Algorithm and VLSI Architecture for High Performance Adaptive Video Scaling

Arun Raghupathy, *Member, IEEE*, Nitin Chandrachoodan, and K. J. Ray Liu, *Fellow, IEEE*

***Abstract*—We propose an efficient high-performance scaling algorithm based on the oriented polynomial image model. We develop a simple classification scheme that classifies the region around a pixel as an oriented or nonoriented block. Based on this classification, a nonlinear oriented interpolation is performed to obtain high quality video scaling. In addition, we also propose a generalization that can perform scaling for arbitrary scaling factors. Based on this algorithm, we develop an efficient architecture for image scaling. Specifically, we consider an architecture for scaling a Quarter Common Intermediate Format (QCIF) image to 4CIF format. We show the feasibility of the architecture by describing the various computation units in a hardware description language (Verilog) and synthesizing the design into a netlist of gates. The synthesis results show that an application specific integrated circuit (ASIC) design which meets the throughput requirements can be built with a reasonable silicon area.**

***Index Terms*—Adaptive scaling, oriented polynomial interpolation, video zoom, VLSI architecture.**

I. Introduction

IN VIDEO applications, the raw data rates involved are extremely high. This implies that a very large bandwidth is required to transmit video signals. A number of widely accepted video compression standards (such as JPEG, MPEG, H.263) have been developed in order to reduce the data rate. Inspite of these developments, in applications in which the available bandwidth is limited, the image size is restricted. For example, in PC video-telephony applications QCIF/CIF is the standard format. Obviously, a larger image size format will improve the perceptual quality. But, this quality comes at the cost of increased data rate (and, hence, bandwidth). A high-performance scaling algorithm that can scale an image without introducing much distortion will allow us to transmit image data using a small image size format while maintaining perceptual quality. Another motivation for developing a good scaling algorithm is to enable display of lower resolution images on higher resolution monitors. For example, if an NTSC (National Television Standards Committee) format picture is to be displayed on HDTV monitors. Image scaling is also sometimes referred to as spatial up-conversion.

Scaling techniques that are commonly used include simple pixel replication, bilinear interpolation and cubic convolutional interpolation. These techniques can be considered as separable FIR filters [1] of two, three and seven taps, respectively, in each dimension. However, when the images contain sharp edges or thin lines, these techniques cause visible effects such as jagged or blurred edges.

Nonlinear model based interpolation techniques have been proposed in [1]–[4]. Martinez and Lim [2] approximated edges in interlaced images by a line shift model and interpolated each point in the missing line by averaging two points in adjacent lines separated by the estimated shift parameter. The technique proposed by Jensen and Anastassiou [3] used a bilevel step-edge model to approximate an edge block and assigned one of these two values to each pixel. Salonen [4] proposed an edge and motion adaptive interpolation for interlaced video. The edge orientations are obtained using compass operators and interpolation is performed in the dominant direction. Wang and Mitra [1] proposed an image model based on oriented polynomials that can model various types of edges, including both step and ramp-type edges and thin lines. Using this model, an image scaling technique was proposed [1] that performed very well. However, this technique was considered difficult to implement in practical systems because of the complexity of the classification and the filtering.

In this paper, our focus is to develop an efficient algorithm based on the oriented polynomial image model to enable practical implementation. We develop a simplified classification scheme that classifies the region around a pixel as an oriented or nonoriented block. In addition, we also propose a generalization that can perform scaling for arbitrary scaling factors. This algorithm can be efficiently implemented in software or hardware. Based on the algorithm, we develop efficient architectures for the various computation units required for scaling. Starting with a simple system architecture we discuss how applying transformations to the data flow graph can systematically lead us to an efficient architecture that meets the throughput requirements. Then the results of synthesis based on a Verilog description of the architecture are discussed. Results indicate that an ASIC design that can scale a QCIF video sequence to 4CIF format at 30 frames/s is possible. We also suggest how this architecture can be extended to larger image input formats (which correspond to larger throughput requirements) by applying speed-up techniques.

Manuscript received September 29, 2000; revised February 11, 2002. This work was supported in part by the National Science Foundation NYI Award MIP9457397. The associate editor coordinating the review of this paper and approving it for publication was Dr. Sethuraman Panchanathan.

A. Raghupathy was with the Department of Electrical and Computer Engineering and Institute for Systems Research, University of Maryland, College Park, MD 20742 USA. He is now with Qualcomm, Inc., San Diego, CA 92121 USA (e-mail: arunr@qualcomm.com).

N. Chandrachoodan was with the Department of Electrical and Computer Engineering and Institute for Systems Research, University of Maryland, College Park, MD 20742 USA. He is now with the Intel Software Radio Laboratory, Indian Institute of Technology, Madras, India.

K. J. R. Liu is with the Department of Electrical and Computer Engineering and Institute for Systems Research, University of Maryland, College Park, MD 20742 USA (e-mail: kjrliu@eng.umd.edu).

Digital Object Identifier 10.1109/TMM.2003.813282

1520-9210/03$17.00 © 2003 IEEE

The paper is organized as follows. In Section II, we discuss the details of our algorithm. Then, in Section III, we discuss an architecture for an implementation of the algorithm in Section II. We also discuss the synthesis results of a Verilog description of the system. We show that the throughput required to perform a video scaling from QCIF to 4CIF can be met. Finally, in Section IV, we present the conclusions.

II. Description of Our Algorithm

Image scaling is the process of converting an image from its original dimension to a new dimension. The success of this process is based on our ability to determine the values of missing pixels located between the given original pixels. Normally, the process is based on fitting a continuous function through the discrete input samples.

The image scaling system is designed to take an input image in the YC_bC_r color format as specified by CCIR Recommendation 601-1 and performs arbitrary size scaling. The input image has one 8-bit luminance component Y, and two 8-bit chrominance components C_b and C_r. We note that the scaling system can be applied equally well to 8-bit gray scale images by simply performing the luminance operation on the image while skipping the chrominance operations. All our algorithms have been implemented and tested on the digital YC_bC_r signal with 8-bit depth. In the discussion that follows, we refer to the chrominance components as the U and V components.

It is well known that the human visual system (HVS) is not very sensitive to the chrominance components. Indeed, this fact has been used in most video coding standards where the source chrominance components are subsampled before encoding. In our scaling system, we also take advantage of this fact and simply apply bilinear interpolation to scale the chrominance components. On the other hand, the HVS is highly sensitive to the luminance component. Thus, an adaptive interpolation system is used to scale the luminance component. The adaptive interpolation system used to process the luminance component is composed of three modules namely classification module, oriented interpolation module and bilinear interpolation module. Basically, the adaptive interpolation system performs classification for each output pixel using the input pixels to determine whether the output pixel is located in a region with a distinct orientation (edge) or not. Afterward, appropriate filtering (interpolation) will be carried out using the input pixels to produce the output pixel. For pixels located along edges, directional filtering are performed along the edges rather than across it. This ensures that the visually annoying "staircase effect" does not appear along edges.

We found that in color images it is sufficient to apply bilinear interpolation for the chrominance (U and V) components. An adaptive interpolation is used for the luminance (Y) component. The classification module determines whether the Y component's neighborhood is oriented or nonoriented. Based on this result, the desired resampled Y component is interpolated using the appropriate interpolation module.

The terminology that we use to explain the algorithm is defined below. For each input Y component, we define it's associated block and neighborhood as shown in Fig. 1, where each dot

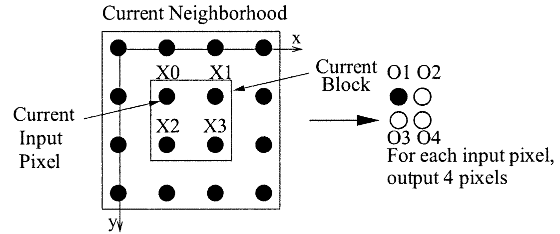


Fig. 1. Input/output pixel relationship.

represents the Y component of the pixel from the input image. A Y pixel's block consists of the area enclosed by the square formed from four adjacent pixels, with the current input Y pixel being the top-left vertex (i.e., the inner square in Fig. 1). The neighborhood of the Y component of the pixel is defined as the set of Y pixels that are enclosed by the outer square in Fig. 1. Let $f(x,y)$ denote the luminance value of the digital image at location (x,y). Also, let N_b denote the set of Y pixels in the neighborhood of the current input pixel b. In the following, when we refer to a pixel we mean the Y component of the pixel unless we explicitly specify otherwise. The U and V components of the pixel will be explicitly referred to as U pixels and V pixels, respectively. In Sections II-A and II-B, we consider two scenarios of image expansion.

A. *Image Expansion by Powers of Two*

We will describe the algorithmic implementation of expansion by two in each dimension. In order to expand by a factor of 2^k, we can simply apply the factor of two expansion k times. The interpolation process for a scaling by two in each dimension proceeds as follows. Since we are doubling the dimensions of the input image, we will produce four output pixels for each input pixel. The chrominance components U and V are bilinearly interpolated except for the last row or column where pixel replication is applied. We note here that for input Y pixels that are located on the border of the image where a valid neighborhood cannot be defined due to the lack of pixels, we do not use adaptive interpolation. Rather, we simply use bilinear interpolation except for the Y pixel located on the last row or last column where nearest neighbor interpolation is used due to nonexistence of a right/bottom adjacent Y pixel component.

