI.

V. Tutorial on Fractional Wavelet Filter

This section explains the *fractional wavelet filter* as a technique to compute fractional values of each wavelet subband, thus allowing a low-cost camera sensor node with less than 2 kByte of RAM to perform a multi-level 9/7 image wavelet transform. With 2 kByte of RAM, the image dimension can be up to $\boldsymbol{256\times 256}$ using fixed-point arithmetic and up to $\boldsymbol{128\times 128}$ using floating-point arithmetic. In Section VI we apply the technique on a typical sensor node platform that consists of a 16 bit microcontroller extended with external flash memory (MMC card).

A. Overview

In this subsection we give an overview of the fractional wavelet filter. We first note that the data on the MMC-card can only be accessed in blocks of 512 Bytes; thus, sample by sample access, as easily executed with RAM memory on PCs, is not feasible. Even if it is possible to access a smaller number of samples of a block, the read/write time would significantly slow down the algorithm, as the time to load a few samples is the same as for a complete block. The fractional filter takes this restriction into account by reading complete horizontal lines of the image data.

Throughout this section we consider the forward (analysis) wavelet transform. We explain two versions of the fractional wavelet filter, namely a floating-point version and a fixed-point version. The floating point version in Section V-B uses floating-point arithmetic and achieves high precision, i.e., an essentially lossless transform for most practical purposes.

The fixed point version in Section V-C uses fixed-point arithmetic and needs less memory while being computationally more suitable for a 16 bit controller. The fixed-point version introduces minor image degradations that are evaluated in Section VI.

As illustrated in Fig. 10, for the first transform level, the algorithm reads the image samples line by line from the MMC-card while it writes the subbands line by line to a different destination on the MMC-card (SD-card). For the next transform level the LL subband contains the input data. Note that the input samples for the first level are of the type unsigned char (8 bit), whereas the input for the higher level is either of type float (floating point filter) or INT16 (fixed-point filter) format. The filter does not work in place and for each level a new destination matrix is allocated on the MMC-card. However, as the MMC-card has plenty of memory, this allocation strategy does not affect the sensor's resources. This allocation strategy allows reconstruction of the image from any transform level (and not necessarily from the highest level, as it would be necessary for the standard transform outlined in Section III).

B. Floating-Point Filter

The floating point wavelet filter computes the wavelet transform with a high precision using 32 bit floating point

TABLE V
EXAMPLE OF AN ONE-DIMENSIONAL FIXED-POINT WAVELET TRANSFORM FOR AN IMAGE INPUT LINE (FIRST LINE). THE SECOND LINE GIVES THE APPROXIMATIONS AND DETAILS COMPUTED ON A PC WITH A HIGH-LEVEL LANGUAGE (WE USED *Octave*) FOR SELECTING THE APPROPRIATE $\boldsymbol{Q}$ FORMAT FOR THE APPROXIMATIONS AND DETAILS. THE THIRD LINE GIVES THE FIXED-POINT VERSION OF THE TRANSFORM IN THE $\boldsymbol{Q7.8}$ FORMAT. THESE NUMBERS ARE INTERPRETED IN LINE 4 BY DIVIDING THE FIXED-POINT NUMBERS BY $\boldsymbol{2^8}$. THE RECONSTRUCTED FIXED-POINT SIGNAL VALUES ARE GIVEN IN LINE 5. THESE VALUES CAN BE TRANSFORMED TO THE ORIGINAL DATA FORMAT BY DIVIDING BY $\boldsymbol{2^8}$.

	input signal							
	$\boldsymbol{s_0}$	$\boldsymbol{s_1}$	$\boldsymbol{s_2}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$	$\boldsymbol{s_5}$	$\boldsymbol{s_6}$	$\boldsymbol{s_7}$
1. orig (char)	31	58	50	44	47	52	56	62
	approximations				details			
	$\boldsymbol{a_0}$	$\boldsymbol{a_1}$	$\boldsymbol{a_2}$	$\boldsymbol{a_3}$	$\boldsymbol{d_0}$	$\boldsymbol{d_1}$	$\boldsymbol{d_2}$	$\boldsymbol{d_3}$
2. $\boldsymbol{T}$(Octave)	60.60714	73.88776	65.01055	80.76063	13.97674	$\boldsymbol{-4.72244}$	0.46590	3.89484
3. $\boldsymbol{T(Q7.8)}$	15516	18916	16643	20675	3578	$\boldsymbol{-1208}$	119	997
4. $\boldsymbol{T(Q7.8)/2^8}$	60	73	65	80	13	$\boldsymbol{-4}$	0	3
	reconstructed signal							
	$\boldsymbol{s_0}$	$\boldsymbol{s_1}$	$\boldsymbol{s_2}$	$\boldsymbol{s_3}$	$\boldsymbol{s_4}$	$\boldsymbol{s_5}$	$\boldsymbol{s_6}$	$\boldsymbol{s_7}$
5. $\boldsymbol{R(Q7.8)}$	7937	14847	12800	11265	12032	13310	14336	15871
6. $\boldsymbol{R(Q7.8)/2^8}$	31	57	50	44	47	51	56	61

