Efficient Scalable Image Compression Algorithms with Low Memory and Complexity

Ali Kadhim Jaber Al-Janabi, PhD
Department of Electrical Engineering
Faculty of Engineering
University of Kufa
Najaf, Iraq

Abdulkareem Abdulrahman Kadhim, PhD
College of Information Engineering
Al-Nahrain University
Baghdad, Iraq

ABSTRACT
The set partitioning embedded block (SPECK) image compression algorithm has excellent performance, low computational complexity, and produces a rate scalable compressed bitstream that can be decoded efficiently at multiple bit-rates. Unfortunately, it consumes a huge amount of computer memory due to employing lists that store the coordinates of the image pixels and the coordinates of the sets that are generated during the coding process. In addition, it has complex memory management due to using an array of random access linked lists to store these sets according to their sizes. In this paper, we propose two algorithms that are based on SPECK. The main contribution of the first algorithm is that, as compared to SPECK, the amount of the algorithm's usable memory is reduced to about 75% and at the same time its processing speed is increased and its rate distortion efficiency is preserved as will be demonstrated. The second algorithm has higher processing speed but has slightly lower rate distortion performance than the first algorithm.

General Terms
Image Compression, Image Coding.

Keywords
DWT, Embedded Coding, Low Memory Scalable Image Compression, Set Partitioning algorithms, SPECK, SPIHT, Wavelet-based Image Compression

1. INTRODUCTION
Modern image compression algorithms are transform-based by which the image is transformed to compact most of the pixels' energies into a few numbers of transformed coefficients. As such the other coefficients tend to have small magnitudes. After quantization, most of these small magnitude coefficients will be zero. The coding stage then attempts to code these zero valued coefficients efficiently [1]. A rate scalable image compression algorithm produces an embedded bitstream that can be truncated at any point while attempting to maintain the best possible image quality for the selected bit-rate [2]. An embedded bitstream can be achieved using successive approximation bit-plane coding where all coefficients are coded on bit-plane basis rather than on pixel basis. That is, the coefficients are coded bit-plane by bit-plane starting from the most significant bit (MSB) to the least significant bit-plane (LSB). In this way, the most effective bit (the MSB) of each coefficient is put in the bitstream before its less effective bit (the LSB) [3]. In addition, in order to the bitstream to be an optimal embedded bitstream in the sense of obtaining the best rate distortion (R-D) performance at any truncation point, its information bits must be ordered in decreasing order of distortion reduction, i.e., within each bit-plane level, the bits that have the steepest R-D slope, defined as the amount of distortion reduction per coded bit, should be put first in the bitstream followed by the bits with the less R-D and so on. This is known the embedding principle [4, 5].

Most scalable image compression algorithms employ the two dimensional discrete wavelet transform (2D-DWT) because – in addition to its good energy compaction – it has interesting features namely, its space-frequency localization, and its multi resolution properties. The localization property permits to apply the DWT to the entire image facilitating rate scalability. The multi-resolution nature of DWT facilitates resolution scalability that permits to reconstruct the image at several sizes. The 2D-DWT first decomposes the image into four subbands labeled LL_1, LH_1, HL_1, and HH_1. Then the LL_1 subband is further decomposed into four subbands labeled LL_2, HL_2, LH_2, and HH_2. This dyadic decomposition can be performed L times resulting in 3L+1 subbands. Figure 1 depicts the dyadic 2D-DWT for three decomposition levels (L = 3). It can be shown that the best DWT efficiency can be achieved with L = 5 [2].

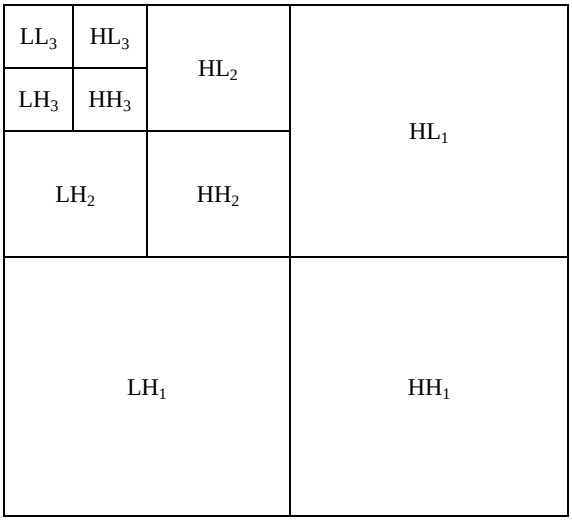

Fig 1: dyadic 2D-DWT with 3 decomposition levels

Among all wavelet-based rate scalable image compression algorithms, the set partitioning in hierarchical trees (SPIHT) [6], and the set partitioning embedded block (SPECK) [7] are the most popular ones. They are very efficient and employ a low complexity set partitioning technique to jointly code the zero bits using some kind of significance testing for sets of coefficients. Both algorithms make use of three linked lists termed the list of insignificant pixels (LIP), list of significant pixels (LSP), and list of insignificant sets (LIS) to store the (i, j) coordinates of the pixels and sets that need to be tested and coded. However, the algorithms differ by the way the sets are formed and partitioned. SPIHT uses the spatial orientation trees that exploit the correlation between wavelet coefficients across the different subbands. In contrast, SPECK uses quad