In the following, we describe the adaptive interpolation procedure for those pixels that have valid neighborhoods. For each input pixel, we first classify whether its block is located in an oriented or nonoriented region by examining its neighborhood. Intuitively, an oriented block possesses certain directional pattern that often occurs in regions containing strong edges while a nonoriented block does not. Therefore, directional or oriented interpolation is used to interpolate the missing samples inside oriented blocks while bilinear interpolation is used for nonoriented blocks. Each oriented block is further classified by its dominant direction, which is quantized into eight major directions to reduce complexity and maintain robustness. The classification process is shown in Fig. 2.

As a first test, we compute $L_{\max}$ and $L_{\min}$, which is defined as the maximum and the minimum luminance value of all the pixels in the current neighborhood. If the difference between $L_{\max}$ and $L_{\min}$ is less than 25, then the block is classified

Input: Neighborhood of a pixel in the image.
Output: Classification of the pixel as either `oriented` or `nonoriented` so that oriented or bilinear interpolation can be applied.

```
1: Compute maximum (L_max) and minimum (L_min) luminance in the neighborhood of the pixel.
2: if L_max − L_min < 25 then
3:    return nonoriented
4: else
5:    for all pixels in neighborhood do
6:       if orientation already computed then
7:          retrieve orientation from RAM
8:       else
9:          compute and quantize the orientation, and store it in RAM
10:      end if
11:   end for
12:   compute the histogram of the orientation to obtain the most frequently occuring orientation.
13:   H_max ← number of times most frequent orientation occurs in neighborhood
14:   if H_max > 6 then
15:      return oriented
16:   else
17:      return nonoriented
18:   end if
19: end if
```

Fig. 2. Pixel classification.

as nonoriented. Otherwise, we proceed with the classification process. The work in [1] suggests that a block can be classified as "constant" based on the idea of "just noticeable difference", which is a piecewise linear function of the average background intensity in the neighborhood of the block. Since implementing such a function in hardware is complex, the possibility of a fixed threshold was considered. Based on experiments with test data, a value of 25 was chosen for the threshold. Further, as noted in Section III, the experiments also showed that this step can be omitted without significant loss in visual quality.

In the next step, we need to compute the quantized orientation of each pixel in the current neighborhood. We set up a table in RAM whose dimensions are equal to the input image. Each table entry contains the computed quantized orientation for the corresponding pixel of the input image at that location. All entries of the table are initialized with a special flag indicating that no quantized orientation has been computed for this pixel and as the classification process proceeds, we will gradually replace the flag with quantized orientation values. Since the classification process needs to be carried out for each pixel and adjacent pixels share common pixels in their neighborhood, we can eliminate redundant computation of the quantized orientation by organizing our information in this way. Therefore, we see that we will only compute the quantized orientation O of the pixels in our neighborhood, if it is not available already. Now, the quantized orientation O is computed as follows:

$$\begin{aligned} f_x &= f(i+1,j+1) + 2\times f(i+1,j) + f(i+1,j-1) \\ &\quad - f(i-1,j+1) - 2\times f(i-1,j) - f(i-1,j-1) \\ f_y &= f(i+1,j+1) + 2\times f(i,j+1) + f(i-1,j+1) \\ &\quad - f(i+1,j-1) - 2\times f(i,j-1) - f(i-1,j-1) \\ \theta &= -57.2958 \times \tan^{-1}(f_x/f_y) \bmod 180^\circ \\ O &= \lfloor \frac{\theta - 11.25}{22.5} \rfloor + 1 \bmod 7 \end{aligned} \tag{1}$$

where f_x and f_y are the gradients of $f(x,y)$ in the x and y directions, respectively, computed at (i,j), θ is the full precision orientation and O is the quantized orientation.

Once we have computed the quantized orientation O of each pixel in the neighborhood, we will calculate the histogram of quantized orientation. Subsequently, we find the dominant quantized orientation from the histogram. Let $H_{\max}$ denote the number of pixels that has the dominant quantized orientation in the neighborhood. If $H_{\max} > 6$ then we declare the current block as oriented with direction set to the dominant quantized orientation. Otherwise, the block is said to be a nonoriented block. The threshold value for deciding if the block is oriented is based on the fact that the neighborhood of a pixel consists of 16 pixels, and we would like to classify the block as oriented only if a significant number of these show the same orientation. Based on experiments with test data, a value of 6 was found to be suitable for classifying the block as oriented. After the classification, different interpolation is applied to find the missing samples inside the block according to the block type.

For a nonoriented block, bilinear interpolation is used to obtain the missing sample Y from the surrounding four original image pixels X_0, X_1, X_2, and X_3 (see Fig. 1). The general formula for bilinear interpolation is given as follows:

$$\begin{aligned} Y = (1-x)((1-y)\times X_0 + y \times X_1) \\ + x((1-y)\times X_2 + y\times X_3). \end{aligned} \tag{2}$$

Since the location of resampled pixels are known to us, we can further simplify the evaluation of the bilinear interpolation. In this case, the four output pixels (see Fig. 1) are computed from the input pixels as $O_1 = X_0, O_2 = (X_0 + X_1)/2, O_3 = (X_0 + X_2)/2$ and $O_4 = (X_0 + X_1 + X_2 + X_3)/4$.

For an oriented block, we use an adaptive directional interpolation technique. We model the missing samples inside the block using oriented polynomials up to degree d. The oriented model for the block is given by

$$f(x,y) = \sum_{j=0}^{d-1} a_j (y\cos\alpha - x\sin\alpha)^j \tag{3}$$

where $j = 0,\ldots,d-1$. Here, $f(x,y)$ denotes the desired interpolated point inside the block and α denotes the dominant orientation of the block. The parameter d denotes the degree used for the α orientation. In our system, we used degree 4 for the horizontal and vertical direction and degree 7 for the other six directions. As mentioned in [1], a higher order approximation is used for the skew-oriented blocks as they have a higher degree of freedom along the orthogonal direction (i.e., for a given orientation, more parallel lines can exist within the neighborhood for a skewed direction than for the horizontal and vertical

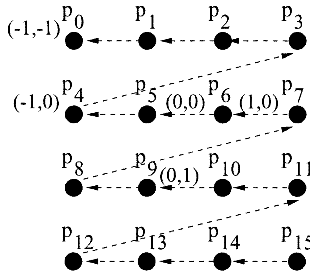


Fig. 3. Pixel arrangement.

cases). The coefficients a_j are found through least square modeling using the known original pixels in the neighborhood of the block. It can be shown that the following relationship holds:

$$f(x,y) = \mathbf{h}(\mathbf{H}^{\mathbf{T}}\mathbf{H})^{-1}\mathbf{H}^{\mathbf{T}}\mathbf{f} \tag{4}$$

where vector $\mathbf{f}$ is a 16×1 vector containing the luminance of the known pixels in the neighborhood of the block with entries organized as shown in Fig. 3, $\mathbf{f} = [\mathbf{p_0}, \mathbf{p_1}, \ldots, \mathbf{p_{15}}]^{\mathbf{T}}, \mathbf{H}$ is a size $16 \times d$ matrix with entries $H_{i,j} = (y_i \cos\alpha - x_i \sin\alpha)^{j-1}$ where (x_i, y_i) are the corresponding coordinate of p_i (see Fig. 3), and $\mathbf{h}$ is 1xd vector with entries $h_i = (y \cos\alpha - x \sin\alpha)^{j-1}$.

Let us define the vector $\mathbf{z}(\alpha, x, y) = \mathbf{h}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T$. Clearly, $\mathbf{z}$ is a function of the orientation α and the resampled pixel's location (x, y). Since we know the location of the missing samples inside the block *a priori*, we can precompute the following three weight vectors for each orientation used to compute the output pixels $O2, O3, O4$ as shown in Fig. 1.

$$\begin{aligned} \mathbf{a}(\alpha) &= \mathbf{z}(\alpha, 0.5, 0.0) \\ \mathbf{b}(\alpha) &= \mathbf{z}(\alpha, 0.0, 0.5) \\ \mathbf{c}(\alpha) &= \mathbf{z}(\alpha, 0.5, 0.5) \end{aligned} \tag{5}$$

where each one of these weight vectors is 1×16. The weight vectors $\mathbf{a}, \mathbf{b}$ and $\mathbf{c}$ can be precomputed and stored in a table (say, for example, in ROM). There are a total of eight sets of weight vectors indexed by the eight quantized orientations. Each component of the weight vector can be rounded to four decimal accuracy. The configuration for our adaptive interpolation filter is shown in Fig. 4. The inputs to the system are the dominant quantized orientation and the neighborhood of the current pixel. The orientation is used to address the appropriate bank of coefficients. These coefficients are then used in the three filters on the right portion of the schematic. The output pixels are found through the following relationship.

$$\begin{aligned} O1 &= p_5 \\ O2 &= \mathbf{a}(\alpha)\mathbf{f} \\ O3 &= \mathbf{b}(\alpha)\mathbf{f} \\ O4 &= \mathbf{c}(\alpha)\mathbf{f}. \end{aligned} \tag{6}$$

B. Direct Arbitrary Dimension Expansion

In this section, we discuss expansion by an arbitrary factor. Let the original image have width $x0$ and height $y0$. Let the desired expansion have width $x1$, and height $y1$. In the case of scaling by 2, we took a forward warping approach. For arbitrary scaling, we use a backward warping approach. A 1-D illustration of the backward warping approach is shown in Fig. 5. In the backward warping approach, the pixels of the expansion image are mapped back to the original image through coordinate transformation. For those pixels that fall on the integer lattice of the original image, we simply copy the pixel from the original image to the expansion image. On the other hand, expansion pixels that do not fall on the integer lattice of the original image are obtained through interpolation using the pixels of the original image (see Fig. 6). We organize the computation

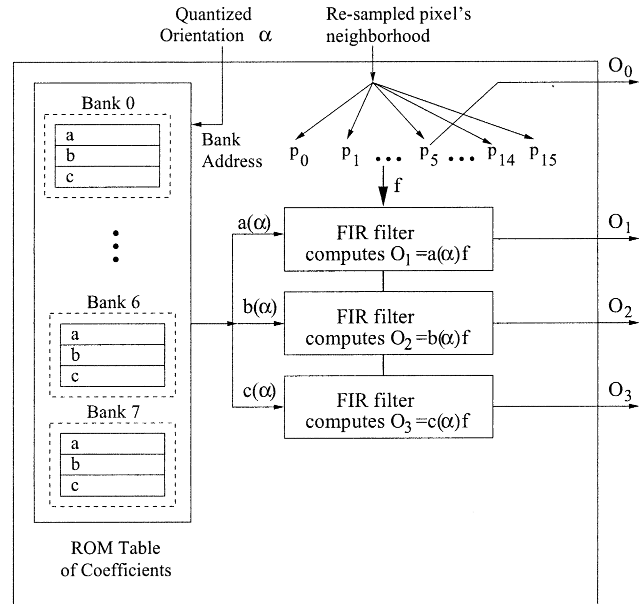


Fig. 4. Computation requirements for adaptive interpolation in scaling by two.