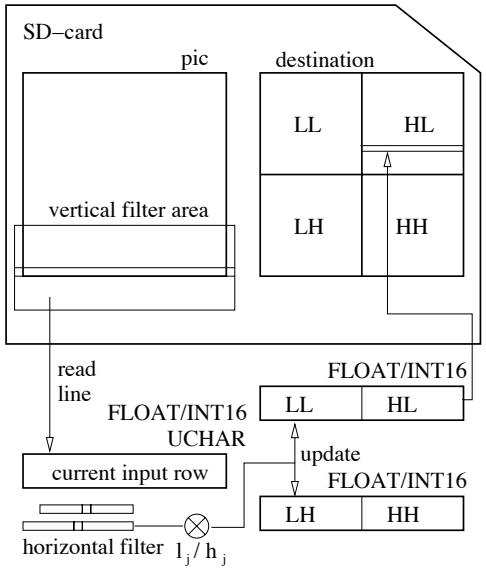


Fig. 10. Illustration of fractional image wavelet transform. The horizontal wavelet coefficients are computed on the fly and employed to compute the fractional wavelet coefficients that update the subbands. For each wavelet level, a different destination object is written to the SD-card. Thus, the image can be reconstructed from any level.

variables for the wavelet and filter coefficients as well as for the intermediate operations. Thus, the images can be reconstructed essentially without loss of information. For the considered image of dimension $\boldsymbol{N \times N}$, the wavelet filter uses three buffers of dimension $\boldsymbol{N}$, one buffer for the current input line and two buffers for two destination lines, as illustrated in the bottom part of Fig. 10. One destination (output) line forms a row of the LL/HL subbands and the other destination line forms a row of the LH/HH subbands.

The fractional wavelet filter approach shifts an input (vertical filter) area across the image in the vertical direction. The input area extends over the full horizontal width of the image. The vertical height of the input area is equal to the number of wavelet filter coefficients; for the considered Daubechies 9/7 filter, see Table I, the input area has a height of nine lines to accommodate the nine lowpass filter coefficients. The filter computes the horizontal wavelet coefficients on the fly. The vertical wavelet coefficients for each subband are computed iteratively through a set of fractional subband wavelet coefficients (fractions) $\boldsymbol{ll(i,j,k)}$, $\boldsymbol{lh(i,j,k)}$, $\boldsymbol{hl(i,j,k)}$, and $\boldsymbol{hh(i,j,k)}$. These fractions are later summed over the vertical filter index $\boldsymbol{j}$ to obtain the final wavelet coefficients. More specifically, the indices $\boldsymbol{i}$, $\boldsymbol{j}$, and $\boldsymbol{k}$ have the following meanings:

- $\boldsymbol{i}$ with $\boldsymbol{i = 0, 1, \ldots, N/2 - 1}$, gives the vertical position of the input (vertical filter) area as $\boldsymbol{2i}$. For each vertical filter area, i.e., each value of $\boldsymbol{i}$, nine input lines are read for lowpass filtering with nine filter coefficients. The filter moves up by two lines to achieve implicit vertical down sampling; thus, a total of $\boldsymbol{N/2 \cdot 9}$ lines have to be read. Note that there have to be $\boldsymbol{N/2}$ sets of final results, each set consisting of an LL, an HL, an LH, and an HH subband row.
- $\boldsymbol{j}$ with $\boldsymbol{j = -4, -3, \ldots, +4}$, specifies the current input line as $\boldsymbol{\ell = 2i + j}$, that is, $\boldsymbol{j}$ specifies the current line within the nine lines of the current input area. From the perspective of the filtering, $\boldsymbol{j}$ specifies the vertical wavelet filter coefficient.
- $\boldsymbol{k}$ with $\boldsymbol{k = 0, 1, \ldots, N/2 - 1}$, gives the horizontal position of the center of the wavelet filter as $\boldsymbol{2k}$ for the horizontal lowpass filtering and $\boldsymbol{2k+1}$ for the horizontal highpass filtering. That is, $\boldsymbol{k}$ specifies the current position of the horizontal fractional filter within the current input area.

For a given set of indices $\boldsymbol{i}$, $\boldsymbol{j}$, and $\boldsymbol{k}$, and with $\mathbf{s}$ containing the current input line $\boldsymbol{\ell = 2i + j}$, the fractional coefficients are computed as

$$\begin{aligned} \boldsymbol{ll(i,j,k)} &= \boldsymbol{l_j} \cdot \mathrm{convp}(\mathbf{s}, \mathbf{l}, \boldsymbol{2k}) \\ &= \boldsymbol{l_j} \cdot \sum_{\boldsymbol{m=-4}}^{\boldsymbol{4}} \boldsymbol{s_{2k+m} \cdot l_m} \qquad (45) \\ \boldsymbol{lh(i,j,k)} &= \boldsymbol{h_j} \cdot \mathrm{convp}(\mathbf{s}, \mathbf{l}, \boldsymbol{2k}) \\ &= \boldsymbol{h_j} \cdot \sum_{\boldsymbol{m=-4}}^{\boldsymbol{4}} \boldsymbol{s_{2k+m} \cdot l_m} \qquad (46) \\ \boldsymbol{hl(i,j,k)} &= \boldsymbol{l_j} \cdot \mathrm{convp}(\mathbf{s}, \mathbf{h}, \boldsymbol{2k+1}) \end{aligned}$$

