# Efficient, Low-Complexity Image Coding with a Set-Partitioning Embedded Block Coder

William A. Pearlman, Asad Islam, Nithin Nagaraj, and Amir Said

**Abstract**

We propose an embedded, block-based, image wavelet transform coding algorithm of low complexity. It uses a recursive set-partitioning procedure to sort subsets of wavelet coefficients by maximum magnitude with respect to thresholds that are integer powers of two. It exploits two fundamental characteristics of an image transform—the well defined hierarchical structure, and energy clustering in frequency and in space. The two partition strategies allow for versatile and efficient coding of several image transform structures, including dyadic, blocks inside subbands, wavelet packets, and DCT (discrete cosine transform). We describe the use of this coding algorithm in several implementations, including reversible (lossless) coding and its adaptation for color images, and show extensive comparisons with other state-of-the-art coders, such as SPIHT and JPEG2000. We conclude that this algorithm, in addition to being very flexible, retains all the desirable features of these algorithms and is highly competitive to them in compression efficiency.

**Keywords:** image coding, color image coding, lossless coding, wavelet coding, hierarchical coding, embedded coding, entropy coding

## I. Introduction

Effective and computationally simple techniques of transform-based image coding have been realized using set partitioning and significance testing on hierarchical structures of transformed images. Said and Pearlman [4] introduced such a technique in their SPIHT (Set Partitioning In Hierarchical Trees) algorithm, in their successful effort to extend and improve Shapiro's EZW (Embedded Zerotree Wavelet) algorithm [3]. The SPIHT algorithm has since become a standard benchmark in image compression.

The SPIHT scheme employs an iterative partitioning or splitting of sets or groups of pixels (or transform coefficients), in which the tested set is divided when the maximum magnitude within it exceeds a certain threshold. When the set passes the test and is hence divided, it is said to be significant. Otherwise it is said to be insignificant. Insignificant sets are repeatedly tested at successively lowered thresholds until isolated significant pixels are identified. This procedure sorts sets and pixels by the level of their threshold of significance. The results of these so-called significance tests describe the path taken by the coder to code the source samples. Since the binary outcomes of these tests are put into the bit stream as a '1' or '0', the decoder at the destination can duplicate the execution path of the encoder.

The principle of set partitioning and sorting by significance is the key to excellent coding performance with very low computational complexity. This recognition has spawned more algorithms in this category, among which are Amplitude and Group Partitioning (AGP) [5], SWEET [7], NQS [13], and Set Partitioning Embedded bloCK (SPECK), which is the subject of this paper. An important characteristic that this class of coders possesses is the capability of progressive transmission and embeddedness. Progressive transmission refers to the transmission of information in decreasing order of its information content. In other words, the coefficients with the highest magnitudes are transmitted first. Since these coding schemes transmit value information in decreasing order of significance, this ensures that the transmission is progressive. Schemes like EZW, SPIHT, and the herein to be described SPECK maintain a list of significant pixels, so that their bits can be sent in decreasing bit plane order. Such a transmission scheme makes it possible for the bitstream to be embedded, i.e., a single coded file can used to decode the image at almost any rate less than or equal to the coded rate, to give the best reconstruction possible with the particular coding scheme.

W.A. Pearlman is with the Electrical, Computer and Systems Engineering Dept., Rensselaer Polytechnic Institute, Troy, NY 12180, USA; E-mail:pearlw@rpi.edu.

A. Islam is with Nokia, Inc. , Irvington, TX 75039, USA; E-mail: Asad.Islam@nokia.com.

N. Nagaraj is with GE John F. Welch Technology Center, Bangalore, India; E-mail: nithin.nagaraj@geind.ge.com.

A. Said is with Hewlett-Packard Laboratories, Palo Alto, CA 94304, USA; E-mail: said@hpl.hp.com

The SPECK image coding scheme has its roots primarily in the ideas developed in the SPIHT [4], AGP [5], and SWEET [7] image coding algorithms. An algorithm similar to SPECK was independently developed by Munteanu *et al.* [2]. SPECK is different from SPIHT and EZW in that it does not use trees which span and exploit the similarity across different subbands of a wavelet decomposition; rather, like AGP, SWEET,and NQS, it makes use of sets in the form of blocks of contiguous coefficients within subbands. A more detailed discussion describing the position of this coding scheme with respect to these coders is given in Section III-E. The main idea is to exploit the clustering of energy in frequency and space in hierarchical structures of transformed images.

The SPECK algorithm has been utilized in several successful image coders. SPECK was first introduced in July 1998 by Islam and Pearlman as a low complexity option for JPEG2000 [11] and was reported publicly in a conference paper in January 1999 [16]. In March 1999, it was again presented before the JPEG2000 Working Group with additional data on comparative run-times with JPEG2000's VM 3.2A (Verification Model, version 3.2A) [15], which was essentially the EBCOT coder by Taubman [1]. These tests showed that SPECK was 4.6 to 15.7 faster than VM 3.2A in encoding and 8.1 to 12.1 faster in decoding on the average over a set of four images and a set of four rates, 0.25, 0.50, 1.0, and 2.0 bits per pel. The range of runtime factors resulted from the different versions of scalability and use or non-use of back-end entropy coding used in the tests. Because of the simplicity of these coders, there were reductions of PSNR from that of VM 3.2A ranging from a minimum of 0.48 dB for entropy-coded versions to a maximum of 0.85 dB for non-entropy-coded versions. Later that year in June, SPECK was incorporated into the JPEG2000 coding framework, where a simple command line switch initiated the SPECK coding engine in place of EBCOT [17]. This implementation was named Subband Hierarchical Block Partitioning (SBHP). The programming and tests were carried out at Hewlett Packard Laboratories, but there were contributors from Texas Instruments, Canon Information Systems Research in Australia (CISRA), Teralogic, and Rensselaer Polytechnic Institute. The work was later published in condensed form in the ICASSP2000 Proceedings [18]. SBHP used a very simple fixed Huffman code of 15 symbols for encoding the significance map bits delivered by the SPECK algorithm. As with SPECK, the other types of bits, the magnitude refinement and sign bits, were not further encoded. Extensive tests showed that for natural images, such as photographic and medical images, the reductions in PSNR from VM 4.2 were in the range of 0.4–0.5 dB. SBHP showed losses in bit rate at the same PSNR level from 5–10% for lossy compression and only 1–2% for lossless compression. Yet encoding time was about 4 times faster and decoding time about 6 to 8 times faster for the embedded version and as much as 11 times faster for the non-embedded version, in which case the complexity of SBHP becomes close to baseline JPEG. In fact, embedded SBHP proved to be faster even than SPIHT with unencoded output in these experiments. The non-embedded coding is actually progressive by value of bitplane threshold, so that higher magnitude wavelet coefficients are encoded into the bitstream before lower values. There is no counterpart of this additional functionality in JPEG2000.

A higher complexity variant of SPECK, called EZBC (Embedded Zero Block Coding), was reported by Hsiang and Woods [20]. EZBC uses the SPECK algorithm to produce the significance map, magnitude refinement and sign bits, but then uses the context-based adaptive, arithmetic coding of EBCOT to encode all these kinds of bits. EZBC outperformed SPECK, on the average over several natural images and rates up to 1.0 bit per pel by about 0.45 dB, because of this additional encoding. In fact, it also outperformed JPEG2000 VM 3.1A in most cases. Although higher in complexity than basic SPECK, EZBC is still somewhat lower in complexity than JPEG2000, because at each threshold it only needs to pass through coefficients that have previously become significant. The JPEG2000 coder requires passage through all coefficients at each threshold.

In this paper, we shall describe the SPECK algorithm, its characteristic features, and the implementation and performance of its low complexity forms in monochrome and color still image compression. The paper is organized into the following sections. Section II provides the terminology and methodology used in the algorithm, followed by Section III which explains and lists in pseudo-code the actual algorithm. The sub-section of III-E provides the motivation behind SPECK and explains its position among the class of similar hierarchical coding schemes. The final sub-section III-F gives the numerical and visual results obtained with this and other competing coding schemes. In Section IV, we present an embedded color implementation of SPECK and show that its performance is comparable to or superior than SPIHT and JPEG 2000 on YUV frames extracted from standard MPEG-4 video sequences. SBHP, a SPECK variant implemented in the JPEG2000 platform, is then described in Section V and compared to JPEG2000, followed by the concluding statements in Section VI.

## II. The Coding Methodology

In this section, we explain the idea of the SPECK coding scheme and its terminology. Consider an image $\mathcal{X}$ which has been adequately transformed using an appropriate subband transformation (most commonly, the discrete wavelet transform). The transformed image is said to exhibit a hierarchical pyramidal structure defined by the levels of decomposition, with the topmost level being the root. The finest pixels lie at the bottom level of the pyramid while the coarsest pixels lie at the top (root) level. The image $\mathcal{X}$ is represented by an indexed set of transformed coefficients $\{c_{i,j}\}$, located at pixel position $(i,j)$ in the transformed image. The terms *coefficients* and *pixels* will be used interchangeably, since there is usuallly no need to distinguish between value and position when referencing them. Pixels are grouped together in sets which comprise regions in the transformed image. Following the ideas of SPIHT, we say that a set $\mathcal{T}$ of pixels is *significant* with respect to $n$ if

$$\max_{(i,j)\in\mathcal{T}}\{|c_{i,j}|\} \geq 2^n$$

otherwise it is *insignificant*. We can write the significance of a set $\mathcal{T}$ as

$$\Gamma_n(\mathcal{T}) = \begin{cases} 1, & \text{if } 2^n \leq \max\limits_{(i,j)\,\in\,\mathcal{T}} |c_{i,j}| < 2^{n+1} \\ 0, & \text{else} \end{cases} \tag{1}$$

The SPECK algorithm makes use of rectangular regions of image. These regions or sets, henceforth referred to as sets of type $\mathcal{S}$, can be of varying dimensions. The dimension of a set $\mathcal{S}$ depends on the dimension of the original image and the subband level of the pyramidal structure at which the set lies.

We define the size of a set to be the cardinality $\mathcal{C}$ of the set, i.e., the number of elements (pixels) in the set.

$$\text{size}(\mathcal{S}) = \mathcal{C}(\mathcal{S}) \equiv |\mathcal{S}| \tag{2}$$

During the course of the algorithm, sets of various sizes will be formed, depending on the characteristics of pixels in the original set. Note that a set of size 1 consists of just one pixel.

The other type of sets used in the SPECK algorithm are referred to as sets of type $\mathcal{I}$. These sets are obtained by chopping off a small square region from the top left portion of a larger square region. A typical set $\mathcal{I}$ is illustrated in Fig. 3. A set $\mathcal{I}$ is always decomposed into $\mathcal{S}$ sets in a prescribed way, so as to progress through the transformed image from coarser to finer resolution subbands. The coding part of SPECK always takes place on the $\mathcal{S}$ sets.

To encode an $\mathcal{S}$ set, the SPECK algorithm follows closely the methodology used in the SPIHT algorithm [4]. The difference lies in the sorting pass where instead of using spatial orientation trees for significance testing, we use sets of type $\mathcal{S}$ as defined above. The motivation behind this is to exploit the clustering of energy found in transformed images and concentrate on those areas of the set which have high energy. This ensures that pixels with high information content are coded first.

We maintain two linked lists: LIS – List of Insignificant Sets, and LSP – List of Significant Pixels. The former contains sets of type $\mathcal{S}$ of varying sizes which have not yet been found significant against a threshold $n$ while the latter obviously contains those pixels which have tested significant against $n$. Alternatively, as will become obvious later on, we can use an array of smaller lists of type LIS, each containing sets of type $\mathcal{S}$ of a fixed size, instead of using a single large list having sets $\mathcal{S}$ of varying sizes. These smaller lists are ordered by size from smallest single pixel sets first (top) to largest sets last (bottom). This ordering is the functional equivalent of separating the LIS into two lists, an LIP (list of insignificant points) and an LIS (list of insignificant multi-point sets), as done in SPIHT. Use of multiple lists will speed up the encoding/decoding process.