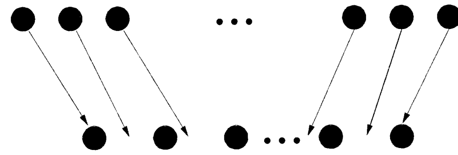


Fig. 5. Illustration of 1-D backward warping.

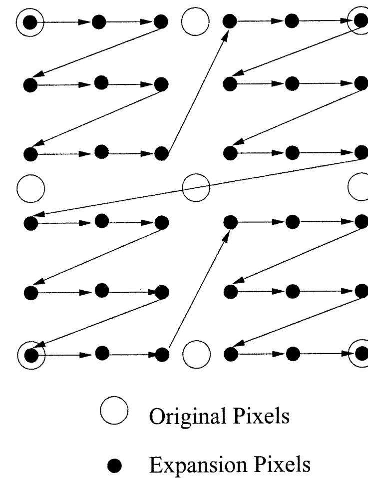


Fig. 6. Scanning pattern.

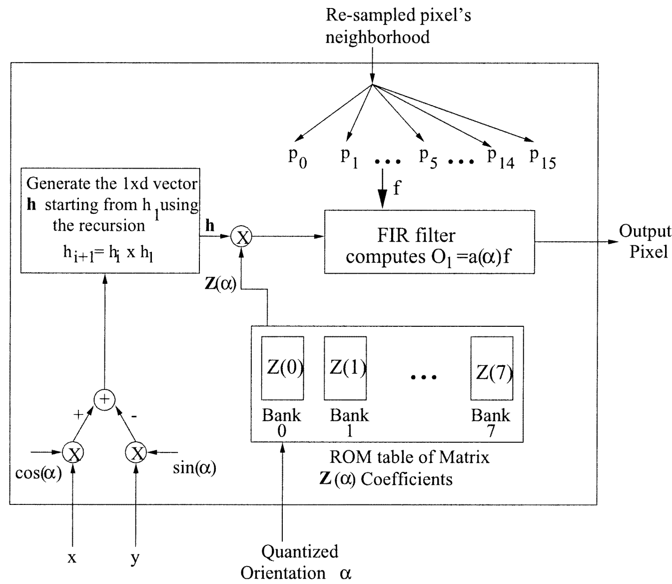


Fig. 7. Computation requirements for adaptive filtering in arbitrary expansion.

of the output pixels in the following way. First, we visit each block in the input image in a raster scan format. Then for each block, we compute all of the output pixels that map into this block before we leave the block. This process is shown in Fig. 6. The advantage of organizing the computation this way is that the classification process will only be done once per block. For each input image block, we perform classification as described for the scaling-by-two case. Then we interpolate the output pixels that are mapped back inside this input block. For output pixels that fall inside a nonoriented input block, we use bilinear interpolation. For output pixels that fall inside a nonoriented input block, the directional adaptive filter is used. Since we do not know the location of the output pixels (for any arbitrary irrational scaling factor) inside the input block *a priori*, the following simplified computational procedure will be taken. Let us define the $d \times 16$ matrix $\mathbf{Z}$ as $\mathbf{Z}(\alpha) = (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T$. Here d is the degree of oriented polynomial used for the α orientation. The matrices $\mathbf{Z}(\alpha)$ can clearly be precomputed for each of the eight orientations and stored in ROM as a table. Then, we can obtain our desired output pixel through the following relationship with $\mathbf{h}$ and $\mathbf{f}$ based on (4):

$$f(x,y) = \mathbf{h}\mathbf{Z}(\alpha)\mathbf{f}. \tag{7}$$

Fig. 7 shows the implementation of the direct arbitrary dimension expansion system.

If the expansion factor is irrational, then the computation shown in Fig. 7 needs to be performed. In the special case when the expansion factor is a rational number p/q where p and q are relatively prime, some simplifications can be made. We observe that there are $p^2 - 1$ noninteger pixel positions possible after backward warping of the expansion pixels onto the original image. In other words, there are only $p^2 - 1$ possibilities for $\mathbf{h}$ and only $8 \times (p^2 - 1)$ possible filters $\mathbf{h}\mathbf{Z}(\alpha)$. Therefore, if p is small, we can precompute the filters (instead of performing the matrix vector multiplication on-line to find $\mathbf{h}\mathbf{Z}(\alpha)$ shown in Fig. 7) and store them in ROM (just as in Fig. 4). For example, Fig. 6 shows a scaling of $2.5 = 5/2$. For this scaling factor, we can have $5^2 - 1 = 24$ possible values for the vector $\mathbf{h}$ corresponding to different values of x and y. We can precompute and store $24 \times 8 = 192$ filters of 16 coefficients each in ROM.

We found that when performing an arbitrary dimension expansion, better visual quality can be obtained if we combined the factor of two expansion with arbitrary expansion method described above. In other words, we apply the factor of two expansions iteratively until the dimensions of the resulting intermediate expansion image is no longer contained entirely inside the desired expansion dimension. At this point, we apply the arbitrary expansion as described above to obtain the final image from the intermediate expansion image. For example, if we desire an expansion of 2.5 in each dimension, we perform an expansion of 2 followed by an expansion of 1.25 in each dimension. This gives better visual quality than if we performed an expansion of 2.5 directly using the arbitrary expansion algorithm described in this section.

The adaptive scaling algorithm was applied to the Car Phone sequence in QCIF format. Frame 10 of the original sequence is shown in Fig. 8(a). Fig. 8(b) shows Frame 10 when scaled by a factor of 4 (i.e., 4CIF) in each dimension using only bilinear interpolation. Fig. 8(c) shows the same frame scaled using Cubic Spline interpolation. Fig. 8(d) shows Frame 10 when scaled using the adaptive scaling algorithm. Finally, Fig. 8(e) shows Frame 10 when scaled using an emulation of a Verilog description of the adaptive scaling hardware (we will describe the hardware architecture in detail in Section III). The emulation takes into account finite precision hardware effects in the orientation angle computation as well as in the filtering. No perceptable difference is observable between Fig. 8(d) and (e).

Due to the printing process, the visual quality of the images may be lost.

III. Architecture and Design

In this section, we develop an efficient hardware architecture based on the algorithm developed in the previous section. In particular, we focus on developing an architecture for scaling by a factor of two in each dimension. Then, we discuss the implementation tradeoffs based on this architecture. Area/throughput estimates based on the synthesis results of a Verilog description of this system will be discussed to show the feasability of a single chip ASIC implementation.

The data flow graph for the U and V components is shown in Fig. 9. The architecture for the two bilinear units required for the U and V components are discussed in Section III-E. We now focus on developing an architecture for Y component computation. The flow diagram of the computation for the Y component is shown in Fig. 2.

In hardware implementations, it is desirable to reduce the number of branches (or decision boxes) in the flow diagram. We found that the classification can be simplified with an imperceptible loss in performance. In particular, we can avoid the computation of the maximum and minimum luminance values in the neighborhood of a block as well as the check on the magnitude of $L_{\max} - L_{\min}$. One possible reason for this observation is as follows: Our algorithm applies bilinear interpolation

(a)

(b)

(c)

(d)

(e)

Fig. 8. (a) Frame 10 of car salesman. (b) Bilinear interpolation. (c) Interpolation using cubic spline. (d) Adaptive interpolation algorithm. (e) Adaptive interpolation result from Verilog description.

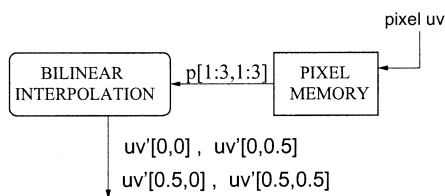


Fig. 9. Dataflow graph of computations for the U/V component.

to compute pixel values both for the case where a block is constant ($L_{\max} - L_{\min} < \text{threshold}$), and for the case where no single orientation is clearly dominant ($H_{\max} < \text{threshold}$). For blocks that are nearly constant, it is very likely that no single orientation will be dominant, so most such blocks will undergo bilinear scaling rather than directional adaptive scaling. This could account for the negligible loss in visual performance even when this step is omitted. In software implementations that do not penalize conditional branches as heavily as hardware, it may be worthwhile to retain this check to ensure that constant blocks are scaled using a normal bilinear interpolation. The modified flow diagram can be obtained from Fig. 2 by removing the computations in lines 1 and 2.