TABLE VI
PSEUDO CODE OF FRACTIONAL WAVELET FILTER FOR THE FIRST LEVEL FORWARD WAVELET TRANSFORM.

```
1.  Allocate destination buffer LL_HL with N elements
2.  Allocate destination buffer LH_HH with N elements
3.  Allocate input buffer s with N elements
4.  For i = N/2 − 1, N/2 − 2, ..., 0:
5.      Initialize destination buffers:
6.        For m = 0, 1, ..., N − 1: LL_HL_m = 0, LH_HH_m = 0
7.      For j = −4, −3, ..., 4:
8.          line index ℓ = 2i + j
9.          Symmetric extension:
10.           If ℓ < 0: ℓ ← −ℓ
11.           else, if ℓ > N − 1: ℓ ← 2N − 2 − ℓ
12.         Read N values starting at position ℓ · N
13.           from flash memory into input buffer s
14.         For k = 0, 1, ..., N/2 − 1:
15.             L = convp(s, l, 2k)
16.             LL_HL_k += l_j · L    // update LL
17.             LH_HH_k += h_{j−1} · L    // update LH
18.             H = convp(s, h, 2k + 1)
19.             LL_HL_{k+N/2} += l_j · H    // update HL
20.             LH_HH_{k+N/2} += h_{j−1} · H    // update HH
21.     Write N elements of buffer LL_HL
22.       to flash memory starting at position i · N
23.     Write N elements of buffer LH_HH
24.       to flash memory starting at position (i + N/2) · N
```

$$= l_j \cdot \sum_{m=-3}^{3} s_{2k+1+m} \cdot h_m \quad (47)$$

$$hh(i,j,k) = h_j \cdot \text{convp}(\mathbf{s}, \mathbf{h}, 2k+1) = h_j \cdot \sum_{m=-3}^{3} s_{2k+1+m} \cdot h_m. \quad (48)$$

Note that the fractional approximations are computed through the convolutions with the analysis lowpass filter with the filter coefficients from Table IIIa) in Eqns. (45) and (46). The fractional details are obtained in Eqns. (47) and (48) through convolution with the analysis highpass filter.

Note that these fractional coefficients are intermediate results for the computation of two output lines. Only a single line of the input image is read at a time to keep the memory requirements low. All the lines of a given input area are read consecutively to achieve the final result. Throughout, the horizontal index k and vertical index i refer to the position of the wavelet filter coefficient in the final transformed image, i.e., the result image.

Recall from Section III-C that the image transform first requires to filter all pixels horizontally, and then to filter the intermediate result vertically. In summary, the horizontal coefficients are computed immediately, whereas the vertical coefficients are computed through successive updates, thereby achieving the significant memory savings.

The final coefficients are computed by iteratively summing the fractions over the vertical filter lines $j = -4, -3, \ldots, 4$. Specifically, we first initialize $LL(i,k) = LH(i,k) = HL(i,k) = HH(i,k) = 0 \;|_{\forall i,k}$. For $j = -4, -3, \ldots, 4$, the summing proceeds as:

$$LL(i,k) \mathrel{+}= ll(i,j,k) \quad (49)$$

$$LH(i,k) \mathrel{+}= lh(i,j,k) \quad (50)$$

$$HL(i,k) \mathrel{+}= hl(i,j,k) \quad (51)$$

$$HH(i,k) \mathrel{+}= hh(i,j,k). \quad (52)$$

TABLE VII
RAM MEMORY REQUIRED FOR THE FRACTIONAL WAVELET FILTER. THE NUMBER OF REQUIRED BYTES OF MEMORY ARE GIVEN FOR EACH WAVELET LEVEL lev FOR A 128×128 IMAGE FOR THE FLOATING-POINT FILTER AND FOR A 256×256 IMAGE FOR FIXED-POINT FILTER. THE DATA FORMAT FOR FIXED-POINT CALCULATION IS GIVEN IN THE TEXAS INSTRUMENTS $Qm.n$ NOTATION.

	floating-point		fixed-point		
N	lev	$Bytes$	lev	$Bytes$	Format
256		-	1	1280	$Q10.5$
128	1	1152	2	768	$Q11.4$
64	2	640	3	640	$Q12.3$
32	3	320	4	512	$Q13.2$
16	4	160	5	384	$Q14.1$

We summarize the computations for the floating-point version of the fractional wavelet filter for the first wavelet transform level in the pseudocode in Table VI. The evaluations of the fractions, i.e., Eqns (45)–(48), and final coefficients, i.e., Eqns. (49)–(52) take place in Lines 15–20. Notice that the subscript of the lowpass filter is j in Lines 16 and 19, whereas the subscript is $j-1$ for the highpass filter coefficient in Lines 17 and 20. This vertical displacement of the filter coefficients by one line in conjunction with advancing the filter area in steps of two lines (Lines 4 and 8) ensures that the centers of the lowpass and highpass filters align with alternate lines of the input image. Intuitively, each filter "takes in" one half of the input image, and they jointly "take in" the complete image.

Note that in the pseudocode of the fractional wavelet filter in Table VI, in Lines 8–20 the vertical filter coefficient j stays constant for updating all subband rows. The process of updating the destination lines in Lines 16, 17, 19, and 20 is repeated until the final subband coefficients have been computed.

Observe that when the filter input area moves up (Line 4 in Table VI), the last input line of the preceding filter area could be used as the first input line for the new filter area. Based on this observation we could slightly reduce the number of repetitive readings. For an $N \times N$ image, the number of line readings would reduce to $8 \cdot N/2$. For simplicity, we do not consider this slight optimization.