## III. The SPECK Algorithm

Having set-up and defined the terminology used in the SPECK coding method, we are now in a position to understand the actual algorithm.

```
1) Initialization
    • Partition image transform $\mathcal{X}$ into two sets: $\mathcal{S} \equiv$ root, and $\mathcal{I} \equiv \mathcal{X} - \mathcal{S}$ (Fig. 3).
    • Output $n_{max} = \lfloor \log_2(\max_{(i,j)\in\mathcal{X}} |c_{i,j}|) \rfloor$
    • Add $\mathcal{S}$ to LIS and set LSP $= \phi$
2) Sorting Pass
    • In increasing order of size $|\mathcal{S}|$ of sets (smaller sets first)
        – for each set $\mathcal{S} \in$ LIS do ProcessS($\mathcal{S}$)
    • if $\mathcal{I} \neq \emptyset$, ProcessI()
3) Refinement Pass
    • for each $(i,j) \in$ LSP, except those included in the last sorting pass, output the $n$-th MSB of $|c_{i,j}|$
4) Quantization Step
    • decrement $n$ by 1, and go to step 2
```

Fig. 1. The SPECK Algorithm.

The main body of the SPECK coding algorithm is presented in pseudo-code in Fig. 1. It consists of four steps: the initialization step; the sorting pass; the refinement pass; and the quantization step. These steps call four functions, `ProcessS()`, `CodeS()`, `ProcessI()` and `CodeI()`, which are described in Fig. 2.

We shall now describe the operations of the SPECK algorithm presented in Fig. 1. We start with our source, a rectangular image $\mathcal{X}$, that has undergone an appropriate subband transformation. The image $\mathcal{X}$ consists of transformed coefficients $\{c_{i,j}\}$, located at pixel position $(i,j)$. Such an image exhibits a hierarchical pyramidal structure having subbands at different levels of its decomposition. The topmost band is the root of the pyramid.

The algorithm starts by partitioning the image into two sets: set $\mathcal{S}$ which is the root of the pyramid, and set $\mathcal{I}$ which is everything that is left of the image after taking out the root (see Fig. 3). To start the algorithm, set $\mathcal{S}$ is added to the LIS. We keep a note of the maximum threshold $n_{max}$ such that $c_{i,j}$ is insignificant with respect to $n_{max}+1$, $\forall c_{i,j} \in \mathcal{X}$, but at least one $c_{i,j} \in \mathcal{X}$ is significant against the threshold $n_{max}$.

## *A. Quadtree Partitioning*

Set $\mathcal{S}$ in LIS is processed by testing it for significance against the threshold $n = n_{max}$ (function `ProcessS`()). If not significant, it stays in the LIS. If $\mathcal{S}$ is significant, it is quadrisected, i.e., partitioned into four subsets $\mathcal{O}(\mathcal{S})$, each having size approximately one-fourth the size of the parent set $\mathcal{S}$ (function `CodeS`()). Fig. 4 gives an illustration of this partitioning process. In the following procedure `CodeS`(), each of these offspring sets $\mathcal{O}(\mathcal{S})$ is tested for significance for the same $n$ and, if significant, is quadrisected once more. If not significant, it is added to the LIS.

Each significant subset is, in turn, treated as a set of type $\mathcal{S}$ and processed recursively, via `ProcessS`() and `CodeS`(), until pixel-level is reached where the pixels that are significant in the original set $\mathcal{S}$ are located and thereby coded. The pixels/sets that are found insignificant during this selection process are added to LIS to be tested later against the next lower threshold.

The binary result of every significance test is sent to the code bitstream. Whenever a set $\mathcal{S}$ of size greater than one is significant, tests of four offspring sets follow, so that the binary significance decision paths map onto a quadtree. The motivation for this so-called quadtree partitioning of such sets achieves two goals: (1) to quickly identify the areas of high energy (magnitude) in the set $\mathcal{S}$ and code them first; and (2) to locate structured groups of coefficients that are below a decreasing sequence of magnitude thresholds, so as to limit the number of bits needed for their representation.

## *B. Octave Band Partitioning*

At this stage of the algorithm, all current sets of type $\mathcal{S}$ have been tested against $n$. The set $\mathcal{I}$ is processed next, by testing it against the same threshold $n$ (function `ProcessI`()). If it is found to be significant, it is partitioned by yet another partitioning scheme—the octave band partitioning. Fig. 5 gives an illustration of this partitioning scheme. Set $\mathcal{I}$ is partitioned into four sets—three sets of type $\mathcal{S}$ and one of type $\mathcal{I}$ (function `CodeI`()). The size of each of these three sets $\mathcal{S}$ is the same as that of the chopped portion of $\mathcal{X}$. The new set $\mathcal{I}$ that is formed by this partitioning process is now reduced in size.

```
Procedure ProcessS($\mathcal{S}$)
  1) output $\Gamma_n(\mathcal{S})$
  2) if $\Gamma_n(\mathcal{S}) = 1$
     • if $\mathcal{S}$ is a pixel, then output sign of $\mathcal{S}$ and add $\mathcal{S}$ to LSP
     • else CodeS($\mathcal{S}$)
     • if $\mathcal{S} \in$ LIS, then remove $\mathcal{S}$ from LIS
  3) else
     • if $\mathcal{S} \notin$ LIS, then add $\mathcal{S}$ to LIS
  4) return

Procedure CodeS($\mathcal{S}$)
  1) Partition $\mathcal{S}$ into four equal subsets $\mathcal{O}(\mathcal{S})$ (see Fig. 4)
  2) for each set $\mathcal{S}_i \in \mathcal{O}(\mathcal{S})$ $(i = 0, 1, 2, 3)$
     • output $\Gamma_n(\mathcal{S}_i)$
     • if $\Gamma_n(\mathcal{S}_i) = 1$
       – if $\mathcal{S}_i$ is a pixel, output its sign and add $\mathcal{S}_i$ to LSP
       – else CodeS($\mathcal{S}_i$)
     • else
       – add $\mathcal{S}_i$ to LIS
  3) return

Procedure ProcessI()
  1) output $\Gamma_n(\mathcal{I})$
  2) if $\Gamma_n(\mathcal{I}) = 1$
     • CodeI()
  3) return

Procedure CodeI()
  1) Partition $\mathcal{I}$ into four sets—three $\mathcal{S}_i$ and one $\mathcal{I}$ (see Fig. 5)
  2) for each of the three sets $\mathcal{S}_i$ $(i = 0, 1, 2)$
     • ProcessS($\mathcal{S}_i$)
  3) ProcessI()
  4) return
```

Fig. 2. The functions used by the SPECK Algorithm.

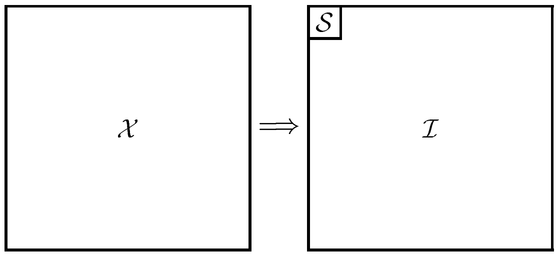

Fig. 3. Partitioning of image $\mathcal{X}$ into sets $\mathcal{S}$ and $\mathcal{I}$.

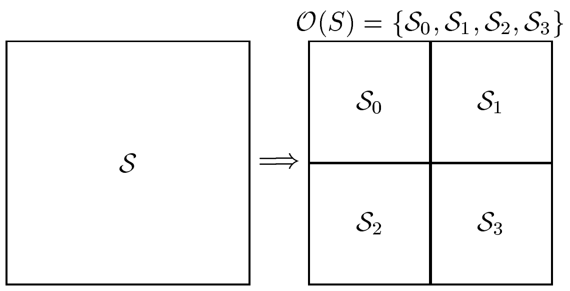

Fig. 4. Partitioning of set $\mathcal{S}$.

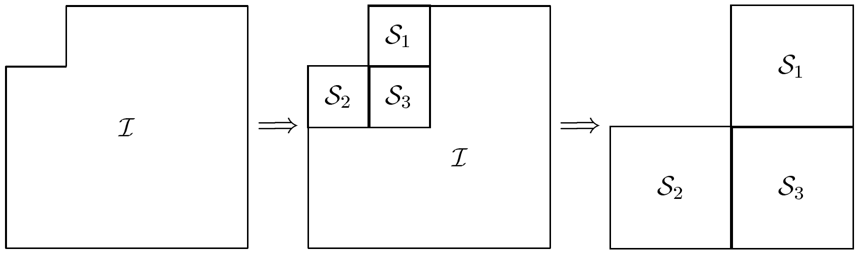

Fig. 5. Partitioning of set $\mathcal{I}$.

The idea behind this partitioning scheme is to exploit the hierarchical pyramidal structure of the subband decomposition, where it is more likely that energy is concentrated at the top most levels of the pyramid and as one goes down the pyramid, the energy content decreases gradually. If a set $\mathcal{I}$ is significant against some threshold $n$, it is more likely that the pixels that cause $\mathcal{I}$ to be significant lie in the top left regions of $\mathcal{I}$. These regions are decomposed into sets of type $\mathcal{S}$, and are put next in line for processing.

In this way, regions that are likely to contain significant pixels are grouped into relatively smaller sets and processed first, while regions that are likely to contain insignificant pixels are grouped into a large set. A single bit may be enough to code this large region against the particular threshold. Hence, once the set $\mathcal{I}$ is partitioned by the octave band partitioning method, the three sets $\mathcal{S}$ are processed in the regular image-scanning order, after which the newly formed reduced set $\mathcal{I}$ is processed.

It should be noted that processing the set $\mathcal{I}$ is a recursive process, and depending on the characteristics of the image, at some point in the algorithm, the set $\mathcal{I}$ will cover only the lower-most (bottom) level of the pyramidal structure. When, at this point, the set $\mathcal{I}$ tests significant against some threshold, it will be broken down into three sets $\mathcal{S}$ but there will be no new reduced set $\mathcal{I}$. To be precise, the new set $\mathcal{I}$ will be an empty set. Hence, the functions `ProcessI`() and `CodeI`() will have no meaning in the SPECK algorithm after this event. This octave band partitioning is done only during the initial sorting pass at the highest significance level $n = n_{max}$.

Once all the sets have been processed for a particular threshold $n$, the refinement pass is initiated which refines the quantization of the pixels in the LSP, i.e., those pixels that had tested significant during the previous sorting passes. Once this is done, the threshold is lowered and the sequence of sorting and refinement passes is repeated for sets in the LIS against this lower threshold. This process is repeated until the desired rate is achieved or, in case of lossless or nearly lossless compression, all the thresholds up to the last, corresponding to $n = 0$, are tested.

It should be mentioned at this point that sometimes embedded bit plane coding, as described, is not needed or desired for various reasons. In this case, one can bypass the refinement passes and output a binary code for the value of a point immediately upon its being found significant. Then there is no need to maintain an LSP. The values sent to the bit stream are still partially ordered according to significance threshold, so the bitstream has a partially embedded character. Furthermore, although the finely embedded property is lost, the encoding and decoding are faster and simpler.

## *C. Processing Order of Sets*

An important step in the execution of the sorting pass comes after the first run of the algorithm. Once one pass has occurred, sets of type $\mathcal{S}$ of varying sizes are added to LIS. During the next lower threshold, these sets are processed in a particular order. The list LIS is not traversed sequentially for processing sets $\mathcal{S}$; rather, the sets are processed in increasing order of their size. In other words, say for a square image, sets of size 1 (i.e. pixels) are processed first, sets of size 4 (blocks of $2\times2$ pixels) are processed next, and so on.