A data flow graph based on this flow diagram is shown in Fig. 10(b). Fig. 10(a) shows the notation used for the pixels, an-

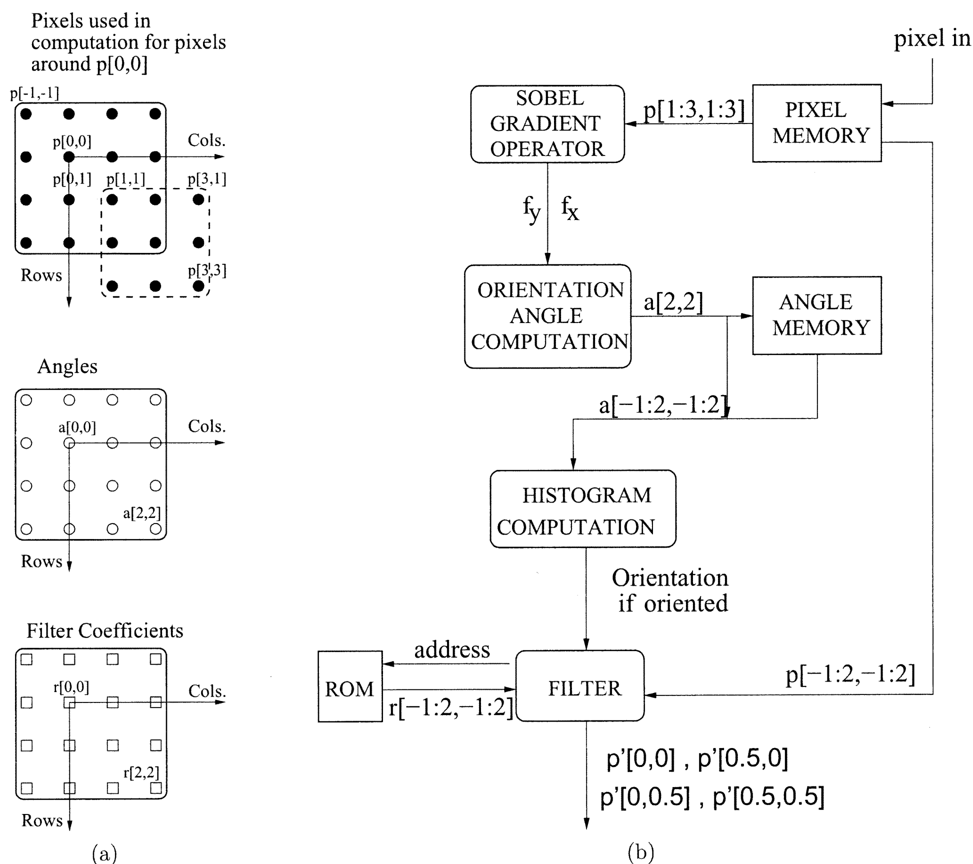


Fig. 10. (a) Neighborhood of the present block. (b) Dataflow graph of computations in required in adaptive scaling.

gles and filter coefficients. In Fig. 10(b), output pixels are being computed for the block corresponding to pixel $p(0,0)$ in the original image. The angle memory stores the previously computed orientation angles so that recomputation can be avoided as mentioned in Section II-A. The pixel memory stores the pixels of the original image. The Sobel block computes the gradients f_x and f_y at $p[2,2]$ using the pixels shown within dotted lines in Fig. 10. The angle block finds the quantized orientation angle for pixel $p(2,2)$. Then, a histogram is computed for the orientation angles in the neighborhood to find a dominant orientation (if it exists) for the block. In particular, the orientation that occurs the maximum number of times is found. Then $H_{\max}$ is compared with 6 to determine whether the block is oriented or not. If the neighborhood is oriented, then the appropriate filter coefficients are loaded from ROM and the interpolated pixels within the block are found. Note that in a scaling by two, we need to compute only three interpolated pixels per block. The fourth output pixel matches exactly with $p(0,0)$. The filtering block computes three inner products as mentioned in Section II-A. Note that bilinear interpolation case can also be handled by the same filtering unit. This can be achieved by storing another filter in ROM with coefficients corresponding to the bilinear interpolation. Thus we do not require separate hardware for the bilinear interpolation of the Y component.

From the dataflow graph in Fig. 10(b), we observe that the memory bandwidth required from the pixel memory is $9+16=25$ pixels for every input pixel (or every four output pixels). Similarly, the memory access rate per input pixel for the angle memory is 15. This memory bandwidth can be reduced by studying the structure of data that is required by the algorithm. For example, the memory bandwidth can be reduced by introducing some storage elements (i.e., registers) into the Sobel, histogram and filtering units as shown in Fig. 11(a). In this case, the memory access rate for the pixel data memory can be reduced to seven per input pixel. Similarly, the memory access requirements for the angle memory are reduced from 15 to three. The pixel memory access requirements can be further reduced to five if pixels $p(1,2)$ and $p(2,2)$ which are available in the Sobel unit can be shared with the filtering unit. This is shown in Fig. 11(b). Also, note that the pixel and angle memories are shown as delay lines. Due to the structure of the data accesses in Fig. 11(a) and (b), delay lines can be used instead of random access memories.

We also observe that the graph is feedforward. Therefore, we can apply pipelining at any feedforward cutset. For example, pipelining can be applied by using the Cutset 2 shown in Fig. 11. It can also be noted that if such a pipelining is applied, the pipeline registers for the pixels $p[2,-1:0]$ can be merged with the pixel delay lines. Similarly, the pipeline registers for $p[2,1:2]$ can be removed by sharing data available in the Sobel unit's registers. Additional pipeline stages can be applied at other feedforward cutsets to obtain further speedup.

In general, if the original image sequence has R rows and C columns at F frames per second. Then, the system must be able to accept input pixels at the rate $R\times C\times F$. This must be kept in mind when designing the individual units as well as

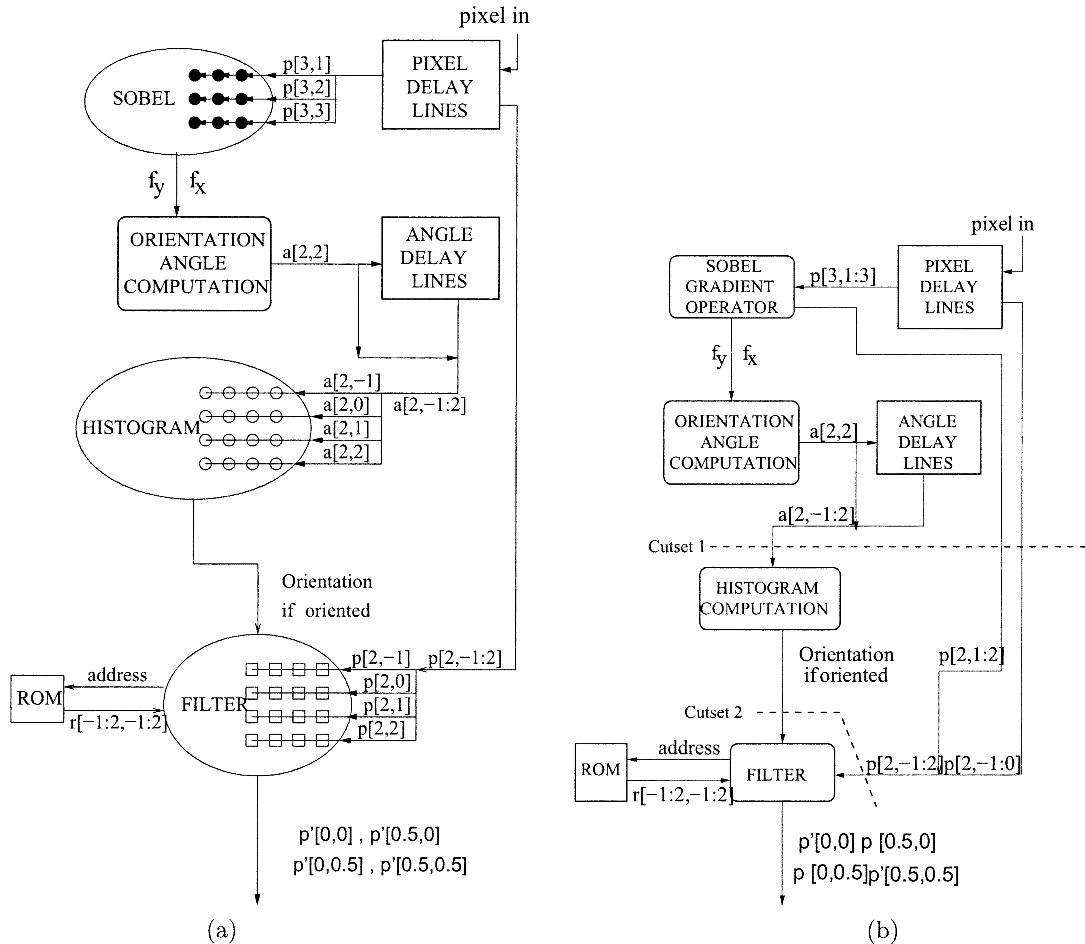


Fig. 11. Modified dataflow graph (a) showing reduced memory bandwidth requirements and (b) showing further reduction in memory bandwidth.

when deciding whether pipelining stages are required between the various units. In the next few sections, we consider each unit in detail and discuss various architecture level tradeoffs based on the system requirements.