trees that exploit the correlation between wavelet coefficients within the same subbands. In addition, in SPECK, the generated sets have different sizes. Thus SPECK replaces LIS by an array of smaller lists each containing sets of a fixed size. That is, LIS is replaced by an array of linked lists LIS_z, z = 2, 4... Z, where Z is the size of the largest set. In any coding pass, LIS_z is processed in increasing order, i.e. LIS_2, LIS_4… LIS_Z. In this way information ordering is preserved because the probability of finding significant pixels is inversely proportional to the set sizes [7].

Unfortunately, these algorithms suffer from the high memory requirements and the complex memory management due to using these lists [8, 9]. In particular, LIP and LSP which store the individual pixels, dominate the total memory. In addition, the actual size of each one of LSP, LIP and each one of the lists in LIS_z array depends on the image size, image type and on the compression bit-rate. Therefore, the algorithm must either initialize each list to the maximum size or use the dynamic memory allocation technique. It is well known that the former solution increases the memory requirements of the algorithm while the latter slows the algorithm due to its high computational complexity [10, 11]. On the other hand, these lists are the main reason of the algorithm's reduced complexity, and its good embedding performance. The reduced complexity is mainly due to processing only the pixels and sets that are stored in these lists. The good embedding performance is attained because the pixels and sets are stored in these lists following a specific order which is consistent with the embedding principle [12]. Therefore, removing the lists may lead to increase the algorithm's complexity and/or reduce its efficiency.

2. RELATED WORK

Several works attempted to reduce the memory requirements of the wavelet-based rate scalable algorithms. Latte et al. [12] presented a low memory SPECK called Listless SPECK (LSK). It stores the DWT image coefficients in a 1-D array. In addition, each coefficient has 4 bits marker to determine its status. Unfortunately, the cost of this memory reduction is the increase of algorithm's complexity due to the repeated search and test for all coefficients twice per every bit-plane coding pass. Senapati [13] presented the listless embedded block partitioning (LEBP) algorithm. LEBP also has about the same memory as LSK but it improved the performance of LSK especially at very low bit-rates by testing a cluster of subbands instead of individual subbands. However, this increased the coding time by about 1.3 times as compared to LSK. Lamsrichan [8] presented an efficient memory embedded algorithm that uses two status bits per pixel only. In this coder, the wavelet coefficients are coded using the wavelet difference reduction (WDR) coding technique in combination with a context-based adaptive arithmetic coder. The algorithm has good performance but at the expense of highly increased complexity since the complexity of the context-based arithmetic coder is about (4-8) times higher than that of the set partitioning technique used by SPIHT or SPECK [15, 16].

In [17] Al-Janabi A.K presented the single list SPIHT (SLS) algorithm. SLS uses two status bits per pixel and a list of size equal to ¼ the image size to store the coordinates of the roots of the spatial orientation trees. The proposed SLS preserves the same information ordering as the original SPIHT. Consequently, both algorithms have nearly the same performance. More importantly, SLS has nearly the same complexity as the original SPIHT. This is due to avoiding the repeated search for all coefficients in every bit-plane coding pass and due to simplifying the memory management as the list is implemented as a simple array that is accessed sequentially instead of the random access linked list that are used by SPIHT.

The remainder of the paper is organized as follows: section 3 presents the proposed algorithms. Section 4 investigates the simulation results of the proposed and some existing algorithms. Finally, section 5 concludes the paper.

3. THE PROPOSED ALGORITHMS

The proposed algorithms use nearly the same approach of SLS [17] to reduce the memory of SPECK. In addition, they employ the step-wise quad tree partitioning coding method adopted by the ultrafast and efficient scalable image compression algorithm presented by Al-Janabi A.K [18] instead of the recursive quad tree partitioning used by SPECK [7] to simplify its memory management.