The idea behind this strategy is that during the course of its execution, the algorithm sends those pixels to LIS whose immediate neighbors have tested significant against some threshold $n$ but they themselves have not tested significant against that particular threshold. Chances are, because of energy clustering in the transform domain, that these insignificant pixels would have magnitudes close to the magnitudes of their neighboring pixels already tested significant, although lesser. So it is likely that these pixels will test positive to some nearby lower threshold and add to the reduction in the overall distortion of the coded image.

Moreover, the overhead involved in testing a single pixel and moving it to the LSP is much lower than that involved in testing a group of pixels and moving the significant ones within it to the LSP. Of course, if a whole sorting pass has completed, this scheme offers no advantage since all the sets of type $\mathcal{S}$ in the LIS would be tested in either case. However, if the coding algorithm were to stop in the middle of a sorting pass, as it might if the desired rate is achieved, and the sets in the LIS are processed in increasing order of their size, then we certainly get performance improvement.

It may seem that processing sets of type $\mathcal{S}$ in increasing order of their size involves a sorting mechanism—something which is not desirable in fast implementation of coders. However, there is a simple way of completely avoiding this sorting procedure. Note that the way sets $\mathcal{S}$ are constructed, they lie completely within a subband. Thus, every set $\mathcal{S}$ is located at a particular level of the pyramidal structure. The size of the four offspring $\mathcal{O}(\mathcal{S})$ of a quadrisected set $\mathcal{S}$ corresponds to that of the subbands one level down in the pyramid. Hence, the size of a set $\mathcal{S}$ for an arbitrary image corresponds to a particular level of the pyramid. If we use an array of lists, each corresponding to a level of the pyramid, then each list stores sets of a fixed size. Processing the lists in an order that corresponds to increasing size of sets completely eliminates the need for any sorting mechanism for processing the sets $\mathcal{S}$. Thus, we do not need to compromise the speed of the algorithm by employing some kind of sorting mechanism.

It should be noted that using an array of lists does not place any extra burden on the memory requirements for the coder, as opposed to using a single list. This is because the total number of sets $\mathcal{S}$ that are formed during the coding process remain the same. Instead of storing these sets in one large list, we are storing them in several smaller lists with the aim of speeding up the coding process.

The decoder uses the same mechanism as the encoder. It receives significance test results from the coded bitstream and builds up the same list structure during the execution of the algorithm. Hence, it is able to follow the same execution paths for the significance tests of the different sets, and reconstructs the image progressively as the algorithm proceeds.

A numerical example of the SPECK algorithm, worked out in detail for two complete passes, with coding actions and contents of the LIS and LSP tabulated, may be viewed or downloaded from

`http://www.cipr.rpi.edu/˜pearlman/speck_example.pdf.`

## *D. Entropy Coding*

Entropy coding of the significance map is done using arithmetic coding with simple context-based models. The significance maps are the paths in the quadtree created by the recursive partitioning process. Referring to the coding algorithm of Sec. III, in the function `CodeS()`, the significance test results of the four subsets $\mathcal{O}(\mathcal{S})$ of set $\mathcal{S}$ (see Fig. 4) comprising a quadtree node are not coded separately—rather, they are all coded together first before further processing the subsets. We use conditional coding for coding the significance test result of this four-subset group. In other words, the significance test result of the first subset is coded without any context, while the significance test result of the second subset is coded using the context of the first coded subset, and so on. In this way, previously coded subsets form the context for the subset being currently coded.

Also, we make use of the fact that if a set $\mathcal{S}$ is significant and its first three subsets are insignificant, then this ensures that the fourth subset is significant and we do not have to send the significance test result of the last subset. This fact is utilized in reducing the bit budget. Results have shown that because of the nature of energy clustering in the pyramidal structure, the number of scenarios of the above mentioned type occur slightly more if the four-subset group is coded in reverse-scanning order than in the usual forward-scanning order. This saves some overhead in bit budget and provides corresponding gains.

We have chosen here not to entropy-code the sign and magnitude refinement bits, as small coding gains are achieved only with substantial increase in complexity. The SPECK variant, EZBC [20], has chosen this route, along with a more complicated context for the significance map quadtree coding. The application will dictate whether the increase in coding performance is worth the added complexity.

## *E. Discussion*

The SPECK algorithm is motivated by the features inherent in the SPIHT, SWEET and AGP algorithms, and although it uses ideas from all these coding schemes, it is different from these coders in various respects.

The SPIHT coding scheme works by grouping pixels together in the form of spatial orientation trees. It is well known that a subband pyramid exhibits similarities across its subbands at the same spatial orientation. This property can be seen to be well illustrated if we partition the image transform in the form of spatial orientation trees. The SPIHT algorithm exploits this characteristic of the image transform by grouping pixels together into such structures.

The ASSP (or AGP) algorithm, on the other hand, is a block-based coding algorithm and partitions the image transform in the form of blocks. The blocks are recursively and adaptively partitioned such that high energy areas are grouped together into small sets whereas low energy areas are grouped together in large sets. Such a type of adaptive quadtree partitioning results in efficient coding of the source samples. The SWEET coding algorithm is also block based and uses octave-band partitioning to exploit the pyramidal structure of image transforms.

Whereas SPIHT is a tree-based fully embedded coder which employs progressive transmission by coding bit planes in decreasing order, the AGP and SWEET coding algorithms are block-based coders which are *not* embedded and do not employ progressive transmission. SWEET codes a block up to a certain bit-depth, $n_{min}$, before moving on to the next one. Different rates of compressed images are obtained by appropriately choosing the minimum bit-plane, $n_{min}$, to which to encode. Finer sets of compression ratios are obtained by scaling the image transform by some factor prior to the coefficient coding. The AGP algorithm is a block entropy coder employing quadtree partitioning via a group (node) maximum rule. The node maxima have to be found and encoded for transmission. The quadtree partitioning rule in NQS (Nested Quadratic Split) is similar to that of AGP, but the coding of the coefficients and maxima are different [14]. In AGP, once a partition reduces to a $2\times2$ block, the pixels in the block can be encoded altogether if the maximum is sufficiently small. Otherwise, they can be encoded singly or in two $2\times1$ blocks.

Block-based coding is an efficient technique for exploiting the clustering of energy found in image transforms. It is a known fact that the statistics of an image transform vary markedly as one moves from one spatial region to another. By grouping transform source samples in the form of blocks and coding those blocks independently, one is able to exploit the statistics of each block in an appropriate manner. This is one of the reasons that block-based coders work quite well. However, there is an increasing demand for some desirable properties for image coders, such as embeddedness and progressive transmission, which are very useful and much needed in the fast growing multimedia and networking environment. Both SWEET and AGP, although very efficient, do not possess these desirable properties.

The SPECK coding algorithm solves this problem by exhibiting these important properties lacking in almost all block-based coding schemes. It is a fully embedded block-based coder which employs progressive transmission by coding bit planes in decreasing order. It employs octave-band partitioning of SWEET to exploit the hierarchical structure of the subband pyramid and concentrate more on potentially high energy subbands. It makes use of the adaptive quadtree splitting scheme of AGP to zoom into high energy areas within a region to code them with minimum significance maps. And it uses the significance map schemes of EZW and SPIHT to code the image transform progressively in decreasing bit-plane order. All this makes SPECK a very efficient block-based embedded image coding scheme.

We remark that the EBCOT coding method found in JPEG2000 is also an embedded, block-based coder. It encodes subblocks of wavelet subbands through context-based, adaptive arithmetic coding of bit planes. Other than encoding blocks, the methods of SPECK and EBCOT are quite different. We also remark that these block-based, quadtree methods for encoding subband blocks can also be used for encoding other transform blocks, such as those of the DCT and LOT (lapped othogonal transform), or even original image blocks. AGP coding of DCT and LOT blocks has achieved performance comparable to the best coders [5], [6]. In fact, the results obtained with the DCT were better than those of any known DCT coding method. There is no reason to doubt that any of the block-based, quadtree methods would obtain similarly excellent results.

## *F. Numerical Results*

We present two sets of results: the first with the three standard monochrome, 8 bpp, $512 \times 512$ images, Lena, Barbara, and Goldhill; and the second with four large images from the JPEG2000 test set. We used 5-level pyramids constructed with the $9/7$ tap biorthogonal filters [8] and using a reflection extension at the image edges. The bit rates are calculated from the actual size of the compressed files. Since the codec is embedded, the results for various bit rates are obtained from a single encoded file.

Table I shows the PSNR obtained by this coding method at the rates 0.25, 0.5 and 1.0 bpp for the first set of three images 'Lena', 'Barbara' and 'Goldhill'. These results are obtained by arithmetic coding of the significance decision

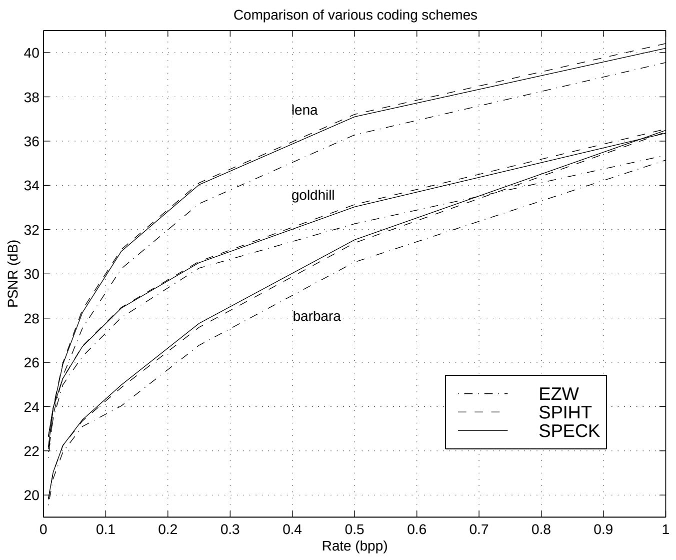

Fig. 6. Comparative evaluation of the new coding method at low rates

TABLE I
COMPARISON OF LOSSY CODING METHODS FOR COMMON TEST IMAGES

| Coding method | PSNR (dB) | | |
|---|---|---|---|
| | 0.25 bpp | 0.5 bpp | 1.0 bpp |
| Lena image ($512 \times 512$) | | | |
| EZW | 33.17 | 36.28 | 39.55 |
| AGP | 34.10 | 37.21 | 40.38 |
| SPIHT | 34.11 | 37.21 | 40.44 |
| SPECK | 34.03 | 37.10 | 40.25 |
| Barbara image ($512 \times 512$) | | | |
| EZW | 26.77 | 30.53 | 35.14 |
| AGP | 27.81 | 31.61 | 36.55 |
| SPIHT | 27.58 | 31.40 | 36.41 |
| SPECK | 27.76 | 31.54 | 36.49 |
| Goldhill image ($512 \times 512$) | | | |
| EZW | 30.31 | 32.87 | 36.20 |
| AGP | 30.53 | 33.13 | 36.53 |
| SPIHT | 30.56 | 33.13 | 36.55 |
| SPECK | 30.50 | 33.03 | 36.36 |

bits output by the coding scheme. A similar result is also included for the EZW, SPIHT and AGP coding schemes, which also employ entropy coding of the significance map.

The rate-distortion curves for SPECK, SPIHT, and EZW are also plotted in Fig. 6 for the three images at rates up to 1 bpp. The curves and tables show that SPECK is comparable to SPIHT and AGP, being slightly worse than both for Lena and Goldhill, but slightly better than SPIHT, and a little worse than AGP for Barbara. In Fig. 7, we show the reconstructions of Barbara for SPECK, SPIHT, and JPEG2000 (VM 8.5) encodings at 0.25 bits/pel. For this image, JPEG2000 has the upper hand, showing slightly more detail in the patterns than SPECK or SPIHT.