A. Sobel Computation

The Sobel operator involves computing f_x and f_y as given by (1). A straightforward implementation would require ten adders to compute f_x and f_y. We can rewrite (1) as

$$\begin{aligned} f_x &= (f(i+1,j+1) - f(i-1,j-1)) \\ &\quad + 2 \times (f(i+1,j) - f(i-1,j)) + (f(i+1,j-1) \\ &\quad - f(i-1,j+1)) \\ f_y &= (f(i+1,j+1) - f(i-1,j-1)) - (f(i+1,j-1) \\ &\quad - f(i-1,j+1)) \\ &\quad + 2 \times (f(i,j+1) - f(i,j-1)). \end{aligned} \tag{8}$$

We assume for simplicity of discussion that all the adders are 12 bits (the output precision when no rounding is performed within the computation). By using subexpression sharing as suggested by (8), we can reduce the complexity of the Sobel operator to eight 12-bit additions. Clearly, all the adders need to be carry propagate adders (CPA) to permit sharing. Then the critical path, if the data flow graph is implemented as a combinational circuit, is given by two 1-bit full adders and one 12-bit carry propagate adders (CPA). If the throughput of the system is not constrained by the Sobel operator, we can fold the data flow graph so that we use only four 12-bit adders and reuse the adders over three clock cycles. Now the critical path is determined by a single 12-bit adder but we need three clock cycles. The adder complexity has been halved while some additional control circuitry and registers are required.

On the other hand, if we wanted a fast Sobel operator, then we would use the architecture shown in Fig. 12. Here carry save adders (CSA) are used (except at the final stage). A total of ten 12-bit adders (eight 12-bit CSA s and two 12-bit CPA s) are required. The critical path would now consist of three 1-bit full adders plus one 12-bit CPA. There seems to be no apparent advantage with this circuit. However, by speeding up only the last stage adder, the critical path of the combinational implementation can be reduced. In a conventional implementation [15], all adders would need to be speeded up to obtain critical path reduction. In summary, we can conclude that the most efficient implementation is a folded version of a conventional implementation. However, when a high speed implementation is required, the computation can be performed as in Fig. 12 to obtain speed-up.

B. Orientation Angle Computation

After f_x and f_y have been computed, the next step is to compute the corresponding quantized orientation angle O as de-

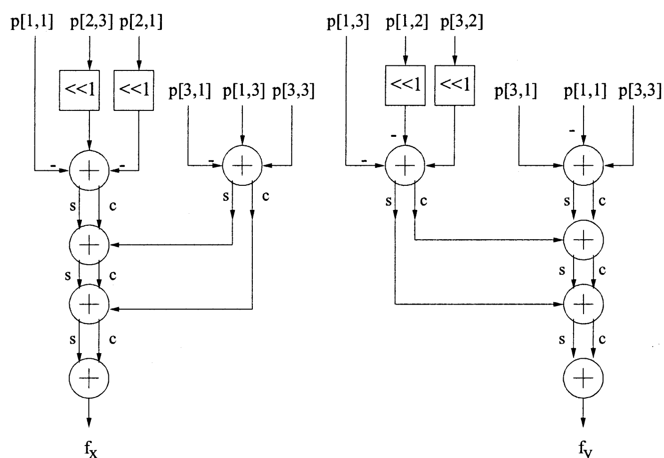


Fig. 12. Sobel operator computation using carry save technique.

scribed by (1). The two 12 bit integers f_x and f_y are divided and the result f_x/f_y is computed at an appropriate precision. Then the quantized angle O is computed using a ROM lookup table. The ROM table is a quantized version of $\theta = 57.5958 \times \tan^{-1}(-f_x/f_y) \bmod 180°$. Using the quantization terminology in [5], the angles $\phi_i = 11.25° + 22.5° \times i; i = 0, 1, \ldots 7$ correspond to the transition levels and the angles $\gamma_i = 22.5° \times i; i = 0, 1, \ldots 7$ correspond to the reconstruction levels. Clearly, the size of the ROM table and the precision required in the division would depend on the accuracy with which we need to define the orientations. For example, if we could allow a 2° error in specification of the transition levels, then we can compute the precision that would be required in the division operation. In particular, a 2° error in the quantization region at $11.25°$ would imply that we need to choose the precision so that we can represent the difference between $\tan(11.25°)$ and $\tan(13.25°)$ (i.e., $\tan(11.25°) - \tan(13.25°) = 0.036$). We consider the transition level $11.25°$ because at this angle the derivative of the tan function is the smallest amongst all the transition levels. A precision of at least 5 bits after the decimal point is required to represent 0.036. This requires that the division provide an output of $12 + 5 = 17$ bits. The size of the ROM look-up required would be $2^{17} = 128$ K words of size 3 bits each. Clearly, the hardware complexity required for this approach is extremely large.

The CORDIC trignometric computation technique was proposed in [6] as a technique for computing plane rotations as well as in conversions from rectangular to polar coordinates. The CORDIC computing technique also allows us to compute various trignometric functions. In particular, when the CORDIC processor is used in the angle accumulation mode [7], the angle $\theta = \tan^{-1}(Y/X)$ corresponding to the initial point coordinates (X, Y) is computed. The angle is found by applying a series of shift and add operations on the initial coordinates (X, Y). Each CORDIC circular rotation step performs

$$\begin{bmatrix} x(i+1) \\ y(i+1) \end{bmatrix} = \begin{bmatrix} 1 & -\mu_i 2^{-s(i)} \\ \mu_i 2^{-s(i)} & 1 \end{bmatrix} \begin{bmatrix} x(i) \\ y(i) \end{bmatrix}; \quad i = 0, 1, \ldots, n-1 \tag{9}$$

where $\mu_i = -\text{sign of } x_i \cdot y_i$. Note that the angle can be represented as a sequence of μ_i's. The angle is accumulated as $z(i+1) = z(i) - \mu_i a(i)$ where $a(i) = 2^{-s(i)}$. With this sequence of $s(i)$, any angle between $-90°$ and 90° can be accumulated. Usually, we set $s(i) = i$ so that at step $i, (x(i), y(i))$ is rotated by an angle $\tan(2^{-i})$. Since we want to define the quantization regions with an accuracy of 2°, the angle accumulation must be approximated within 2°. This means that we need at least six stages of CORDIC followed by a look-up table (addressed by the vector $\mu_i; i = 0, 1, \ldots, n-1$). The look-up table has 2^6 words of size 3 bits. Later we show how a modified CORDIC-like procedure can be used to perform the required computation using five stages followed by a look-up table of size 2^4 words of size 3 bits. In fact, the look-up table can be simplified into three combinational functions of four variables each.

Let us assume that as a first step we find the absolute value of f_y and f_x. This ensures that we have a point in the first quadrant. In particular, we want to efficiently represent the angles that separate the quantization regions (i.e., 11.25°, 33.75°, 56.25°, 78.75°) in terms of the elementary angles $\text{atan}(2^{s(i)}); i = 0, 1, \ldots, 5$. For example, we can write $78.75° \approx \tan^{-1}(2^0) + (\tan^{-1}(2^{-2}) + \tan^{-1}(2^{-3})) + (\tan^{-1}(2^{-3}) + \tan^{-1}(2^{-4}))$. Similarly, $56.25° \approx \tan^{-1}(2^0) + (\tan^{-1}(2^{-2}) + \tan^{-1}(2^{-3})) - (\tan^{-1}(2^{-3}) + \tan^{-1}(2^{-4})), 33.75° \approx \tan^{-1}(2^0) - (\tan^{-1}(2^{-2}) + \tan^{-1}(2^{-3})) + (\tan^{-1}(2^{-3}) + \tan^{-1}(2^{-4}))$ and $11.25° \approx \tan^{-1}(2^0) - (\tan^{-1}(2^{-2}) + \tan^{-1}(2^{-3})) - (\tan^{-1}(2^{-3}) + \tan^{-1}(2^{-4}))$. This means that if we use a CORDIC type computation with the shift sequence $s(i) = 0, 2, 3, 3, 4; i = 0, 1, 2, 3, 4$ and appropriate control to decide whether to add or subtract at each stage, we can obtain the required quantized orientation angle. The complete architecture is shown in Fig. 13. We note that apart from the signs of f_x and f_y, we need the sign of $y(i)$ for $i = 1, 3, 5$ to determine the orientation, as shown in Table II. This is shown as a combinational function in Fig. 13. Note that this signal flow graph can be folded [8] onto a single CORDIC like processor to obtain an area efficient implementation. If the throughput of the angle computation is critical, the unfolded data flow graph shown in Fig. 13 can be directly implemented. See [15] for the details of the folded implementation.

C. Histogram Computation

After the orientation angles in the current pixel's neighborhood have been computed, the next step is to compute the histogram of the pixel orientations in the neighborhood. As mentioned before, a complete histogram is not necessary. We need to find the dominant orientation in the neighborhood as well as the number of times that orientation occurs. The most obvious way to implement this computation is shown in Fig. 14. In this architecture, the 16 registers holding the pixel orientations in the neighborhood are each connected to a 3 to 8 decoder. If, for example, the pixel orientation is i, then the ith output is 1 and all others are 0. Then, for each orientation, a (16, 4) compressor counts the number of times each orientation occurs. An (n, m) compressor is a combinational circuit that sums its n inputs and provides an m-bit encoding of the sum as its output. Finally, the dominant orientation is found along with the number of times it occurs.