Like other rate scalable algorithms, the proposed algorithms produce an embedded bitstream for the wavelet image using some form of bit-plane coding. To this end, every wavelet coefficient is initially quantized by rounding it to the nearest integer and represented by a sign-magnitude format using W bits (e.g. 16 bits), where the first bit is the sign bit (e.g. 0 for positive and 1 for negative coefficient) and the other W−1 bits are the magnitude bits. Then the quantized wavelet image (Q*W*) is encoded by multiple coding passes where each pass corresponds to bit-plane level. The maximum bit-plane n_{max} depends on the maximum value of the coefficients in Q*W* and it is given by [6, 7]:

$$\boldsymbol{n_{max} = \left\lfloor log_2 max_{\forall (i,j)\in QW}(|q_{ij}|)\right\rfloor} \quad (1)$$

where $\lfloor x \rfloor$ is the nearest integer $\leq$ x and q_{ij} is a quantized wavelet coefficient at location (i, j) in Q*W*. n_{max} should be sent to the decoder in order to start decoding at bit-plane n_{max}. In the following description, the terms pixel and coefficient are used interchangeably. At any bit-plane, *n*, a pixel q_{ij} is considered significant (SIG) with respect to *n* if:

$$\boldsymbol{2^n \leq |q_{ij}| < 2^{n+1}} \quad (2)$$

otherwise q_{ij} is insignificant (ISIG). Similarly, a set of pixels *T* is considered SIG with respect to *n* if *T* contains at least one SIG pixel; otherwise *T* is ISIG.

3.1 Codec1

Compared to SPECK, the proposed Codec1 has widely lower memory requirements and lower computational complexity. More importantly, these features are attained without affecting the embedding performance of the original algorithm as will be demonstrated experimentally. Memory is greatly reduced by removing the lists LIP and LSP which store the individual pixels. Complexity is reduced due to avoiding the repeated search of all coefficients in each coding pass as done in [8, 9, 12, 13] and due to simplifying the memory management. This is achieved by replacing the array of the linked lists LIS_z by two lists only: the List of Pixels Roots (LPR) and LIS. LPR stores the SIG and ISIG sets that have (2×2) pixels and LIS stores the ISIG sets that have at least (4×4) pixels. As shown shortly in the proposed coding method, once a set is *added* to LPR, the set will be *never* removed. Therefore LPR can be implemented as a simple 1-D array that is accessed sequentially in first in first out (FIFO) manner which is the fastest and simplest access method [11]. Each entry *r* in LPR –in addition to the set's (i, j) coordinates– is provided by a single bit marker termed δ_r to distinguish the significance

state of the set. That is, for each entry *r*, its δ_r is initialized as ISIG and updated to SIG once its corresponding set T_{ij}^r becomes SIG, where T_{ij}^r denotes a set at entry *r* whose root is at location (i, j) in the image.

Codec1 removes LIP and LSP by exploiting the fact that the pixels that are stored in these lists belong to the SIG sets of size (2×2) pixels. So, instead of storing these pixels in these lists, they can be easily deduced from these sets. That is, each (2×2) set at location (i, j) in LPR can act as a root for the four pixels at location (i, j), (i+1, j), (i, j+1), and (i+1, j+1) in the image. Notice that LIP stores the pixels that are tested ISIG in the previous coding passes while LSP stores the pixels found SIG in the previous coding passes and the pixels that are just become SIG in the current pass. So in order to distinguish between these three types of pixels, each pixel q_{ij} in *QW* is provided by a two status bits marker termed σ_{ij} to identify the pixel's type as follows:

- σ_{ij} = 0: q_{ij} is tested ISIG pixel in the previous passes and will be tested again in the current pass.
- σ_{ij} = 1: q_{ij} is New SIG (NSIG) pixel that just becomes SIG in the current bit-plane pass and will not be refined in the current coding pass.
- σ_{ij} = 2: q_{ij} is Previous SIG (PSIG) pixel that is found SIG in the previous passes and will be refined in the current pass.

As mentioned before, each coefficient in *QW* can be represented by W bits with typical value of at least 16 bits (2 bytes) [6]. It is worth noting that the maximum value of the coefficients in *QW* ($|q_{max}|$) is equal to 2^L times the maximum pixel value in the original image, where L is the number of wavelet decomposition levels [19, 20]. Therefore, for an image with pixel bit depth of *b* bits, $|q_{max}|$ is given by:

$$|\boldsymbol{q_{max}}| = \mathbf{2^L \times (2^b - 1) = 2^{(L+b)} - 2^L \leq 2^{(L+b)}} \qquad (3)$$

Thus, L+*b*+1 bits are sufficient to represent each coefficient, where the additional bit is for the sign bit. Typical value for *b* is 8 bits and for L is 5, so each coefficient can be represented by 5+8+1 = 14 bits. Therefore, there are 16-14 = 2 unused bits per coefficient. These 2 bits can be used as the status marker bits (σ) [19, 21], i.e. there is no need to use an auxiliary status marker bits.