For the second set of results in Table II on JPEG2000 test images, we include more rates and comparison to JPEG2000 VM 4.1, generic scalable mode with $64 \times 64$ code blocks, and EZBC. (The EZBC results are taken from [21].) Also included in Table II is SBHP, a lower complexity form of SPECK to be described later. For lossless coding,

TABLE II

COMPARISON OF METHODS FOR LOSSY AND LOSSLESS CODING OF JPEG2000 TEST IMAGES

| Coding method | PSNR (dB) | | | | | | Lossless rate (bpp) |
|---|---|---|---|---|---|---|---|
| | 0.0625 bpp | 0.125 bpp | 0.25 bpp | 0.5 bpp | 1.0 bpp | 2.0 bpp | |
| Bike image (2048 × 2560) | | | | | | | |
| JP2K | 23.74 | 26.31 | 29.56 | 33.43 | 37.99 | 43.95 | 4.520 |
| SBHP | 23.02 | 25.36 | 28.53 | 32.39 | 37.07 | 43.04 | 4.724 |
| EZBC | 23.75 | 26.11 | 29.58 | 33.53 | 38.25 | 44.33 | 4.359 |
| SPIHT | 23.44 | 25.89 | 29.12 | 33.01 | 37.70 | 43.80 | 4.480 |
| SPECK | 23.31 | 25.59 | 28.84 | 32.69 | 37.33 | 43.1 | 4.492 |
| Cafe image (2048 × 2560) | | | | | | | |
| JP2K | 19.03 | 20.77 | 23.10 | 26.76 | 31.96 | 39.01 | 5.384 |
| SBHP | 18.76 | 20.49 | 22.64 | 26.01 | 31.08 | 38.26 | 5.466 |
| EZBC | 19.11 | 20.87 | 23.32 | 27.00 | 32.43 | 39.62 | 5.125 |
| SPIHT | 18.95 | 20.67 | 23.03 | 26.49 | 31.74 | 38.91 | 5.277 |
| SPECK | 18.93 | 20.61 | 22.87 | 26.31 | 31.47 | 38.75 | 5.286 |
| Woman image (2048 × 2560) | | | | | | | |
| JP2K | 25.59 | 27.33 | 29.95 | 33.57 | 38.28 | 43.97 | 4.541 |
| SBHP | 25.26 | 27.09 | 29.59 | 33.11 | 37.98 | 43.69 | 4.636 |
| EZBC | 25.71 | 27.54 | 30.31 | 34.00 | 38.82 | 44.48 | 4.291 |
| SPIHT | 25.43 | 27.33 | 29.95 | 33.59 | 38.28 | 43.99 | 4.419 |
| SPECK | 25.50 | 27.34 | 29.88 | 33.46 | 38.07 | 43.73 | 4.396 |
| Aerial2 image (2048 × 2048) | | | | | | | |
| JP2K | 24.60 | 26.47 | 28.54 | 30.60 | 33.23 | 38.05 | 5.471 |
| SBHP | 24.42 | 26.34 | 28.15 | 30.40 | 33.03 | 37.70 | 5.502 |
| EZBC | 24.76 | 26.65 | 28.70 | 30.79 | 33.49 | 38.51 | 5.203 |
| SPIHT | 24.63 | 26.52 | 28.49 | 30.60 | 33.32 | 38.22 | 5.331 |
| SPECK | 24.60 | 26.49 | 28.45 | 30.59 | 33.25 | 38.26 | 5.259 |

the S+P C filter is used for SPECK and EZBC, while the integer (5,3) is used in VM 4.1 and SBHP. The winner in these results for lossy coding is EZBC, which, you will recall, is a higher complexity version of SPECK that utilizes context-based, adaptive arithmetic coding for sign and refinement bits in addition to the same for the significance map bits. SPECK, SPIHT, and JPEG2000 are fairly comparable, with JPEG2000 showing slightly better performance for the Bike image. We show in Fig. 8 the original Bike image compared to reconstructions from SPECK, SPIHT, and JPEG2000 encoded at 0.25 bpp. Because this image has such high resolution where differences are not discernible, we show in the following Fig. 9 a 284x284 section from the original and these reconstructions. Referring again to Table II, averaged over all rates and the four images, EZBC beats JPEG2000 by 0.25 dB, while JPEG2000 beats SPIHT and SPECK by 0.1 dB and 0.18 dB, respectively. Clearly, these results are quite close. SBHP is the laggard, but only by 0.2 dB from SPECK.

For lossless coding, EZBC is the best overall once again, showing the lowest rate on all four images. The performance order of the methods in terms of efficiency loss from EZBC, averaged over the four images, are SPECK at 2.4 %, SPIHT at 2.8%, JPEG2000 at 4.3%, and SBHP at 7.1%. Surprisingly, considering its considerably higher complexity, JPEG2000 is not as efficient as SPECK or SPIHT and is only more efficient than SBHP for lossless coding.

## IV. COLOR IMAGE CODING

### A. Introduction

This section investigates lossy color embedded image compression with SPECK. Following the philosophy of retaining embeddedness in color representation as has been done with SPIHT, we propose a lossy color embedded image coding scheme using SPECK, called CSPECK. Its performance is evaluated with respect to some of the other famous color-embedded image coders such as Predictive EZW (PEZW), SPIHT and JPEG2000. Here, as in SPIHT, there is no bitstream reorganization needed to achieve color-embedding, as is needed for JPEG2000. Furthermore, no explicit rate allocation among color planes is needed. In order to take advantage of the interdependencies of the color components for a given color space, the set-partitioning scheme of SPECK is extended across the color components and coding is done in such a way as to retain embeddedness.

## *B. Color Spaces*

Images are represented in a tristimulus color space for viewing on a CRT or projection display. The normal tristimulus colors are designated as red (R), green (G) and blue (B), called RGB space, or as cyan (C), yellow (Y), and magenta (M), called CYM space. These color planes are usually highly correlated, so that transformation to a less correlated space is mandatory for efficient compression. The most popular of these are the luminance-chrominance spaces YUV, used mainly in video, and YCrCb, used mostly for still images. The luminance component Y is the same in these two spaces, but the two chrominance components signifying hue and saturation, U and V in one and Cr and Cb in the other, are slightly different. From a compression standpoint, it does not seem to matter which of these two spaces is used. Video sequences are stored in YUV format with the display device doing the work to convert to RGB or CYM format for viewing. Here we shall code directly YUV frames extracted from video sequences. These frames are in CIF or QCIF 4:2:0 format, with sizes $352 \times 288$ and $176 \times 144$ respectively, with U and V planes subsampled by two both in horizontal and vertical directions.

## *C. Color Image Coding: CSPECK*

A simple application of SPECK to a color image would be to code each color space plane separately as does a conventional color image coder. Then, the generated bit-stream of each plane would be serially concatenated. However, this simple method would require bit allocation among color components, losing precise rate control and would fail to meet the requirement of full embeddedness of the image codec, since the decoder needs to wait until the full bit-stream arrives to reconstruct and display. Instead, one can treat all color planes as one unit at the coding stage, and generate one *mixed* bit-stream so that we can stop at any point of the bit-stream and reconstruct the color image of the best quality at the bit-rate. In addition, it will automatically allocate bits optimally among the color planes. By doing so, we still maintain full embeddedness and precise rate control of SPECK. We call this scheme Color-SPECK (CSPECK). The generated bit-stream of both methods is depicted in Fig. 10, where the first one shows a conventional color bit-stream, while the second shows how the color embedded bit-stream is generated, from which it is clear that, except for very early in the bitstream, we can stop at any point and still reconstruct a color image at that bit-rate as opposed to the first case.

Let us consider the color space YUV (4:2:0 format) where the chrominance U and V planes are one-quarter the size of the luminance Y plane. Each plane is separately wavelet transformed (using 9/7 filter) to the same number of decomposition levels. Then each color plane is initially partitioned into sets $\mathcal{S}$ and $\mathcal{I}$ as shown in Fig. 11. In this figure we show two levels of wavelet decomposition for ease of illustration, whereas three levels were used in our simulations. An LIS is maintained for each of the three transform planes, each one initialized with the corner coordinates of its top level $\mathcal{S}$. There is just one LSP list. The coding proceeds as in the normal SPECK algorithm starting with Y, but then crosses to U and then V at the same significance threshold, as depicted in Fig. 11. Starting with the maximal significance level $n$ among the color planes (almost always the maximum in Y), SPECK's first sorting pass proceeds for the whole Y plane. Then at the same $n$, this sorting pass is enacted in turn on the full U and V planes. Significant points among the three planes are mixed in the single LSP. Then, the significance level is lowered to $n-1$, and the LIS sets of Y, U, and V are visited in turn on the three lists for the SPECK sorting passes. After completion of these passes at level $n-1$, the refinement pass takes place on the single LSP by sending the $n$-level bits of the binary expansion of the magnitude of points found significant in previous passes. As before, the procedure repeats at lower significance levels until the bit budget is exhausted or all bits have been sent in the lowest bit plane.

## *D. Simulation and Results*

In this section, extensive simulations on various types of color test images will be performed. Color images are YUV 4:2:0 chrominance downsampled versions from the first frames (intraframes) of standard MPEG-4 test sequences. It seems that there are not many reports of color image coding in the literature, since it has been believed that chrominance components are usually easy to code. Consequently, not much attention has been devoted to set up standard criteria for evaluating color image compression. Notwithstanding, here we report the results of color image compression in terms of PSNR of each color plane. We will compare the CSPECK algorithm with other embedded image algorithms such as Predictive Embedded Zero-tree Wavelet (PEZW), which is an improved version of the original EZW and is currently the standard for the texture mode in MPEG-4 [23], SPIHT and JPEG2000 VM 8.0. The test results of PEZW for various types of color images were obtained via private communication [22]. We tried to match the number of bits, type of filters, and number of decompositions as closely as possible for fair comparisons to those of PEZW. With CSPECK and SPIHT, we were able to match the bit numbers exactly, and came as close as possible with JPEG2000, whose generic scalable mode has coarser granularity than CSPECK or SPIHT. The 9/7 bi-orthogonal Daubechies' filter in [8] is used for the 2-D wavelet decomposition. Our test images are the first frames of MPEG-4 standard test video sequences with QCIF ($176 \times 144$), and CIF ($352 \times 288$) format.

In Tables III and IV, comprehensive simulation results on various color test images at various bit-rates with QCIF and CIF formats are shown, where the results of CSPECK have been compared with SPIHT, JPEG2000 and PEZW on color Akiyo image in QCIF and CIF formats at various bit-rates. In general, CSPECK outperforms other codecs for the luminance Y component with a gain of (0.1–2.7 dB) except for the Coast Guard image. The chrominmance U and V components show somewhat lower PSNR values than the other coders, due to the different nature of bit allocation among the color components in these coders. In CSPECK, we leave the U and V components unscaled, so that their magnitudes alone determine the bit assignments. In SPIHT and other algorithms, there is explicit or implicit scaling to affect the rate allocation among the different color components. But since the human eye is more sensitive to changes in brightness, we claim that the gain in dB for the Y component as obtained from CSPECK yields visually better results. This is in accordance with the images shown in Fig. 12 and Fig 13. Clearly, CSPECK is superior to JPEG2000 in the subjective quality of the reconstructed image. For example, the background shadow edge of Foreman seems to be jagged for JPEG2000, whereas with CSPECK, the shadow line is preserved.