For evaluating the memory requirements, note that for the first transform level $lev = 1$, the input buffer $\mathbf{s}$ holds N original one Byte image samples, while each of the buffers $\mathbf{LL\_HL}$ and $\mathbf{LH\_HH}$ holds N float values of four Bytes each. For the higher transform levels $lev > 1$, the input buffer $\mathbf{s}$ has to hold float values from the preceding LL subband. In summary, the number of Bytes required for a wavelet transform with lev levels of an image with the dimension $N \times N$ pixels is

$$Bytes_{\text{float}} = \begin{cases} 9 \cdot N, & \text{lev } = 1 \\ 12 \cdot \frac{N}{lev}, & lev > 1. \end{cases}$$

Table VII gives the required memory in Bytes for the floating point transform of an 128×128 input image with 8 bit grayscale info per pixel for different transform levels.

C. Fixed-Point Filter

The fractional fixed-point filter can be realized by first transforming the real wavelet coefficients to the $Q0.15$ for-

mat, see Table III. The second requirement is that for all add and multiply operations the exponent has to be taken into account. For an add operation, both operands must have the same exponent. A multiply operation requires a result exponent given by the sum of the input exponents. Both operations can be supported by an exponent change function that corresponds to a left or right bit-shift. Aside from these special considerations, the fixed-point filter works similarly as the floating-point filter.

As the number range of the wavelet coefficients increases from level to level, the output data format has to be enlarged from level to level. The study [16] on the required range reports that the data formats in Table VII are sufficient. (Note that there is a general difference when the usage of fixed-point numbers for wavelet transforms is discussed in the literature. Sometimes, the fixed-point numbers are only used for the representation of the wavelet coefficients, as is done in [16]. In this work, when we discuss a fixed-point algorithm, we include the coefficient representation as well as the internal calculations.) For the first wavelet transform level, the input data format is $\boldsymbol{Q15.0}$, as the image samples are integer numbers. Note that the wavelet coefficients $\boldsymbol{L}$ and $\boldsymbol{H}$ computed in Lines 15 and 18 in Table VI are already computed in the data format of the final level (even if they may need a smaller range than the subband coefficients).

For $\boldsymbol{lev = 1}$, each original image sample in buffer $\mathbf{s}$ has one Byte, while each INT16 value in the **LL_HL** and **LH_HH** buffers has two Bytes. For the subsequent transform levels $\boldsymbol{lev > 1}$, the input buffer $\mathbf{s}$ holds the two Byte INT16 coefficients from the preceding LL subband. Thus, the memory requirements for the fixed-point filter are

$$\boldsymbol{Bytes}_{\text{fixed}} = \begin{cases} \mathbf{5} \cdot \boldsymbol{N}, & \boldsymbol{lev = 1} \\ \mathbf{6} \cdot \frac{\boldsymbol{N}}{\boldsymbol{lev}}, & \boldsymbol{lev > 1.} \end{cases}$$

and are given in Table VII for the levels 1–5 for an 256 × 256 image.

D. Inclusion of the lifting scheme

The computing time for the fractional wavelet filter can be reduced by incorporating the lifting scheme, which was outlined in Section III-B. The lifting scheme cannot be applied to the vertical filtering technique of the fractional filter and also not to the horizontal transform of the first level, as the input lines of this level have to use an integer buffer array to avoid exceeding the memory. The lifting scheme allows to compute the one-dimensional transform in place, that is, there is only one buffer for the input and output values, whereby the buffer must have the appropriate size to contain the final wavelet coefficients. It thus can be applied to the higher levels of the horizontal transform, as the input lines for these levels take variables with the larger data format (and not only 8 bit unsigned char as for the first level input). More specifically, the lifting scheme is employed for the computation of the convolution in Lines 15 and 18 in Table VI.

VI. Case Studies: Performance Evaluation of Fractional Wavelet Filter

In this section we report two case studies that employ the fractional wavelet filter and shed light on its performance characteristics. In the first study we emulated the 16-bit fixed-point arithmetic of a microcontroller on a PC to assess the image quality. In the second study, we built a sensor node and conducted time measurements of the fractional wavelet filter. For a detailed performance evaluation that comprehensively evaluates the resource requirements for the fractional wavelet filter and its image quality we refer to [27].

A. Image Quality

For the quality evaluation of the fixed-point wavelet filter, we have selected twelve test images from the *Waterloo Repertoire* (http://links.uwaterloo.ca) and the *standard image data base* (http://sipi.usc.edu/database). The images were converted to an integer data format with the range of $[\mathbf{-128,\ 127}]$ (corresponding to 8 bits = 1 Byte per pixel) using the *convert* command of the software suite *ImageMagick* (available at http://www.imagemagick.org) and the software *Octave*.

In the image quality evaluation we compare the original $\boldsymbol{N \times N}$ image $\boldsymbol{f(j,k),\ j,k = 1,\ldots,N}$, with the reconstructed image $\boldsymbol{g(j,k),\ j,k = 1,\ldots,N}$. The reconstructed image is generated through a forward wavelet transform using the fractional fixed-point wavelet filter of the original image followed by an inverse wavelet transform (using the standard floating point inverse transform). We compute the *mean squared error* (MSE) for the compared monochrome image matrices $\boldsymbol{f}$ and $\boldsymbol{g}$:

$$\text{MSE} = \frac{\mathbf{1}}{\boldsymbol{N^2}} \sum_{\forall j,k} \left(\boldsymbol{f(j,k) - g(j,k)}\right)^{\mathbf{2}}. \tag{53}$$