An important difference between SPECK and the proposed Codec1 lies on the method of sets partitioning. SPECK starts with sets of variable sizes and uses recursive quad-tree partitioning. As such, there isn't any particular order of the sets stored in LIS leading to use the array of linked lists LIS_z that needs complex memory management [7]. In contrast, Codec1 makes use the implicit step-wise quad-tree partitioning adopted in [18] in order to eliminate the need of LIS_z as follows: the algorithm starts with sets of equal size and when a set T_{ij} in LIS is found to be SIG, one step of quad-tree partitioning is performed on T_{ij} by which it is partitioned from middle along each side into four subsets collectively denoted as $O(T)$. The size of each one of these subsets is one half (in each of the x and y dimensions) the size of the parent set T_{ij}. Then these four subsets are added to the end of LIS to be tested later on at the same bit-plane coding pass. The adopted method provides nearly the same embedding performance as the recursive quad tree partitioning used by SPECK without the need to employ the array of linked lists LIS_z leading to simplify the memory management of the algorithm and hence reducing its complexity [17]. However, since the sets stored in LIS have different sizes, each entry in these lists must have an additional field of *s* bits to store the size of the corresponding set. As shown in the next section, for an image of size (512×512) pixels, *s* consumes at most 3 bits per entry.

Another memory management simplification of Codec1 is that in SPECK, each list of LIS_z should be implemented as a linked list as it is subject to add and remove operations. However, in the proposed coding method, LIS is divided to two lists termed the LISO and LISE. LISO is processed in the *Odd* parity coding passes and reinitialized (i.e., it is set as empty) at the beginning of the *Even* parity coding passes while LISE is processed in the *Even* parity coding passes and reinitialized at the beginning of the *Odd* parity coding passes. As shown next, the two lists will be subject to add operation only and thus can be implemented as 1-D arrays that are accessed in FIFO manner leading to simplify the algorithm's memory management and hence reducing its complexity further. Another advantage of list re-initialization at the beginning of each bit-plane is to avoid the inter-dependency on the previous bit-planes. This will improve the algorithm's error resilience since the encoding pass is independent of the previous passes, so if an error occurs in any pass, it will affect the decoding of that pass only [21]. It is worth noting that the size of both LISO and LISE doesn't exceed the size of LIS because they have the same number of sets but rather than storing them in LIS, they are distributed between LISO and LISE.

The coding stage is initialized by computing the maximum bit-plane n_{max}; outputting n_{max} to the bitstream, and setting *n* to n_{max}; and setting LPR as empty list. Then the image is divided into Q sets of equal sizes, where Q is a power of two integer ≥ 4. For instance, for an image size (512×512) pixels and Q = 4, there are four sets of size (256×256) pixels each. The (i, j) coordinates of the roots of these equal size sets are stored in LISO. Each bit-plane coding pass *n* has three sub-passes. The first two sub-passes test the pixels and sets that are still ISIG, while the third sub-pass codes the pixels that are found SIG in the previous bit-plane passes.

The first sub-pass starts by processing the sets of size (2×2) pixels in LPR. Each entry *r* in the LPR is processed according to its significance status δ_r. If δ_r is ISIG, this means that its corresponding set T_{ij}^r is still marked ISIG, so it is passed to the $Code(T_{ij}^r, n)$ procedure to be tested and encoded. $Code(T_{ij}^r, n)$ first tests T_{ij}^r for significance with respect to *n*. If T_{ij}^r is still ISIG, then a '0' is sent to the bitstream and the process is terminated. Else if T_{ij}^r is found to be SIG, (i.e., one or more of its four pixels just become SIG in the current pass), then a '1' is sent; its δ_r is marked as SIG (i.e. T_{ij}^r becomes SIG); finally, each one of its four children which is at location (x, y), x = i, i+1, and y = j, j+1 is passed to the *Code_pixel*(q_{xy},*n*) procedure for encoding. *Code_pixel*(q_{xy},*n*) tests each q_{xy} for significance with respect to *n*. If q_{xy} is still ISIG, then a '0' is sent to the bitstream. Else if q_{xy} just becomes SIG in the current pass, then a '1' and its sign bit are sent to the bitstream and it is marked as NSIG (σ_{xy} = 1). On the other hand, if for entry *r* δ_r is marked SIG, this means that its T_{ij}^r is marked SIG at any one of the previous passes. In this case T_{ij}^r may contain one or more pixels that are found SIG in the previous passes but they are still marked as NSIG pixel and may contain one or more ISIG pixels. Thus each NSIG pixel is marked as a PSIG pixel to be refined later on at the third sub-pass of current coding pass while each ISIG pixel is coded by the *Code_pixel*(q_{xy},*n*). Updating NSIG pixels to PSIG at this stage is necessary in order to distinguish them

from the pixels that will become NSIG later on in the *current* coding pass during the *QTree* procedure described shortly.