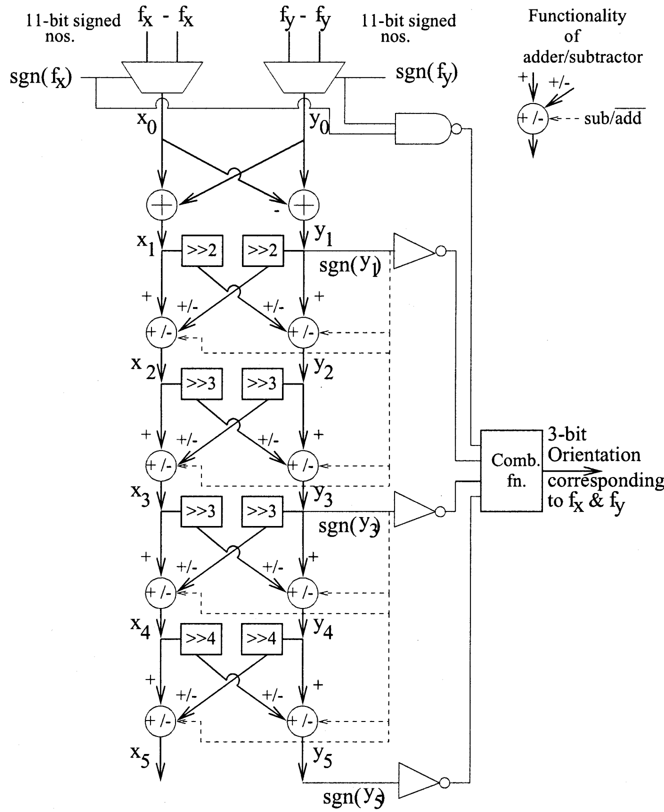


Fig. 13. Angle computation using CORDIC-like technique.

Clearly, this is a complex circuit with global communication between the decoders and the compressors. Such a complex parallel implementation would be required if the histogram computation is the throughput bottleneck. Otherwise, the architecture can be folded to obtain a much simpler implementation (shown in Fig. 15). The contents of the orientation angle registers are compared with the 3-bit counter output every clock cycle. The outputs of the comparators are fed to a (16, 4) compressor to obtain the total number of pixels that have the same orientation corresponding to the 3-bit counter contents. The dominant orientation and the maximum count registers are both initialized to zero. Also, during each cycle the output of the (16, 4) compressor is compared with the previous contents of the maximum count register. If the present output of the (16, 4) compressor is larger than the previous contents of the maximum count register, both the maximum count and dominant orientation registers are updated. The maximum count register is updated with the output value from the (16, 4) compressor, while the dominant orientation register is updated with the present 3-bit counter contents. Therefore, after eight cycles desired histogram data is obtained in the maximum value register and dominant orientation registers.

D. Oriented and Nonoriented Filtering

We observe that when performing a scaling by a factor of 2 in each direction, we need to find three interpolated pixels per input pixel. When the neighborhood around the current pixel is oriented, the oriented filter involves all the pixels in the neighborhood. On the other hand, if the neighborhood is nonoriented, then the bilinear filter involves only the region around the current pixel (see Fig. 1). In general, the same hardware can be used whether the neighborhood is oriented or not. When bilinear filtering is required, the coefficients corresponding to the unused pixels in the neighborhood can be set to zero.

The specific filter coefficients needed for oriented filtering are shown in the Appendix of [15]. In order to enable implementation of the filter using fixed point arithmetic, the coefficients need to be quantized. If l bits are used to represent each signed filter coefficient, then we can place an upper bound on the error in the output due to this quantization. We are in fact calculating an inner product which has the form $y = \sum_{k=0}^{15} h_k p_k$ where h_k are the filter coefficients, p_k are in the input pixels and y is the filter output. All the filter coefficients are such that $|h_k| \leq 1$. The maximum quantization error in the filter coefficients is given by $|\Delta| = 2^{-(l-2)}$. The maximum possible error at the output will be given by either $N_{\text{pos,max}} \times \Delta \times 255$ or $N_{\text{neg,max}} \times \Delta \times 255$ where $N_{\text{pos,max}}$ is the largest number of positive coefficients in any required filter and $N_{\text{neg,max}}$ is the largest number of negative coefficients in any required filter. For our case, $N_{\text{pos,max}} = 10$ and $N_{\text{neg,max}} = 8$. This means that the maximum possible error due to coefficient quantization is given by $2550 \times 2^{-(l-2)}$. In order that the error is upper-bounded by 5, we need $l = 11$ bits. Note that the bound we have derived is not tight. We found that $l = 10$ bits is sufficient and no significant error is introduced by the quantization process.

Clearly, the fastest implementation would require 16 multipliers per output pixel. This means a total of 48 multipliers. Observe also that the required multipliers are *not* fixed coefficient but general purpose multipliers. This is because the filter corresponding to different orientations are different. It would not be feasible to have such a large number of multipliers on a single chip. This would mean that a transformation of some form must be applied so that the multiplier hardware can be reused. One approach is to fold the computation required for each output pixel onto a smaller number multiply-accumulate (MAC) units. The choice of the appropriate number of MAC units (in other words, the folding factor) will depend on the throughput requirements. In particular, if we had one multiply accumulate unit for each output pixel, then we would need a total of three multiply accumulate units. The architecture for computing one output pixel in this case is shown in Fig. 16. The pixel data and the control circuitry is shown on the right hand side of Fig. 16. The partial sums of the filtering operation are accumulated in carry save form. The final addition is performed after the 16 MAC s have been completed.

E. Bilinear Interpolation for U and V Components

As mentioned earlier, it is enough to perform bilinear interpolation for the U and V components (no classification is required). A straightforward implementation would require five adders. A more efficient architecture is shown in Fig. 17 that uses sub-expression sharing [8] to reduce the number of adders. It consists of four 8-bit carry propagate adders. The division by two is achieved by a simple right shift by one bit. Note that two such bilinear units are required—one each for the U and V components.

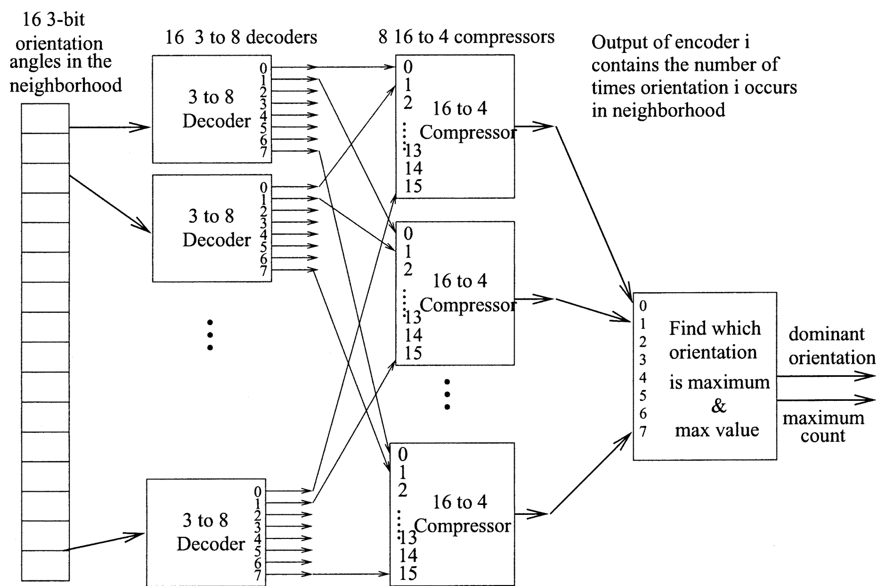


Fig. 14. Histogram computation.

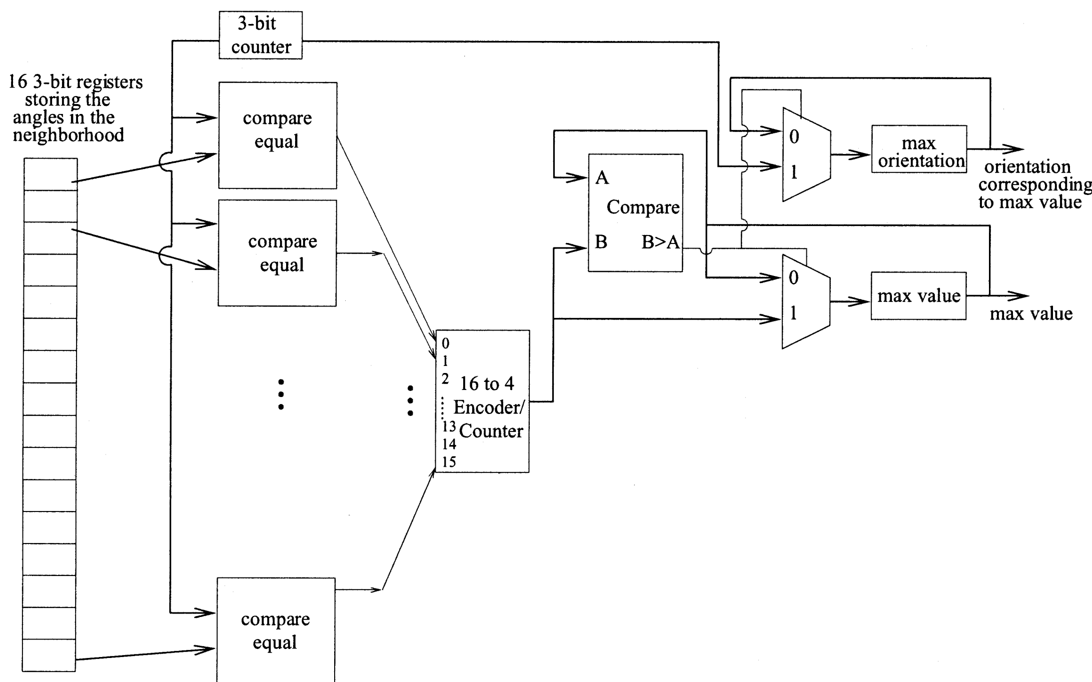


Fig. 15. Folded architecture for histogram computation.