## V. SBHP, a Low Complexity Alternative to JPEG2000

We now describe a SPECK variant called Subband Block Hierarchical Partioning (SBHP) that was originally proposed as a low complexity alternative to JPEG2000. There were other SPECK variants, mentioned in the Introduction, that preceded SBHP, but SBHP captures the essence of those contributions. From a functional point of view, SBHP does exactly the same tasks executed by the entropy coding routines used in every JPEG2000 Verification Model (VM). In consequence, every single feature and mode of operation supported by the VM continues to be available with SBHP. SBHP operates in the EBCOT [1], [10] framework chosen for the JPEG2000 standard. Like EBCOT and other encoders, SBHP is applied to blocks of wavelet coefficients extracted from inside subbands. It produces a fully embedded bit stream that is suitable for several forms of progressive transmission, and for one-pass rate control. Except for the fact that it does not use the arithmetic encoder, it does not require any change in any of the VM functions outside entropy coding.

### *A. SBHP in the JPEG2000 Framework*

The JPEG2000 coding framework was dictated by the necessity to support many features. Among its requirements were the usage of small memory and the capability of encoding images of almost any size. In particular, a line-based wavelet transform method and independent encoding of small subband blocks were identified as essential fairly early in the process. The details of the wavelet transform need not concern us here. The subbands of the wavelet transform of the full image are considered to be divided into square blocks, no larger than typically $64 \times 64$ or $128 \times 128$. Small or odd-size subbands will have smaller or odd-shaped blocks at the boundaries. Anyway, the basic coding unit is a sub-block of a subband, called a code block. In EBCOT coding of these blocks, once the maximal non-all-zero bit plane of a block is found, the context-based, adaptive binary arithmetic coding of this and lower bit planes passes through all coefficients in the code block. As an aside, the maximal bit plane numbers of all code blocks (deemed tag-trees) are encoded by a quadtree coder similar to SPECK and put into the compressed bit stream. The entropy coding in EBCOT is quite complex. Starting from the maximal non-empty bit plane to a low bitplane such as $2^3$ (for lossy coding), every coefficient in the subblock is visited three (originally four) times to determine its logic in the given bit plane and to gather its significance context among its neighbors. The three passses through each bit plane are needed to calculate context for three different logic states and cumulative probabilities for the given context needed for the adaptive arithmetic coding of the bit planes.

TABLE III
COMPREHENSIVE RESULTS OF MPEG-4 TEST COLOR QCIF IMAGES (FIRST FRAMES)

| PEZW | | SPIHT | | JP2K (VM 8) | | CSPECK | |
|---|---|---|---|---|---|---|---|
| bits | PSNR (dB) | bits | PSNR (dB) | bits | PSNR (dB) | bits | PSNR (dB) |
| Akiyo | | | | | | | |
| 10256 | Y: 32.3<br>U: 34.2<br>V: 36.9 | 10256 | Y: 32.6<br>U: 35.0<br>V: 38.3 | 10208 | Y: 30.8<br>U: 37.4<br>V: 38.8 | 10256 | Y: 33.5<br>U: 34.4<br>V: 36.6 |
| 20816 | Y: 37.5<br>U: 39.1<br>V: 41.0 | 20816 | Y: 38.4<br>U: 40.7<br>V: 41.7 | 20768 | Y: 36.9<br>U: 43.3<br>V: 43.8 | 20816 | Y: 39.1<br>U: 38.7<br>V: 39.8 |
| 29240 | Y: 40.8<br>U: 41.5<br>V: 42.6 | 29240 | Y: 41.7<br>U: 43.4<br>V: 44.0 | 29160 | Y: 40.6<br>U: 46.1<br>V: 46.6 | 29240 | Y: 41.9<br>U: 43.1<br>V: 43.6 |
| News | | | | | | | |
| 10536 | Y: 27.5<br>U: 32.5<br>V: 34.2 | 10536 | Y: 27.8<br>U: 33.6<br>V: 34.7 | 10200 | Y: 26.7<br>U: 34.8<br>V: 36.2 | 10536 | Y: 28.6<br>U: 30.8<br>V: 31.6 |
| 20472 | Y: 32.0<br>U: 35.9<br>V: 37.4 | 20472 | Y: 32.5<br>U: 36.6<br>V: 37.8 | 20280 | Y: 31.6<br>U: 38.0<br>V: 39.5 | 20472 | Y: 33.0<br>U: 34.1<br>V: 35.0 |
| 30304 | Y: 35.5<br>U: 38.3<br>V: 39.4 | 30304 | Y: 36.2<br>U: 38.7<br>V: 40.0 | 30176 | Y: 35.3<br>U: 41.5<br>V: 42.2 | 30304 | Y: 36.5<br>U: 37.7<br>V: 38.6 |
| Hall | | | | | | | |
| 10256 | Y: 28.8<br>U: 36.3<br>V: 39.2 | 10256 | Y: 29.3<br>U: 36.2<br>V: 38.8 | 10384 | Y: 28.8<br>U: 37.1<br>V: 40.0 | 10256 | Y: 29.8<br>U: 33.4<br>V: 36.8 |
| 20832 | Y: 34.8<br>U: 38.1<br>V: 40.9 | 20832 | Y: 35.3<br>U: 38.5<br>V: 41.0 | 20608 | Y: 34.6<br>U: 40.3<br>V: 42.3 | 20832 | Y: 35.7<br>U: 36.4<br>V: 38.8 |
| 30448 | Y: 38.5<br>U: 40.0<br>V: 42.5 | 30448 | Y: 39.0<br>U: 41.0<br>V: 42.8 | 30664 | Y: 38.5<br>U: 43.2<br>V: 44.3 | 30448 | Y: 39.3<br>U: 38.6<br>V: 41.0 |

The SBHP coding replaced EBCOT in coding the codeblocks of the subbands. In fact, the SBHP encoder was integrated into VM 4.2, so that one could choose by a command line switch to run either EBCOT or SBHP when encoding the codeblocks. SBHP uses SPECK's octave-band partitioning method on these codeblocks and encodes the $\mathcal{S}$ sets with the quadrature splitting codeS($\mathcal{S}$) procedure of SPECK. Minor differences are that SBHP uses a separate List of Insignificant Pixels (LIP) for insignificant isolated pixels. But the LIP is visited first and then the LIS in order of increasing size sets. Therefore, the two lists LIP and LIS are functionally equivalent to the one LIS list in SPECK.

The partitioning of the codeblock mimics the octave band partitioning in SPECK by starting with a $2 \times 2$ block $\mathcal{S}$ at the upper left with the rest of the block, the $\mathcal{I}$ set. The coding proceeds in the block just as it does for the full-transform SPECK described earlier until the block's target file size is reached. Then the procedure is repeated on the next block until all blocks of the transform in each subband is coded. The subbands are visited in order from lowest to highest frequency in the same order dictated by the octave band splitting in the full-transform SPECK.

When a set is split, the probability that a generated subset is significant is smaller than 1/2. This fact is exploited to reduce the number of compressed bits with simple entropy coding. Since there are four subsets or pixels, we can code them together. We have chosen a Huffman code with 15 symbols, corresponding to all the possible outcomes. (A set is split when at least one of the subset is significant, so not all subsets can be insignificant after splitting.) No type of entropy coding is used to code the sign and the refinement bits. Of course, this results in compression loss, but it is observed that it is very hard to compress these bits efficiently, and nothing is simpler than just moving those "raw" bits to the compressed stream. To optimize the rate-distortion properties of the embedded bit stream we sort the elements

TABLE IV
COMPREHENSIVE RESULTS OF MPEG-4 TEST COLOR CIF IMAGES (FIRST FRAMES)

| PEZW | | SPIHT | | JP2K (VM 8) | | CSPECK | |
|---|---|---|---|---|---|---|---|
| bits | PSNR (dB) | bits | PSNR (dB) | bits | PSNR (dB) | bits | PSNR (dB) |
| Akiyo | | | | | | | |
| 25112 | Y: 34.7 U: 37.7 V: 40.1 | 25112 | Y: 35.3 U: 38.7 V: 40.9 | 24928 | Y: 34.4 U: 41.5 V: 42.1 | 25112 | Y: 35.7 U: 37.3 V: 39.0 |
| 49016 | Y: 39.3 U: 41.3 V: 43.6 | 49016 | Y: 40.3 U: 42.6 V: 44.0 | 49336 | Y: 39.8 U: 45.1 V: 46.2 | 49016 | Y: 40.4 U: 40.7 V: 42.4 |
| 70448 | Y: 42.2 U: 43.2 V: 45.2 | 70448 | Y: 43.2 U: 44.9 V: 46.2 | 70424 | Y: 42.5 U: 47.5 V: 48.3 | 70448 | Y: 43.0 U: 44.4 V: 45.8 |
| Coast | | | | | | | |
| 25248 | Y: 27.9 U: 43.5 V: 45.2 | 25248 | Y: 28.2 U: 41.9 V: 43.3 | 25184 | Y: 28.3 U: 43.9 V: 45.3 | 25248 | Y: 28.3 U: 37.3 V: 42.2 |
| 49312 | Y: 30.2 U: 44.0 V: 45.5 | 49312 | Y: 30.6 U: 43.6 V: 45.5 | 49176 | Y: 30.9 U: 44.5 V: 46.1 | 49312 | Y: 30.6 U: 42.1 V: 43.3 |
| 51600 | Y: 30.4 U: 44.0 V: 45.5 | 51600 | Y: 30.7 U: 43.6 V: 45.5 | 50864 | Y: 31.1 U: 44.6 V: 46.1 | 51600 | Y: 30.8 U: 42.1 V: 43.3 |
| News | | | | | | | |
| 25528 | Y: 29.3 U: 34.5 V: 36.3 | 25528 | Y: 29.7 U: 34.5 V: 36.4 | 25224 | Y: 29.1 U: 36.3 V: 38.1 | 25528 | Y: 30.0 U: 32.8 V: 34.3 |
| 50168 | Y: 33.6 U: 37.6 V: 39.1 | 50168 | Y: 34.3 U: 37.5 V: 38.9 | 50544 | Y: 33.9 U: 40.0 V: 41.1 | 50168 | Y: 34.4 U: 35.6 V: 37.1 |
| 70912 | Y: 35.9 U: 40.1 V: 41.5 | 70912 | Y: 37.1 U: 39.7 V: 40.7 | 70400 | Y: 36.5 U: 42.6 V: 43.2 | 70912 | Y: 37.0 U: 39.1 V: 40.3 |

in the LIS, LIP and LSP.

- LSP and LIP: pixels added first are coded first (FIFO).
- LIS: sets with smallest number of elements are processed first. When sets have the same number of elements, those added first are coded first.

The FIFO is actually the most efficient for list management. As described in SPECK earlier, the sorting of LIS sets by size is accomplished by keeping several sub-lists.

## *B. Rate-Distortion Optimization*

Since the codeblocks are encoded independently, they are all separately embedded. However, if their codes were put sequentially into the compressed bitstream, the composite bitstream would no longer be embedded. But if the bits belonging to the same threshold from every codeblock are put into the bitstream starting from the highest to the lowest threshold, then the composite bitstream would be embedded. The JPEG2000 platform takes care of this bitsteam reorganization.

The other issue is the size of each codeblock's bitstream or rate control. The wavelet coefficient magnitude distribution will vary among the codeblocks, so each will contribute a different number of bits in order to minimize the distortion for a given overall code rate. The one-pass bit-rate control in VM 4.2 requires sorting the data according to its rate distortion properties. The coding function returns the increase in bit rate and the decrease in distortion for each bit-plane coding pass. The computation of the number of bits is a trivial matter, but the computation of decrease in distortion is not. Exact computation of the squared-error would require computation of square values for each coded pixel.