The PSNR in decibels (dB) is calculated as

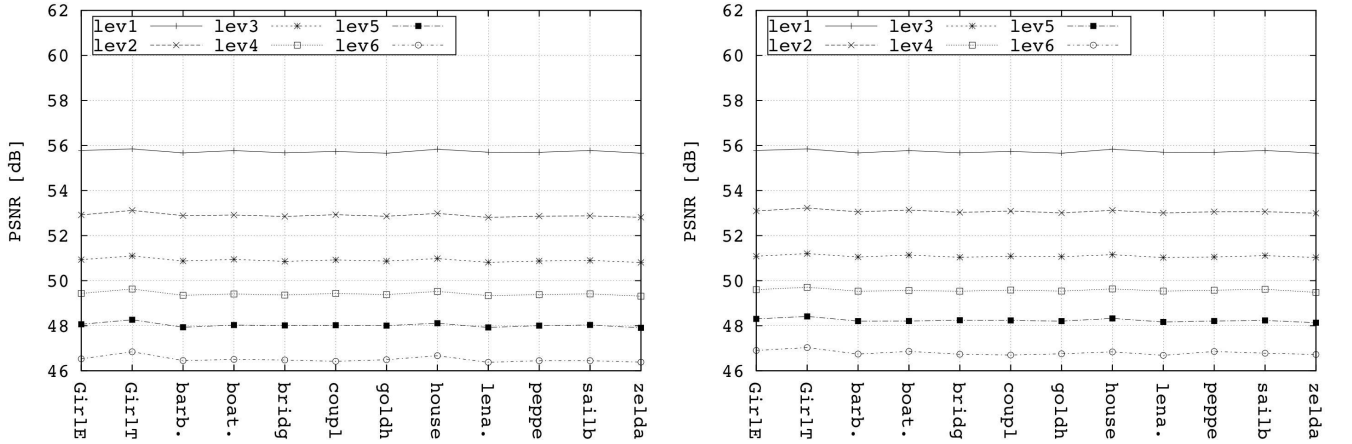
$$\text{PSNR} = \mathbf{10} \cdot \log_{\mathbf{10}} \frac{\mathbf{255^2}}{\text{MSE}}, \tag{54}$$

Image degradations with a PSNR of 40 dB or higher are typically nearly invisible for human observers [44].

For each image we computed six wavelet levels. We did not observe any visible quality degradation. However, the forward wavelet transform with the fractional wavelet filter is not lossless. In Figure 11 we plot the PSNR values for each of the maximum wavelet levels `lev1` through `lev6` for $\mathbf{256 \times 256}$ pixel grayscale images. The computed data format of the wavelet coefficients at the first level is in the $\boldsymbol{Q10.5}$ data format, which requires a division by $\mathbf{2^5}$ to obtain the original data. For each higher level, the format switches to the next larger range, which here would be the $\boldsymbol{Q11.4}$ format.

For an improved version of the fractional wavelet filter we integrated the lifting scheme. Figure 11b) illustrates that the lifting scheme gives the same or even slightly better image qualities. The cost of the lifting scheme is the more complex and longer source code (which we freely provide), which adds on to the required program memory of the microcontroller. Computation time savings are achieved for the levels 2–6. The very slight improvements in image quality are due to the reduced number of computations, which lead to less loss of precision.

The reduction in image quality observed in Fig. 11 is negligible when the transformed image is compressed with a lossy wavelet compression algorithm. For demonstration



a) without lifting scheme b) with lifting scheme

Fig. 11. PSNR image quality a) without lifting and b) with lifting using the fractional fixed-point wavelet filter. The first level gives very high PSNR values as the transform input values are integers. The quality loss is negligible when the fractional wavelet filter is employed in conjunction with a lossy compression algorithm which neglects the least significant bits. While the lifting scheme gives substantial savings in the computation, the qualities are nearly the same or even slightly better than with the standard convolution technique.

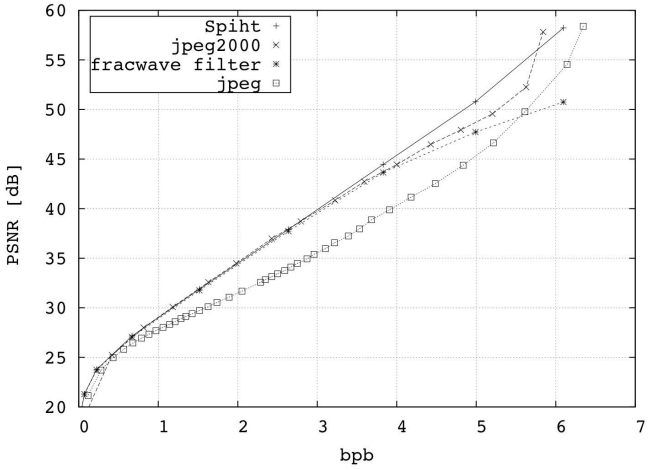


Fig. 12. The introduced wavelet filter combined with a low-memory image coder gives nearly the same performance as JPEG2000, which reflects the state of the art in image compression. While most implementations of JPEG2000 require a personal computer, the combination of fractional wavelet filter and image coder runs on a small 16 bit micro controller with less than 2 kByte of random access memory. Small wireless sensor nodes can thus perform image compression when extended with a flash memory, a small camera with serial interface, and a software update.

we combine the fractional wavelet filter with a low-memory version of the SPIHT coder [23]. We evaluate the compression performance for the *bridge* image in terms of the PSNR as a function of the compression ratio in bits per byte (i.e., per input pixel) denoted by `bpb`. The PSNR is for the forward wavelet transformed, compressed, and subsequently uncompressed and backward transformed image compared to the original image and is denoted by `fracwave filter` in Fig. 12. We compare with state-of-the-art compression schemes, namely `Spiht` [14] obtained with the Windows implementation from Said et al. http://www.cipr.rpi.edu/research/SPIHT/spiht3.html, as well as `jpeg` and `jpeg2000` obtained with the JPEG and JPEG2000 implementations from the *jasper* project http://www.ece.uvic.ca/~mdadams/jasper. These comparison benchmarks combine the image transform and image coding and are designed to run on personal computers with abundant memory.