F. System Discussion and VLSI Design

In this section, we discuss the system level tradeoffs that are involved in the design of a video scaling system. The H.261/H.263 standards [9] for videoconferencing/videotelephony specify two possible video formats, viz., QCIF and CIF. In order to enable a scaling of the videotelephone/videoconferencing signal, we consider here a scaling from QCIF (144 rows by 176 columns) to 4CIF (576 rows by 704 rows) at 30 frames per second. As mentioned before, performing the scaling in two steps (i.e., from QCIF to CIF and then from CIF to 4CIF) leads to better performance. We consider an architecture that can perform scaling in this manner. The system architecture is shown in Fig. 18. There are two identical scaling blocks corresponding to scaling from QCIF to CIF and from CIF to 4CIF. The input delay lines provide Y component of the pixel to the sobel, angle and filter units. They also provide U and V components to the bilinear unit. In all, four input delay lines are needed for the Y component and lines lines (of half the size) for the U and V components. As explained in Section III-D, the output pixels corresponding to a block of pixels (shown in Fig. 1) are generated at one time. In order to convert the output pixels to raster scan (progressive scan) order, output sync delay lines are required. We note that four lines are required for each

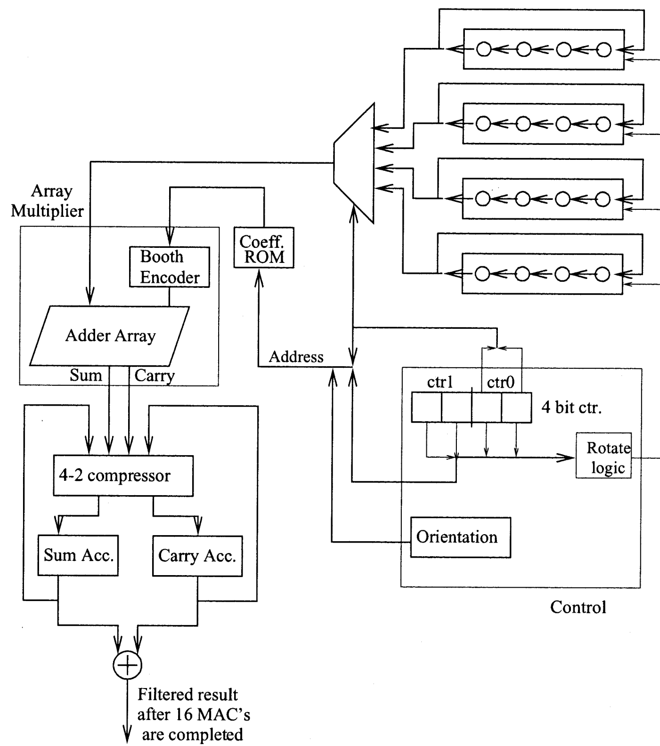


Fig. 16. Folded filter architecture.

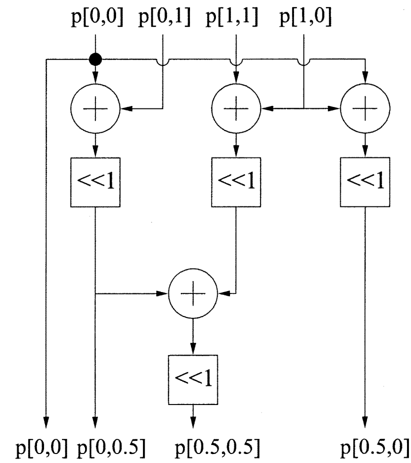


Fig. 17. Bilinear interpolation for U and V components.

of the Y, U and V components (two lines are used to store the output from the filter/bilinear units and while the other two lines provide the outputs in progressive scan order).

A QCIF video sequence at 30 frames/s has a pixel rate of 7.6×10^5 pixels/s while a CIF video sequence has a pixel rate of 3.04×10^6 pixels/s. The first scaling-by-two block (QCIF to CIF) is required to compute the four output pixels corresponding to one input pixel within 1315 ns. The second scaling-by-two block (CIF to 4CIF) is required to compute the output pixels corresponding to one input pixel within 328 ns. In general, for an image with dimension R rows and C columns at a frame rate F, the time available t to compute the output pixels in the region around one input pixel can be written as

$$t = \frac{10^9}{F \times R \times C}, \tag{10}$$

where t is in ns.

We first estimate the number of clock cycles required by each unit to complete the computation corresponding to one input pixel. This will tell us whether or not we need pipelining between units [like that shown in Fig. 11(b)] to obtain the required throughput. Based on the architecture shown in Fig. 16, at least 17 clocks are needed to complete the filtering computation. Similarly, the folded histogram architecture requires about ten clock cycles to complete the histogram computation. The angle computation requires about seven clock cycles while the folded Sobel computation requires about three clock cycles. We estimate that the architecture takes about 37 clock cycles to complete the output pixels corresponding to one input pixel. Taking into account the interface between these units and the output synchronization delay lines, a total of 42 clock cycles are required. This means that for real-time conversion of CIF to 4CIF (corresponding to the second scaling-by-two block in Fig. 18), a clock cycle of less that $328/42 \approx 7.81$ ns is required. The above assumes that there is no pipelining between units.

In order to show that the throughput can be attained in a real design, we described the various blocks in Verilog, synthesized them and performed a static timing analysis to estimate the speed. The synthesis was performed using a 0.8μm standard library using Cadence's Synergy tool. We also estimate the area based on an automatic layout of the standard cells using Cadence's Silicon Ensemble corresponding to the computation and control circuitry. The area of the delay lines and ROM were estimated based on the literature [10]–[12]. Our goal is to perform synthesis to obtain a clock period as close to 8 ns as possible. If we are unable to attain the required throughput, then there are two possible approaches. The first approach is to apply pipelining to obtain a speed-up that allows us to meet the throughput. Alternately, we can use a newer technology such as 0.5μm. If we assume a linear scaling of propagation delay [13] with technology, then it is enough to obtain a critical path of $8 \times 0.8/0.5 = 12.8$ ns. In reality, the scaling will be less than linear so that the requirements using the 0.8μ library will be tighter. In the following, we assume that the target was to achieve the throughput requirements at 0.5μm. Therefore, all synthesis timing requirements were scaled up to 0.8μm. A Verilog description was specified for each computational block. In addition, a controller was designed that handled all interaction between blocks. Besides, the controller also maintained a count of the rows/columns and handled image boundary conditions.

The synthesis results suggested that the final adder of 22 bits in the filtering block be separated into two 11-bit additions performed in consecutive clock cycles. This enabled a clock period less than 12.8 ns. A total of 20 clocks were needed to complete the filtering (assuming that the histogram output was available).

Architectures were specified in Verilog for the other blocks so that the required clock speed can be matched. The Sobel computation was implemented using the carry save technique in Fig. 12. This was required in order to meet the clock speed requirements. A total of three clock cycles were required to complete the computation.

Some optimizations were required in the folded CORDIC-based angle computation unit in order to match the clock speed

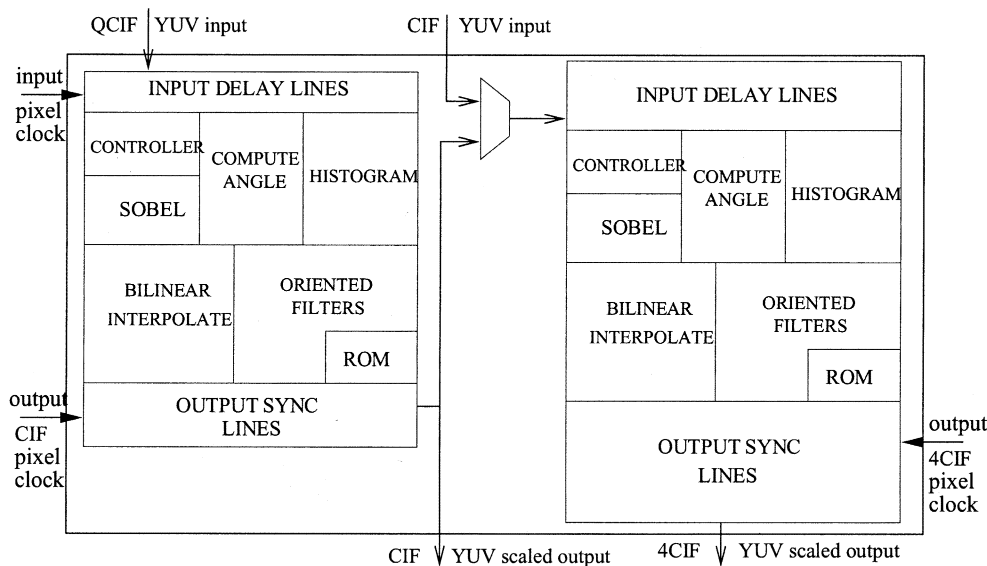


Fig. 18. Overall system architecture for scaling-by-four in each dimension.