SBHP can use two properties to simplify this computation. First, the derivative of the rate-distortion function (required to be equal among the nonzero rate blocks for optimality) can be predicted with high precision at the beginning of each refinement pass (where exactly one bit is used for each pixel and the distortion reduction pretty much follows a simple statistical model). The second property comes from the fact that each coding action corresponds to a list entry. It happens that the average reduction in distortion can be reliably estimated as a function of the number of elements in the LIS, LIP and LSP.

For each bit plane $n$, SBHP computes rate-distortion information at three points in the coding process. This information is the total number of bits ($B_{i,n}$) used so far, and the average decrease in distortion *per coded bit* (derivative of the rate distortion curve, $\delta D_{i,n}$). For that purpose SBHP also records $P_{i,n}$, the number of pixels in the LSP.

The first point, corresponding to the moment when SBHP finishes coding pixels in the LIP, is represented by $(B_{0,n}, P_{0,n})$; the second, when all sets in the LIS had been coded is $(B_{1,n}, P_{1,n})$, and the third, at the end of the refinement pass, is $(B_{2,n}, P_{2,n})$.

For a threshold $2^n$, we found that $\delta D_{i,n}$ can be approximated by

$$\begin{aligned} \delta D_{0,n} &= -2.08(P_{0,n} - P_{2,n+1})2^{2n}/(B_{0,n} - B_{2,n+1}) \qquad (3)\\ \delta D_{1,n} &= -1.85(P_{1,n} - P_{0,n})2^{2n}/(B_{1,n} - B_{0,n}) \qquad (4)\\ \delta D_{2,n} &= -0.24P_{2,n+1}2^{2n}/(B_{2,n} - B_{1,n}) \qquad (5)\end{aligned}$$

The numerical constants are found by experimental refinements from an initial rate-distortion model. Linear interpolation is used to estimate the derivative of the rate-distortion for any bit in the embedded bit-stream. For example, we can use

$$\delta D(B) = \delta D_{i,n} + \frac{(B - B_{i,n})(\delta D_{i+1,n} - \delta D_{i,n})}{(B_{i+1,n} - B_{i,n})}, \quad \text{if } B_{i,n} \le B \le B_{i+1,n}. \qquad (6)$$

A unidimensional search for the optimal value in the derivative ($\delta D(B)$) is used to find the optimal truncation of the bit-streams ($B$), which are then combined in a single file with the desired size [1].

## C. Complexity Analysis

The SBHP (or SPECK) encoder first visits all pixels to gather information about bits in all bit planes (preprocess pass). This pass is actually quite simple, requiring one bitwise OR operation per pixel, following a predetermined sequence, and some analysis of partial results. All other bit-plane coding algorithms must compute the same data to determine the first significant bit plane with at least one non-zero bit.

The set-partitioning process uses exactly one bit to indicate when all bits in a group inside a bit plane are equal to zero. The information about these groups is gathered in the preprocess pass, so only one comparison is required (the decoder just reads the bit) per bit in the compressed stream. This property can be used to easily show how SBHP minimizes the coding complexity and how it is asymptotically optimal.

We can roughly measure the complexity of coding a bit plane by counting the number of bit comparisons (equal to 0 or 1?) used to test the bits in the bit plane. A direct method of compression needs to visit all pixels, so its complexity is proportional to the number of pixels (multiple passes may increase complexity proportionally). SBHP, on the other hand, tests only the elements in its lists. The absolute minimum number of comparisons is unknown, but can be computed by the entropy of the bit plane. Since SBHP uses one compressed bit per comparison, and the number of bits generated per bit plane is equal to the number of comparisons, we can conclude that its number of comparisons is very near the optimal (or we would not have good compression).

Only the most basic operations, like memory access, bit shifts, additions, and comparisons are required by the encoder/decoder. No multiplication or division is required (even in approximate form), simplifying a hardware implementations.

Code profiling has shown that the computational effort is well distributed among the tasks of data access, list management, and writing raw bits to the compressed stream.

The fastest possible hardware and software implementations are achieved with the non-embedded mode of the coder [17]. In this way, there is no need for multiple passes within a block. This form of coding can run in approximately the same time as baseline JPEG, around 11 times faster than the VM 4.2 on the decoder side. One good reason for that is that the largest Huffman code is of length 6 bits and we can use lookup tables instead of binary trees to decode.

The decoder is usually faster than the encoder. The encoder always needs to visit all pixels in a block, unlike the decoder, which can skip over large blocks of zero coefficients.

The complexity analysis of SBHP can be divided in two parts: dependent and independent of the bit rate. The last one is related to the time to preprocess a block before encoding or decoding, and is not really related to entropy coding. The encoder needs one pass to identify the maximum magnitude values of all sets. Each pixel has to be visited only once. A bitwise OR operation is necessary for each pixel, and for each set. The number of sets in SBHP is 1/3 the number of pixels, so we need about 4/3 accesses per pixel. (All bit-plane coders need a similar pass to identify the top bit-plane.) The key point for evaluating the coding complexity of SBHP is the fact that all time-related complexity measures for the algorithm, like number of operations, clock cycles, memory access, etc., are proportional to the number of compressed bits. Our experiment shows that this is indeed a good approximation.

Three facts are important in the list-management complexity analysis. First, the algorithm works with small blocks, so the list memory can be assigned in advance, and no time-consuming memory control is required. Second, the lists are updated in FIFO mode, so they are stored simply as arrays. Third, the lists grow exponentially for each bit-plane pass, and for all bit rates the complexity is mostly determined by the last pass. In other words, even if coding requires several passes, the complexity is typically less than a two-pass (per block) algorithm (and much less than several passes per-bit plane).

It is easy to evaluate the complexity of testing elements in the LSP: for each entry, a bit of a wavelet coefficient is moved to the compressed bit stream. There is no need to move elements in or out of the list. Processing the elements in the LIP is not much more complex. If a magnitude bit (significance) is zero, then the entry stays in the LIP. Otherwise, the coefficient sign is written, and the entry moves to the LSP.

For the most common bit rates, most of the computational effort is spent processing the LIS. Here there is a more pronounced asymmetry depending on the significance test. If a coefficient is insignificant, then it is just left on the LIS. Otherwise, it has to be partitioned into four subsets, with additional operations. If the set is not decomposed into individual pixels, then only new set entries have to be added to the LIS. The decomposition to individual pixels may require coding of the sign bit.

## *D. Test Results*

The SBHP implementation, in the embedded version is substantially faster than VM 4.2. On the encoder side it is around 4 times faster on both PA-RISC and Pentium-II processors. On the decoder side it is around 6 times faster on the PA-RISC platform and around 8 times faster on the Pentium-II processor. Of course those numbers vary depending on the image and the bit rate. In the non-embedded version of the algorithm the decoder can be as much as 11 times faster on the Pentium-II, in which case the complexity of SBHP becomes very close to that of baseline JPEG.

We have already shown in Table II coding results of SBHP among those of the other state of the art coders on four JPEG2000 test images at various rates. SBHP is the simplest of these coders, yet comes close to their PSNR performance, as already noted. We provide now experimental results for 7 different 8-bit gray scale images, (Aerial2, Bike, Cafe, Woman, Gold, Hotel, Txtr2) from the JPEG2000 test set, comparing the performance of SBHP with respect to the verification model (VM 4.2). The characteristics of the first four are presented in Table II. Gold is the full 720x576 Goldhill, Hotel is a 720x576 picture of a resort hotel, and Txtr2 is a 1024x1024 aerial image of lush farm and forest land. We use the bit allocation algorithm described above to truncate the bit stream to the desired rate after compression. The results can be seen in Table V. For other types of images such as compound documents the results may not be as good. But for compound documents JPEG2000 is not a good choice; approaches such as the one in [19] based on segmentations are needed.

Analysis of the experimental results shows that for most images, such as photographic, medical, etc., the SBHP PSNR is only about 0.4-0.5 dB below the VM. As an alternative way to compare the compression ratio loss, we measure SBHP file sizes compared to the VM, for the same quality. The numbers show that SBHP loses only 5-10% in bit rate for lossy compression and, only 1–2% for lossless compression for photographic images.

The results are more favorable when SBHP is used in conjunction with simple filters (5,3). In this case the average loss with respect to VM 4.2, which varies with the bit rate, is no more than 0.5 dB. Interestingly enough this configuration leads to the lowest possible complexity both in terms of memory and in terms of numerical operations. The

TABLE V

AVERAGE RESULTS FOR SBHP ON SEVEN TEST IMAGES COMPARED TO JPEG2000.

| Rate (bpp) | Decrease in PSNR (dB) | | Increase in bit rate (%) | |
|---|---|---|---|---|
| | 5×3 filters | 9×7 filters | 5×3 filters | 9×7 filters |
| 0.0625 | 0.28 | 0.33 | 5.85 | 8.08 |
| 0.125 | 0.32 | 0.36 | 5.89 | 7.71 |
| 0.25 | 0.36 | 0.41 | 5.64 | 7.06 |
| 0.5 | 0.41 | 0.46 | 5.29 | 6.45 |
| 1 | 0.41 | 0.46 | 4.37 | 5.39 |
| 2 | 0.37 | 0.43 | 3.12 | 3.90 |
| Lossless | NA | NA | 1.13 | NA |

image "Bike" gives the worst comparative results with "Aerial2" giving the best. For lossless compression, there is an average loss anywhere from 1% to 2%.

## VI. CONCLUSIONS

We introduced a new set-partitioning, block-based embedded image coding scheme, the SPECK image coder. The key to the algorithm is to properly and efficiently sort sets of transform coefficients according to their maximum magnitude with respect to a sequence of declining thresholds. The sorting mechanism is partitioning by quadrisection guided by threshold significance tests. This allows processing of different regions of the transformed image based on their energy content. The SPECK image coding scheme has all the properties desirable in modern image compression schemes. In particular, it exhibits the following properties:

- It is completely embedded – a single coded bitstream can be used to decode the image at any rate less than or equal to the coded rate, to give the best reconstruction of the image possible with the particular coding scheme.
- It employs progressive transmission – source samples are coded in decreasing order of their information content.
- It has low computational complexity – the algorithm is very simple, consisting mainly of comparisons, and does not require any complex computation.
- It has low dynamic memory requirements – at any given time during the coding process, only one connected region (e.g., a $32 \times 32$ block lying completely within a subband) is processed.
- It has fast encoding/decoding – this is due to the low-complexity nature of the algorithm, plus the fact that it can work with data that fits completely in the CPU's fast cache memory, minimizing access to slow memory.
- It is efficient in a wide range of compression ratios – depending on the choice of the transform, the algorithm is efficient for lossless and nearly lossless coding, as well as for the extreme ranges of lossy image coding, in all cases comparable to low-complexity algorithms available today.

Although the algorithm was stated and tested here for the wavelet transform, it can be used effectively for coding blocks of a transform of any kind or even image blocks. Different implementations of back-end entropy coding were shown to give gains in compression efficiency at the expense of greater computational load. We have concentrated here on the simpler entropy coding realizations, SBHP and the original SPECK, which encode only the significance map with fixed Huffman and adaptive arithmetic coding, respectively. Both gave very good results competitive with the very best yet attained, and were very fast in encoding and decoding. Even EZBC, the most complex with context-based, adaptive arithmetic coding of sign, refinement, and significance map bits, surpasses JPEG2000 in all cases and is still less complex than JPEG2000, because it does not pass through all pixels several times in each bit plane. We explained the operation of SPECK in various modes of computational and memory complexity and presented extensive coding simulation results for lossy and lossless coding of monochrome images and lossy coding of color images. State of the art performance and features of embeddedness and scalability were achieved with relatively low complexity.

## ACKNOWLEDGMENTS

We would like to thank Charles Creusere of the University of New Mexico for supplying us with numerical results of the EZW algorithm on the Goldhill image. We also wish to thank our colleagues in Hewlett-Packard Research Labs, Christos Chrysafis and Alex Drukarev, for their work in the implementation of SBHP for presentation to the JPEG2000 Working Group.