We observe from the figure that the wavelet techniques Spiht, JPEG2000, and our fractional wavelet filter based scheme outperform the general JPEG standard. Importantly, we observe that the fractional wavelet filter based approach achieves state-of-the-art image compression that gives essentially the same PSNR image quality as JPEG2000 and SPIHT for the higher compression rates (i.e., smaller bpb values). For the lower compression rates (larger bpb values), the fractional wavelet filter approach gives slightly lower image quality due to the loss in precision from the fixed-point arithmetic. However, for sensor networks, generally a high compression (small bpb) is required due to the limited transmission energy and bandwidth.

B. Time Measurements

For timing measurements of the fractional wavelet filter, we built a sensor node from the *Microchip dsPIC30F4013*, i.e., a 16 bit digital signal (micro) controller with 2 kByte of RAM, the camera module C328-7640 with an universal asynchronous receiver/transmitter (UART) interface (available at http://www.comedia.com.hk), and an external 64 MByte multimedia card (MMC) as flash memory, connected to the controller through a serial peripheral interface (SPI) bus. The data of the MMC-card is accessed through the filesystem [5]. Camera and MMC-card both can be connected to any controller with UART and SPI ports. The system is designed to capture still images.

For the time measurements the forward fractional wavelet filter without lifting scheme was employed on the camera sensor to perform a six-level transform for a $\mathbf{256 \times 256 \times 8}$ image. The reported times reflect the means of 20 measurements, whereby the values are nearly constant across the different measurements with the exception of the write access times. The variability in the write time is due to the flash memory media, which has to perform internal operations before a block of data can be written.

In additional time measurements, we computed a six-level transform of an $\mathbf{128 \times 128 \times 8}$ image using the floating

TABLE VIII
READ AND WRITE ACCESS TIMES FOR THE MULTIMEDIA (FLASH MEMORY) CARD AND COMPUTATION TIMES IN SECONDS FOR PERFORMING THE FRACTIONAL WAVELET FILTER TRANSFORM FOR SIX LEVELS FOR AN $\mathbf{256 \times 256 \times 8}$ IMAGE ON A 16-BIT MICROCONTROLLER. THE LARGEST TIME PROPORTION IS NEEDED FOR COMPUTATION.

$T_{\mathbf{read}}$	$T_{\mathbf{write}}$	$T_{\mathbf{compute}}$	$T_{\mathbf{total}}$
2.84	1.09	7.72	11.64

point version of the fractional wavelet filter. The compute times were about $T_{\mathbf{compute}} = \mathbf{14.2}$ s, i.e., roughly twice the compute times with fixed-point arithmetic for the four times larger $\mathbf{256 \times 256}$ image in Table VIII. This result indicates that fixed-point arithmetic (see Section IV) achieves significantly faster image transforms than floating point arithmetic, which is implemented through heavy compiler support on the microcontroller.

Interestingly, the flash memory is not the bottleneck of the fractional wavelet filter, even if there is large redundancy in the read process, as the rows are read repetitively by the fractional filter. More specifically, for computing the final coefficients of two output lines a "filter area" spanning nine lines is read. For the computation of the next two output lines, this filter area moves up by two positions. Thus, there is a large overlap with the previous filter area. Nevertheless, this underlying strategy of the fractional wavelet filter—to reduce the memory requirements by introducing some replication in the read process—is well suited for memory constrained sensor nodes with attached flash memory.

We briefly note that a detailed computational complexity analysis [27] revealed that the fractional wavelet filter without the lifting scheme requires about 2.9 times more add operations and 3.25 times more multiply operations than the classical convolution approach (which requires memory for $\mathbf{2}N^{\mathbf{2}}$ pixels with four Bytes per pixel for floating-point and two Bytes per pixel for fixed-point computations). The fractional wavelet filter thus achieves the dramatic reduction in required memory at the expense of somewhat increased number of computations, which in turn affect computation times and energy consumption.

VII. SUMMARY

This tutorial introduced communications and networking generalists without specific background in image signal processing to low-memory wavelet transform techniques. The image wavelet transform is highly useful in wireless sensor networks for preprocessing the image data gathered by the single nodes for compression or feature extraction. However, traditionally the wavelet transform has not been employed on the typical low-complexity sensor nodes, as the required computations and memory exceeded the sensor node resources. This tutorial presented techniques that allow for transforming images on in-network sensors with very small random access memory (RAM).

This tutorial first introduced elementary one-dimensional and two-dimensional wavelet transforms. Then, the computation of the wavelet transform with fixed-point arithmetic on microcontrollers was explained. Building on these foundations, the tutorial explained the fractional wavelet transform which required only 1.5 kByte of RAM to transform a grayscale $\mathbf{256 \times 256}$ image. The techniques taught in this tutorial, thus make the image wavelet transform feasible for sensor networks formed from low-complexity sensor nodes.

The low-memory transform techniques are not lossless. However, the performance evaluation illustrated that the loss is typically not visible, as PSNR values higher than 40 dB are obtained. Combining the low-memory wavelet transform techniques with a low-memory image wavelet coder achieved image compression competitive with state of the art JPEG2000 compression.