TABLE I
CELL AREA AND TIMING FROM SYNTHESIS

Name of Unit	Standard Cell Count	Standard Cell Area	No. of Clock Cycles	Synthesis Timing
Sobel	702	911601.00	3	10.76ns
Angle	815	769414.50	7	10.81ns
Histogram	410	393295.50	10	8.04ns
Filter	841	815625.00	20	9.80ns

requirements. The critical path was governed by the 15-bit adder that was used to perform the CORDIC shift and add operation. The implementation of the 15-bit adder was modified into a carry select adder [14] with three blocks of 5 bits each in order to meet the clock speed requirements. The complete angle computation requires seven clock cycles.

The histogram was implemented as shown in Fig. 15. Again, optimizations were required to meet the throughput requirements. The critical path consists of the 4-bit equality comparators, the (16, 4) compressor followed by the 4-bit comparator. In order to meet the throughput requirements, the unit was internally pipelined. The (16, 4) compressor was split into four (4, 3) compressor circuits followed by an adder that sums up the four results from the four (4, 3) compressors. The first pipeline stage consists of the 4-bit equality comparators and the four (4, 3) compressors. The second pipeline stage consists of the adder that sums up the results and the 4-bit comparator. This allowed a clock speed less than 12.5 ns.

Table I summarizes some of the synthesis results using Cadence's Synergy tool. Note that three such copies of the filter are required to compute all three interpolated output pixels in parallel. The computation and control parts required an area of 35.3 mm^2 at 0.8μm. The delay lines required an estimated area (based on [11]) of $\approx$43 mm^2 at 0.8 μm. This implies a total area of $\approx$78.3 mm^2 at 0.8 μm. Assuming that the areas scale by $(5/8)^2$, an area of about 30.6 mm^2 at 0.5 μm. The results indicate that without pipelining between units we can achieve the throughput required to scale a QCIF image to a 4CIF image at 30 frames/s using a 0.5μm technology.

TABLE II
ORIENTATION AS A FUNCTION OF SIGNS OF y_{-1}, x_{-1}, y_1, y_3 AND y_5

$-sgn(x_{-1}.y_{-1})$	$sgn(y_1)$	$sgn(y_3)$	$sgn(y_5)$	f
-1	-1	-1	-1	000
-1	-1	-1	+1	001
-1	-1	+1	-1	001
-1	-1	+1	+1	010
-1	+1	-1	-1	010
-1	+1	-1	+1	011
-1	+1	+1	-1	011
-1	+1	+1	+1	100
+1	-1	-1	-1	000
+1	-1	-1	+1	111
+1	-1	+1	-1	111
+1	-1	+1	+1	110
+1	+1	-1	-1	110
+1	+1	-1	+1	101
+1	+1	+1	-1	101
+1	+1	+1	+1	100

G. Extension to Larger Image Dimensions

The architecture we have proposed in the previous section can be extended to handle the scaling of larger input image sizes. This will require increasing the throughput by applying a combination of pipelining and parallelism as suggested in Section II. In particular, by using two multipliers per output pixel (instead of the one multiplier we used in Fig. 16), we can complete the filter computation in $\approx$12 cycles. Let us introduce pipeline latches after the Sobel and angle units (at Cutset 1 in Fig. 11)

and after the histogram unit (at Cutset 2 in Fig. 11). The pipeline stage that takes the largest number of clock cycles is the filter stage. This means that we have managed to raise the throughput by a factor of $42/12 \approx 3.5$. This means that at 0.5μm we can expect to handle an image of size about $288 \times \sqrt{3.5}$ rows by $352 \times \sqrt{3.5}$ or nearly 538×658. If we had access to 0.25 μm technology, we can increase the throughput by a further factor of two. In other words, we can handle input image sizes of upto 760×930. This suggests that our architecture could find application in conversion from SDTV to HDTV (see [9] for summary of HDTV standard).

IV. Conclusion

In this paper, we have proposed a high-performance algorithm for image scaling using the oriented polynomial image model. The algorithm was also extended to arbitrary scaling factors. We have also shown that an efficient VLSI architecture and implementation can be obtained for the case when we scale-by-two along each dimension. Static timing analysis of the circuits synthesisized from an RTL Verilog description showed that the throughput requirements can be met for a system that scales QCIF video to 4CIF format at 30 frames/s. The total chip area for such an implementation was estimated to be about 20 mm^2 at 0.5 μm.

References

[1] Y. Wang and S. K. Mitra, "Image representation using block pattern models and its image processing applications," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 15, pp. 321–336, Apr. 1993.

[2] D. M. Martinez and J. Lim, "Spatial interpolation of interlaced television pictures," in *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing*, May 1989, pp. 1886–1889.

[3] K. Jensen and D. Anastassiou, "Saptial resoultion enhancement of images using nonlinear interpolation," in *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing*, Apr. 1990, pp. 2045–2048.

[4] J. Salonen, "Edge and motion controlled spatial upconversion," *IEEE Trans. Consumer Electron.* , vol. 40, pp. 225–233, Aug. 1994.

[5] A. K. Jain, *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

[6] J. E. Volder, "The CORDIC architectures computing technique," *IRE Trans. Electron. Comput.*, vol. EC-8, pp. 330–334, Sept. 1959.

[7] Y. H. Hu, "CORDIC-based VLSI architectures for digital signal processing," *IEEE Signal Processing Mag.*, vol. 9, pp. 16–35, July 1992.

[8] K. K. Parhi, "High-level algorithm and architecture transformations for DSP synthesis," *J. VLSI Signal Process.*, vol. 9, pp. 121–143, Jan. 1995.

[9] *Digital Consumer Electronics Handbook*, McGraw-Hill, New York, 1997.

[10] K. Dejhan, F. Jutand, N. Demassieux, O. Colavin, A. Galisson, and A. Artieri, "A new high-performance programmable delay line IC," *IEEE Trans. Consumer Electron.*, vol. 35, pp. 893–899, Nov. 1989.

[11] H.-J. Mattausch, F. Matthiesen, J. Hartl, R. Tielert, and E. P. Jacobs, "A memory-based high-speed delay line with large adjustable length," *IEEE J. Solid-State Circuits*, vol. 23, pp. 105–110, Feb. 1988.

[12] F. Rothan, C. Joanblanq, and P. Senn, "A video delay line compiler," in *IEEE Int. Symp. Circuits and Systems*, May 1990, pp. 65–68.

[13] N. H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*. Reading, MA: Addison-Wesley, 1993.

[14] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 1996.

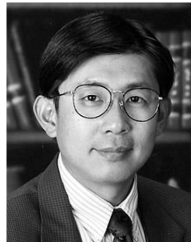
[15] A. Raghupathy, "Low Power and High Speed Algorithms and VLSI Architectures for Error Control Coding and Adaptive Video Scaling,", Ph.D. dissertation, Univ. Maryland, College Park, Dec. 1998.

Arun Raghupathy (S'95-M'99) received the B.Tech degree in electronics and communications engineering from the Indian Institute of Technology, Madras, in 1993, and the M.S and Ph.D degrees in electrical engineering from the University of Maryland at College Park in 1995 and 1998, respectively. His Ph.D. research focused on the development of techniques that enable the implementation of low power high-performance VLSI signal processing systems.

Currently, he is a Staff Engineer in the ASIC department at Qualcomm, Inc., San Diego, CA, where he is involved in the development of modem ASICs for Third Generation (3G) CDMA systems including those used in direct conversion receivers. His areas of interest include signal processing for communications, image and video processing.

Nitin Chandrachoodan was born on August 11, 1975 in Madras, India. He received the B. Tech degree in electronics and communications engineering from the Indian Institute of Technology (IIT), Madras, in 1996, and the M.S. and Ph.D. degrees in electrical engineering from the University of Maryland at College Park in 1998 and 2002, respectively.

He is currently with the Intel Software Radio Laboratory at IIT Madras. His research interests include analysis and representation techniques for system level synthesis of DSP dataflow graphs.



K. J. Ray Liu (F'03) received the B.S. degree from the National Taiwan University, Taipei, Taiwan, R.O.C., in 1983, and the Ph.D. degree from the University of California, Los Angeles, in 1990, both in electrical engineering.

He is Professor with the Electrical and Computer Engineering Department and the Institute for Systems Research, University of Maryland, College Park. His research interests span broad aspects of signal processing algorithms and architectures; multimedia communications and signal processing; wireless communications and networking; information security; and bioinformatics, in which he has published over 280 refereed papers. He was the founding Editor-in-Chief of *EURASIP Journal on Applied Signal Processing* and an editor of *Journal of VLSI Signal Processing Systems*.

Dr. Liu is the Editor-in-Chief of IEEE Signal Processing Magazine and has been an Associate Editor of IEEE Transactions on Signal Processing, a Guest Editor of special issues on Multimedia Signal Processing of Proceedings of the IEEE, a Guest Editor of special issue on Signal Processing for Wireless Communications of IEEE Journal of Selected Areas in Communications, a Guest Editor of special issue on Multimedia Communications over Networks of IEEE Signal Processing Magazine, and a Guest Editor of special issue on Multimedia over IP of IEEE Transactions on Multimedia. He has served as Chairman of the Multimedia Signal Processing Technical Committee of the IEEE Signal Processing Society. He is the recipient of numerous honors and awards, including the IEEE Signal Processing Society 2004 Distinguished Lecturer, the 1994 National Science Foundation Young InvestigatorAward, the IEEE Signal Processing Society's 1993 Senior Award (Best Paper Award), IEEE 50th Vehicular Technology Conference Best Paper Award, Amsterdam, 1999. He also received the George Corcoran Award in 1994 for outstanding contributions to electrical engineering education and the Outstanding Systems Engineering Faculty Award in 1996 in recognition of outstanding contributions in interdisciplinary research, both from UMD.