## References

[1] D. Taubman, “High Performance Scalable Image Compression with EBCOT,” *IEEE Trans. on Image Processing*, vol. 9, pp. 1158-1170, July 2000.

[2] A. Munteanu, J. Cornelis, G. Van der Auwera, and P. Cristea, “Wavelet Image Compression - The Quadtree Coding Approach,” *IEEE Trans. on Information Technology in Biomedicine*, vol. 3, pp. 176–185, Sept., 1999.

[3] J.M. Shapiro, “Embedded image coding using zerotress of wavelet coefficients,” *IEEE Trans. on Signal Processing*, vol. 41, pp. 3445–3462, Dec. 1993.

[4] A. Said and W.A. Pearlman, “A new, fast and efficient image codec based on set partitioning in hierarchical trees,” *IEEE Trans. Circuits and Systems for Video Technology*, vol. 6, pp. 243–250, June 1996.

[5] A. Said and W.A. Pearlman, “Low-Complexity Waveform Coding via Alphabet and Sample-Set Partitioning,” *Visual Communications and Image Processing ’97*, Proceedings of SPIE, vol. 3024, pp. 25-37, Feb. 1997.

[6] X. Zou and W. A. Pearlman, “Lapped Orthogonal Transform Coding by Amplitude and Group Partitioning,” in *Applications of Digital Image Processing XXII*, Proceeedings of SPIE, vol. 3808, pp. 293–304, 1999.

[7] J. Andrew, “A simple and efficient Hierarchical Image coder,” *IEEE Internationall Conf. on Image Proc. (ICIP-97)*, vol. 3, pp. 658–661, Oct. 1997.

[8] M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies, “Image coding using wavelet transform,” *IEEE Trans. on Image Processing*, vol. 1, pp. 205–220, April 1992.

[9] C. Christopoulos, “JPEG2000 Verification Model 4.1,” *ISO/IEC/JTC1/SC29*, WG1 N1286, June 1999.

[10] D. Taubman, “Embedded, independent block-based coding of subband data,” *ISO/IEC JTC 1/SC29*, WG1 N871, July 1, 1998.

[11] W. A. Pearlman and A. Islam, “Brief report on testing for the JPEG2000 core experiment,” Appendix B in “Compression vs. Complexity Tradeoffs for Quadratic Splitting Systems,” TeraLogic, Inc. (C. Chui, J. Spring, and L. Zhong), CISRA (J. Andrew), and RPI (W. Pearlman), *ISO/IEC/JTC1/SC29*, WG1 N873, June 24, 1998.

[12] A. Islam and W. A. Pearlman, “Set Partitioned Sub-Block Coding (SPECK), *ISO/IEC/JTC1/SC29*, WG1 N1188, March 17, 1999.

[13] J. Spring, J. Andrew, and F. Chebil, “Nested Quadratic Splitting,” *ISO/IEC/JTC1/SC29*, WG1 N1191, July 1999.

[14] C. Chui, J. Spring, and L. Zhong, “Complexity and memory usage of quadtree-based entropy coding (QEC),” *ISO/IEC/JTC1/SC29*, WG1 N770, Mar. 20, 1998.

[15] W. A. Pearlman, “Presentation on Core Experimant CodEff 08: Set Partitioned Embedded Block Coding (SPECK),” *ISO/IEC/JTC1/SC29*, WG1 N1245, March 17, 1999.

[16] A. Islam and W. A. Pearlman, “An embedded and efficient low-complexity hierarchical image coder,” *Visual Communications and Image Processing ’99*, Proceedings of SPIE, vol. 3653, pp. 294–305, Jan. 1999.

[17] C. Chrysafis, A. Said, A. Drukarev, W.A Pearlman, A. Islam, and F. Wheeler, “Low complexity entropy coding with set partitioning,” *ISO/IEC/JTC1/SC29*, WG1 N1313, July 1999.

[18] C. Chrysafis, A. Said, A. Drukarev, A. Islam, and W.A Pearlman”, “SBHP - A Low complexity wavelet coder,” *IEEE Int. Conf. Acoust., Speech and Sig. Proc. (ICASSP2000)*, vol. 4, pp. 2035–2038, June 2000.

[19] A. Said and A. Drukarev, “Simplified Segmentation for Compound Image Compression,” *IEEE Int. Conf. on Image Processing ’99*, vol. 1, pp. 229–233, Oct. 1999.

[20] S-T. Hsiang and J. W. Woods, “Embedded Image Coding Using Zeroblocks of Subband/Wavelet Coefficients and Context Modeling,” *IEEE Int. Conf. on Circuits and Systems (ISCAS2000)*, vol. 3, pp. 662–665, May 2000.

[21] S-T. Hsiang, *Highly Scalable Subband/Wavelet Image and Video Coding*, Ph.D. thesis, Electrical, Computer and Systems Engineering Dept., Rensselaer Polytechnic Instute, Troy, NY 12180, USA, Jan. 2002.

[22] J. Liang, *Private communication*, Texas Instruments, Inc., Dallas, TX, USA.

[23] S. A. Martucci, I. Sodagar, T. Chiang, and Y-Q. Zhang, “A zerotree wavelet video coder,” *IEEE Trans. Circuits & Systems for Video Technology*, vol. 7, pp. 109–118, Feb. 1997.

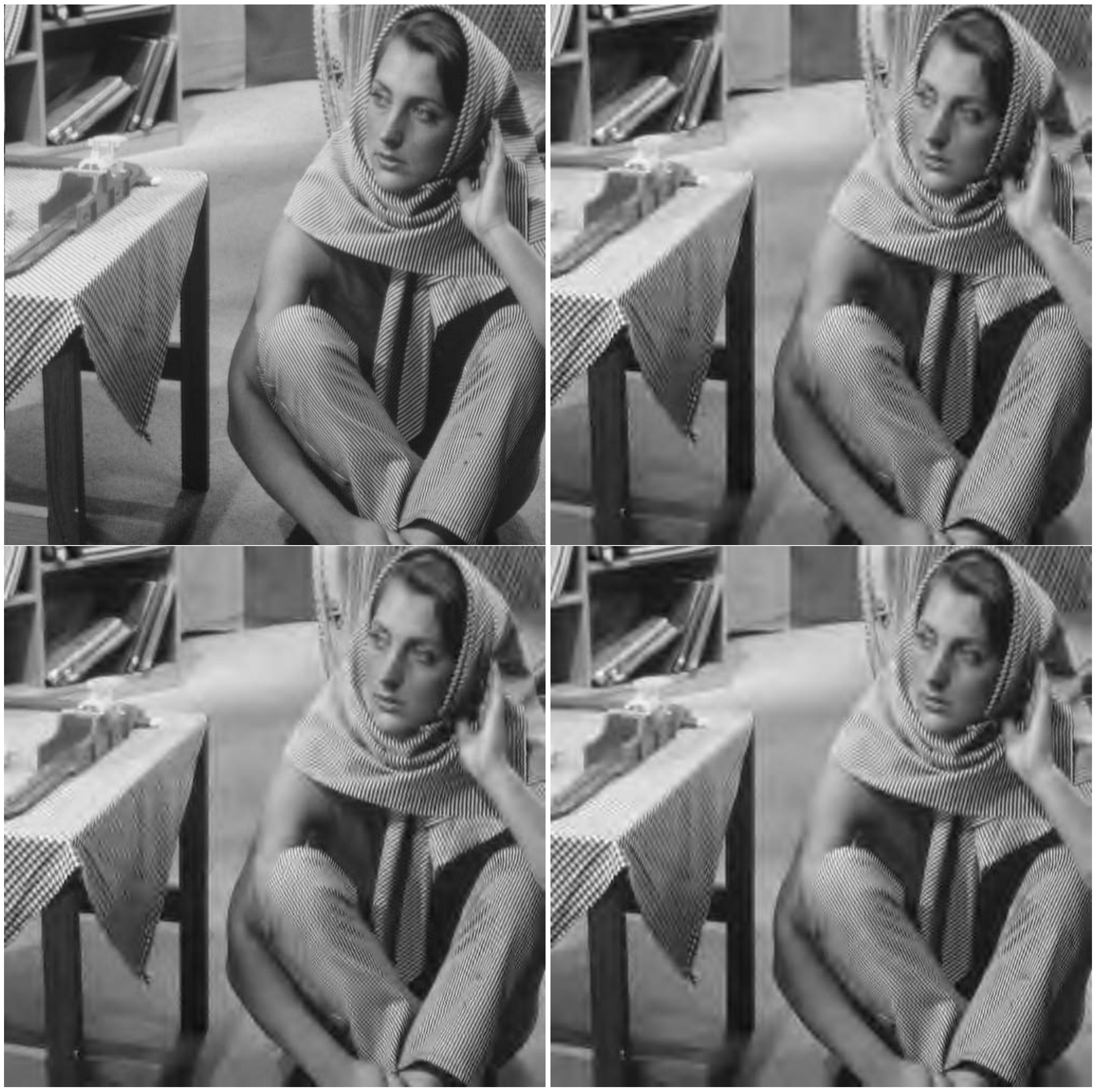

Fig. 7. Comparative coding results of Barbara image at 0.25 bpp: Original (top left), SPECK (top right), SPIHT (bottom left), and JPEG2000 (bottom right).

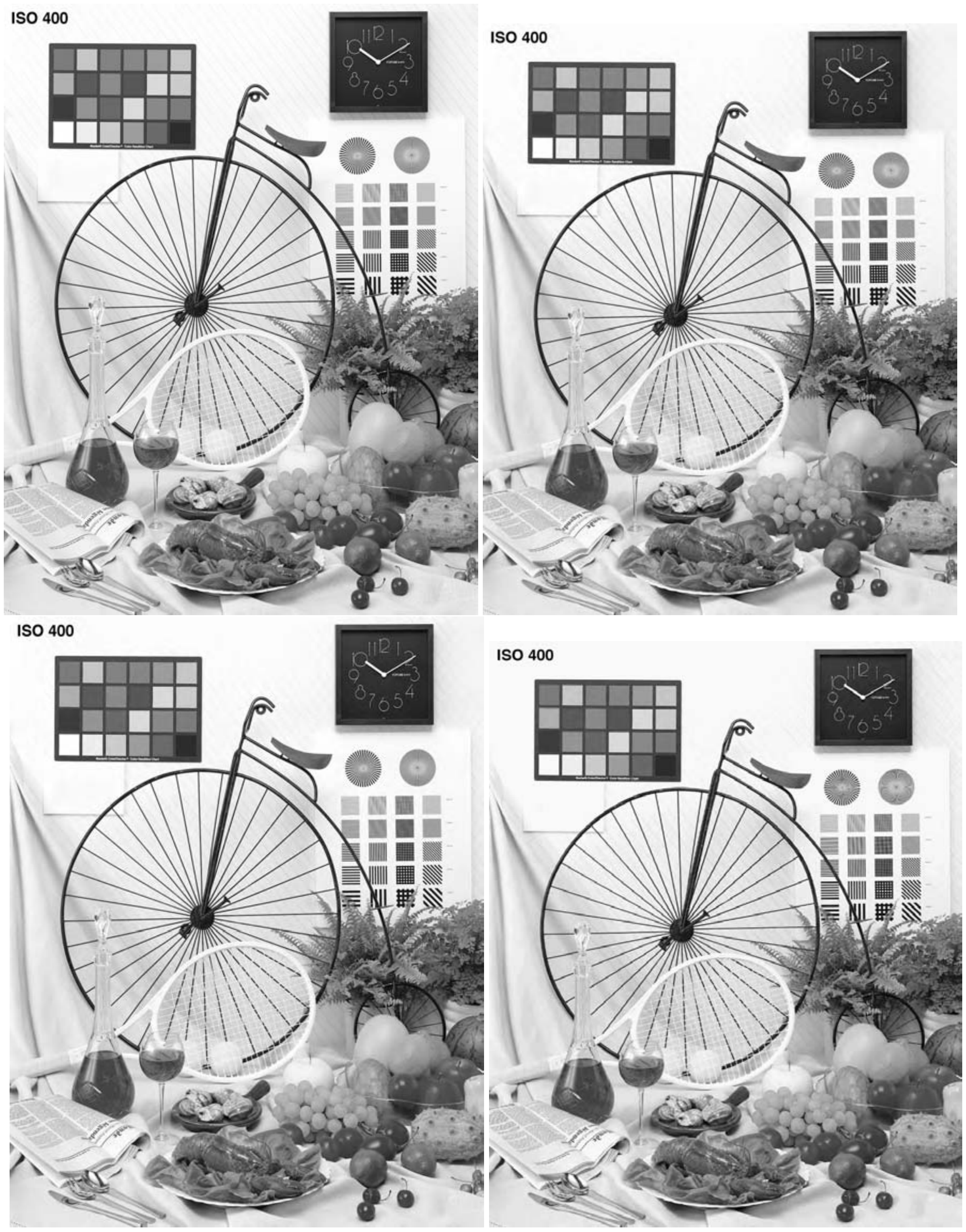

Fig. 8. Comparative coding results of Bike image at 0.25 bpp: Original (top left), SPECK (top right), SPIHT (bottom left), and JPEG2000 (bottom right).