This tutorial and the accompanying freely available C-code provide a starting point for communications and networking researchers and practitioners to employ modern wavelet techniques in sensor networks. With the techniques covered in this tutorial, a typical sensor node can be upgraded to a camera sensor node by attaching a standard flash memory (standard SD card), a small low-cost camera, and a software update.

REFERENCES

[1] H. Karl and A. Willig, *Protocols and Architectures for Wireless Sensor Networks*. John Wiley & Sons, 2005.

[2] A. Seema and M. Reisslein, "Towards Efficient Wireless Video Sensor Networks: A Survey of Existing Node Architectures and Proposal for A Flexi-WVSNP Design," *IEEE Commun. Surveys & Tutorials*, to be published, 2011.

[3] I. F. Akyildiz, T. Melodia, and K. R. Chowdhury, "A survey on wireless multimedia sensor networks," *Computer Networks*, vol. 51, no. 4, pp. 921–960, Mar. 2007.

[4] S. Misra, M. Reisslein, and G. Xue, "A survey of multimedia streaming in wireless sensor networks," *IEEE Commun. Surveys and Tutorials*, vol. 10, no. 4, pp. 18–39, Fourth Quarter 2008.

[5] S. Lehmann, S. Rein, and C. Gühmann, "External flash filesystem for sensor nodes with sparse resources," in *Proc. ACM Mobile Multimedia Communications Conference (Mobimedia)*, July 2008.

[6] A. Leventhal, "Flash storage memory," *Communications ACM*, vol. 51, no. 7, pp. 47–51, July 2008.

[7] G. Mathur, P. Desnoyers, P. Chukiu, D. Ganesan, and P. Shenoy, "Ultra-low power data storage for sensor networks," *ACM Trans. Sensor Netw.*, vol. 5, no. 4, pp. 1–34, Nov. 2009.

[8] D. Lee and S. Dey, "Adaptive and energy efficient wavelet image compression for mobile multimedia data services," in *Proc. IEEE International Conference on Communications (ICC)*, 2002.

[9] T. Yan, D. Ganesan, and R. Manmatha, "Distributed image search in camera sensor networks," in *Proc. 6th ACM Conference on Embedded Network Sensor Systems (SenSys)*, 2008, pp. 155–168.

[10] P. Chen, P. Ahammad, C. Boyer, S.-I. Huang, L. Lin, E. Lobaton, M. Meingast, S. Oh, S. Wang, P. Yan, A. Yang, C. Yeo, L.-C. Chang, J. Tygar, and S. Sastry, "CITRIC: A low-bandwidth wireless camera network platform," in *Proc. Second ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC)*, Sept. 2008, pp. 1–10.

[11] J. Polastre, R. Szewczyk, and D. Culler, "Telos: enabling ultra-low power wireless research," in *Proc. Fourth Int. IEEE/ACM Symposium on Information Processing in Sensor Networks (IPSN)*, Apr. 2005, pp. 364–369.

[12] B. Rinner and W. Wolf, "Toward pervasive smart camera networks," in *Multi-Camera Networks: Principles and Applications*, H. Aghajan and A. Cavallaro, Eds. Academic Press, 2009, pp. 483–496.

[13] J. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," *IEEE Trans. Signal Process.*, vol. 41, no. 12, pp. 3445–3462, Dec. 1993.

[14] A. Said and W. Pearlman, "A new, fast, and efficient image codec based on set partitioning in hierarchical trees," *IEEE Trans. Circuits Sys. Video Technol.*, vol. 6, no. 3, pp. 243–250, June 1996.

[15] S. Rein, S. Lehmann, and C.Gühmann, "Fractional wavelet filter for camera sensor node with external flash and extremely little RAM," in *Proc. ACM Mobile Multimedia Communications Conference (Mobimedia '08)*, July 2008.

[16] T. Fry and S. Hauck, "SPIHT image compression on FPGAs," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 15, no. 9, pp. 1138–1147, Sept. 2005.