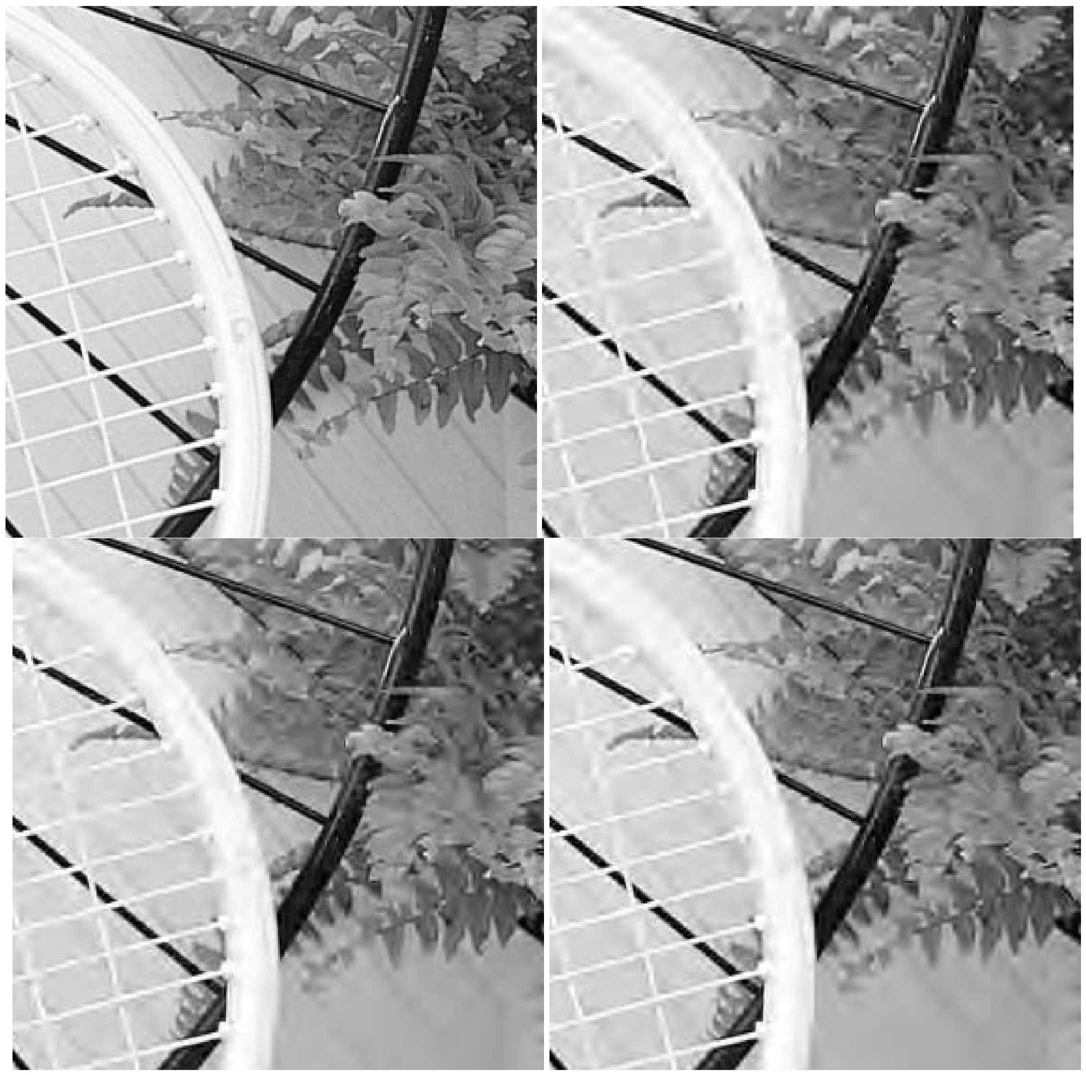

Fig. 9. Comparison of same 284x284 section in original and coded Bike images in Fig. 8: Original (top left), SPECK (27.23 dB PSNR)(top right), SPIHT (27.36 dB PSNR)(bottom left), and JPEG2000 (27.78 PSNR)(bottom right).

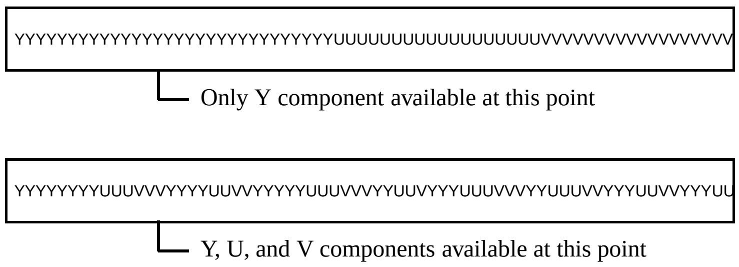
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYUUUUUUUUUUUUUUUUUUVVVVVVVVVVVVVVVVVV
Only Y component available at this point
YYYYYYYYUUUVVVVYYYYUUVVYYYYYUUUVVVYYUUVYYYUUUVVVYYUUUVVYYYYUUVVYYYYUU
Y, U, and V components available at this point

Fig. 10. Compressed color bitstreams: conventional (top) and embedded (bottom).

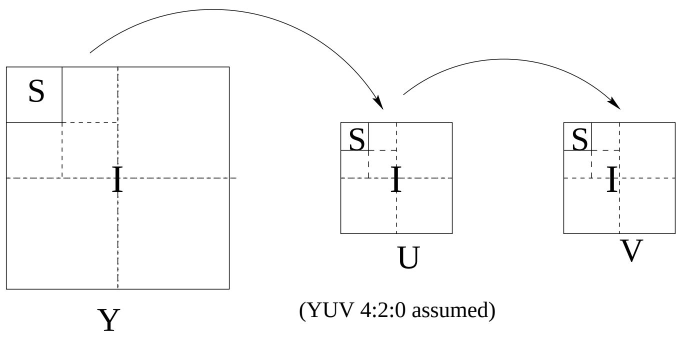

Fig. 11. Set partitioning and color plane traversal for CSPECK. Coding proceeds in order of Y, U, and V at the same significance threshold.

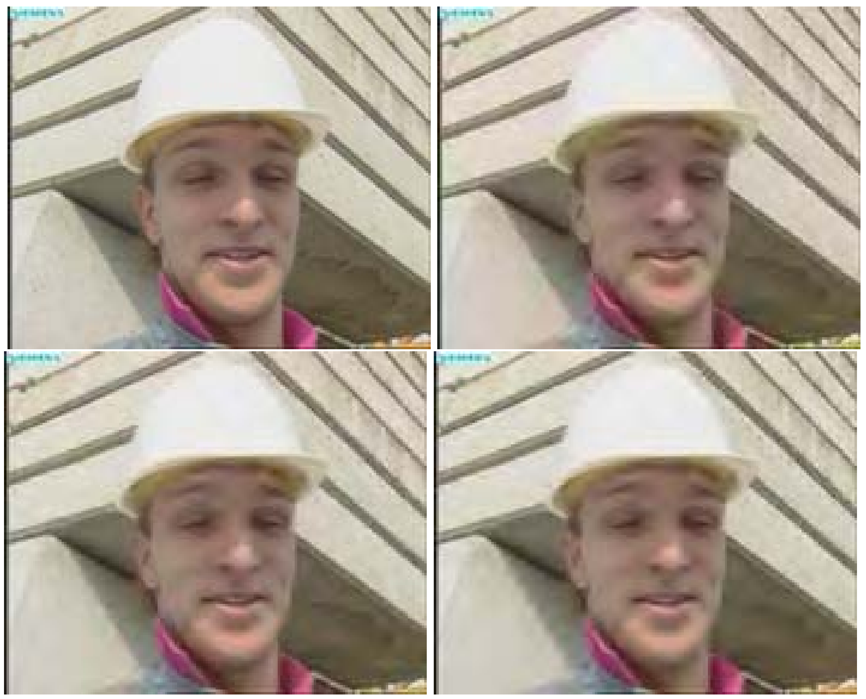

Fig. 12. Reconstructions of first frame of Foreman sequence compressed to 19,300 bits: Original (top left), CSPECK (top right), SPIHT (bottom left), and JPEG2000 (bottom right).

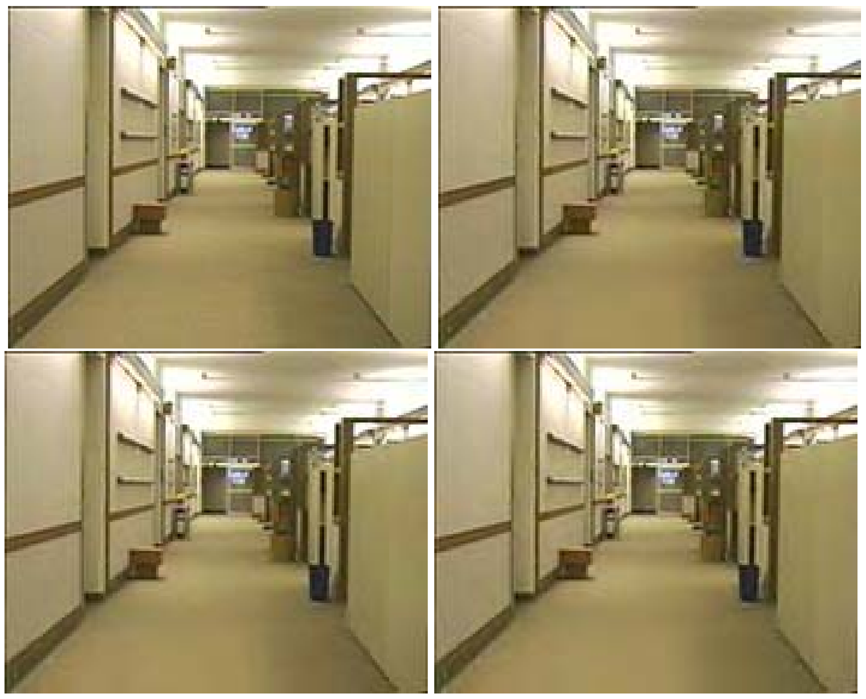

Fig. 13. Recontructions of first frame of Hall Monitor sequence compressed to 28,500 bits: Original (top left), CSPECK (top right), SPIHT (bottom left), and JPEG2000 (bottom right).