[17] K.-C. Hung, Y.-J. Huang, T.-K. Truong, and C.-M. Wang, "FPGA implementation for 2D discrete wavelet transform," *Electronics Letters*, vol. 34, no. 7, pp. 639–640, Apr. 1998.
[18] J. Chilo and T. Lindblad, "Hardware implementation of 1D wavelet transform on an FPGA for infrasound signal classification," *IEEE Trans Nucl. Sci.*, vol. 55, no. 1, pp. 9–13, Feb. 2008.
[19] S. Ismail, A. Salama, and M. Abu-ElYazeed, "FPGA implementation of an efficient 3D-WT temporal decomposition algorithm for video compression," in *Proc. IEEE International Symposium on Signal Processing and Information Technology*, Dec. 2007, pp. 154–159.
[20] S. Rein, C. Gühmann, and F. Fitzek, "Sensor networks for distributive computing," in *Mobile Phone Programming*, F. Fitzek and F. Reichert, Eds. Springer, 2007, pp. 397–409.
[21] A. Ciancio, S. Pattem, A. Ortega, and B. Krishnamachari, "Energy-efficient data representation and routing for wireless sensor networks based on a distributed wavelet compression algorithm," in *Proc. Int. Conference on Information Processing in Sensor Networks (IPSN)*, 2006, pp. 309–316.
[22] H. Wu and A. Abouzeid, "Energy efficient distributed image compression in resource-constrained multihop wireless networks," *Computer Communications*, vol. 28, no. 14, pp. 1658–1668, Sept. 2005.
[23] S. Rein, S. Lehmann, and C. Gühmann, "Wavelet image two line coder for wireless sensor node with extremely little RAM," in *Proc. IEEE Data Compression Conference (DCC)*, Snowbird, UT, Mar. 2009, pp. 252–261.
[24] D.-U. Lee, H. Kim, M. Rahimi, D. Estrin, and D. Villasenor, "Energy-efficient image compression for resource-constrained platforms," *IEEE Trans. Image Process.*, vol. 18, no. 9, pp. 2100–2113, Sept. 2009.
[25] C. Chrysafis and A. Ortega, "Line-based, reduced memory, wavelet image compression," *IEEE Trans. Image Process.*, vol. 9, no. 3, pp. 378–389, Mar. 2000.
[26] J. Oliver and M. Malumbres, "On the design of fast wavelet transform algorithms with low memory requirements," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 18, no. 2, pp. 237–248, Feb. 2008.
[27] S. Rein and M. Reisslein, "Performance evaluation of the fractional wavelet filter: A low-memory image wavelet transform for multimedia sensor networks," *Ad Hoc Networks*, 2010, doi:10.1016/j.adhoc.2010.08.004.
[28] Y. Bao and C. Kuo, "Design of wavelet-based image codec in memory-constrained environment," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 11, no. 5, pp. 642–650, May 2001.
[29] C.-H. Yang, J.-C. Wang, J.-F. Wang, and C.-W. Chang, "A block-based architecture for lifting scheme discrete wavelet transform," *IEICE Trans. Fundamentals*, vol. E90-A, no. 5, pp. 1062–1071, May 2007.
[30] J. Oliver and M. Malumbres, "Low-complexity multiresolution image compression using wavelet lower trees," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 16, no. 11, pp. 1437–1444, Nov. 2006.
[31] I. Daubechies, *Ten Lectures on Wavelets*. SIAM, 1992.
[32] S. Mallat, *A Wavelet Tour of Signal Processing*. Academic Press, 1999.
[33] B. Usevitch, "A tutorial on modern lossy wavelet image compression: Foundations of JPEG2000," *IEEE Signal Process. Mag.*, vol. 18, no. 5, pp. 22–35, Sept. 2001.
[34] D. Taubman and M. Marcellin, *JPEG2000—Image compression, fundamentals, standard and practice*. Kluwer Academic Publishers, 2004.
[35] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," *J. Fourier Analysis and Applications*, vol. 4, no. 3, pp. 247–269, May 1998.
[36] A. Jensen and A. la Cour-Harbo, *Ripples in Mathematics - The Discrete Wavelet Transform*. Springer, 2001.
[37] W. Sweldens, "The lifting scheme: A construction of second generation wavelets," *SIAM J. Math. Anal.*, vol. 29, no. 2, pp. 511–546, 1997.
[38] A. Cohen, I. Daubechies, and J. Feauveau, "Bi-orthogonal bases of compactly supported wavelets," *Comm. Pure & Appl. Math*, vol. 45, no. 5, pp. 485–560, 1992.
[39] A. Haghparast, H. Penttinen, and A. Huovilainen, "Fixed-point algorithm development," Laboratory of Acoustics and Audio Signal Processing. Helsinki University of Technology, April 2006.
[40] "Fixed Point Arithmetic on the ARM," in *Application Note 33*. ARM Advanced RISC Machines Ltd (ARM), September 1996.
[41] D. Ombres, "C and C++ extensions simplify fixed-point DSP programming," *EDN Magazine*, Oct. 1996.
[42] *dsPIC30F Data Sheet*, Microchip Technology Inc., 2002.
[43] R. Yates, "Fixed-point arithmetic: An introduction." Digital Signal Labs, August 2007.
[44] K. Rao and P. Yip, Eds., *The Transform and Data Compression Handbook*. CRC press, 2001.

Stephan Rein studied electrical and telecommunications engineering at RWTH Aachen University and Technical University (TU) Berlin, where he received the Dipl.-Ing. degree in December 2003 and the Ph.D. degree in January 2010. He was a research scholar at Arizona University in 2003, where he conducted research on voice quality evaluation and developed an audio content search machine. From February 2004 to March 2009 he was with the Wavelet Application Group at TU Berlin developing text and image compression algorithms for mobile phones and sensor networks. Since July 2009 he is with the Telecommunication Networks Group at TU Berlin, where he is currently working on multimedia delivery to mobile devices.

Martin Reisslein is an Associate Professor in the School of Electrical, Computer, and Energy Engineering at Arizona State University (ASU), Tempe. He received the Dipl.-Ing. (FH) degree from the Fachhochschule Dieburg, Germany, in 1994, and the M.S.E. degree from the University of Pennsylvania, Philadelphia, in 1996. Both in electrical engineering. He received his Ph.D. in systems engineering from the University of Pennsylvania in 1998. During the academic year 1994–1995 he visited the University of Pennsylvania as a Fulbright scholar. From July 1998 through October 2000 he was a scientist with the German National Research Center for Information Technology (GMD FOKUS), Berlin and lecturer at the Technical University Berlin. He currently serves as Associate Editor for the *IEEE/ACM Transactions on Networking* and for *Computer Networks*. He maintains an extensive library of video traces for network performance evaluation, including frame size traces of MPEG-4 and H.264 encoded video, at `http://trace.eas.asu.edu`. His research interests are in the areas of video traffic characterization, wireless networking, optical networking, and engineering education.