The second sub-pass processes the sets in LISO or LISE according to the pass parity (*P*). If *P* is *odd*, LISE is *reinitialized* and each set T_{ij} in LISO is passed to the procedure *QTree*(T_{ij},*z*,*n*,*P*) for encoding, where $z > 2$ is the set size. *QTree*(T_{ij},*z*,*n*,*P*) starts by testing T_{ij} with respect to *n*. If T_{ij} is still ISIG, '0' is sent to the bitstream, and if *P* is *odd*, (i.e., T_{ij} belongs to LISO), T_{ij} is *not kept* in LISO; rather, it is *added* to LISE to be tested in the next even pass. On the other hand, if T_{ij} is SIG, '1' is sent to the bitstream, then it is quad-tree partitioned into the four children subsets $O(T_{ij})$ each of size *z*/2. Then *z* is updated to *z*/2 and if $z > 2$, these subsets are added to the end of LISO to be passed to the *QTree* procedure later on at the same sub-pass; otherwise, if $z = 2$, each subset $O(T_{ij})$ is added to LPR and passed to $Code(T_{ij}^r, n)$ procedure described before. It is worth noting that if the pass parity *P* is *even*, the same process is performed but the lists are interchanged.

The third sub-pass deals with the sets of (2×2) pixels in LPR that are marked SIG in the previous passes (i.e., except those that are marked as SIG in the current pass). For each one of these set, only PSIG pixels (with $\sigma = 2$) are refined. A PSIG pixel is refined to more bit precision by sending its n_{th} MSB to the bitstream. Finally, the bit-plane *n* is decremented by one to start a new coding pass. This process continues until all the pixels are encoded or until the target bit-rate is achieved. The following pseudo code describes the main coding steps of Codec1 algorithm.

Step1: ***Initialization***

- Compute $n_{max} = \lfloor log_2 max_{\forall (i,) \in QW}(|q_{ij}|) \rfloor$.
- Output n_{max} and Set $n = n_{max}$.
- LPR = ϕ, LISO = $\{T_m\}$, $1 \le m \le Q$.

Step 2: Bit-plane Coding Passes

2.1 ***The first sub-pass***

∀ Entry $r \in$ LPR ***do***:

- ***if*** $\delta_r = 0$ ***then*** $Code(T_{ij}^r, n)$; // T_{ij} is still ISIG
- ***else do***: // T_{ij} *is* SIG

  ∀ $(x, y) \in T_{ij}$: $x = i, i+1$, $y = j, j+1$, ***do*:**
  - ***if*** $\sigma_{xy} = 1$ ***then*** $\sigma_{xy} = 2$;
  - **else** *Code_pixel*(q_{xy}, *n*); // $\sigma_{xy} = 0$

2.2 ***The second sub-pass***

- ***if*** *P* is Odd ***do***: // *pass is odd*
  - LISE = ϕ;
  - ∀ entry $r \in$ LISO ***do***: $QTree(T_{ij}, z, n, P)$;
- ***else do***: // *pass is even*
  - LISO = ϕ;
  - ∀ entry $r \in$ LISE ***do***: $QTree(T_{ij}, z, n, P)$;

2.3 The third sub-pass

∀ entry $r \in$ LPR & δ_r is SIG in previous passes ***do***:

∀$(x, y) \in T_{ij}$ & $\sigma_{xy} = 2$ ***then*** *output* the n_{th} MSB of q_{xy}.

2.4 ***if*** $n > 0$, ***then*** $n = n - 1$ and go to 2.1;
else end coding;

Procedure $Code(T_{ij}^r, n)$ {

- ***if*** T_{ij} is ISIG ***then*** output(0);
- ***else do***: // T_{ij} is SIG
  - output(1);
  - $\delta_r = 1$
  - ∀ $(x, y) \in T_{ij}$ ***do***: *Code_pixel*(q_{xy}, *n*); }

Procedure *Code_pixel*(q_{xy}, *n*) {

- ***if*** q_{xy} is ISIG ***then*** output (0);
- ***else do***: // q_{xy} is SIG
  - output(1);
  - output sign(q_{ij});
  - $\sigma_{xy} = 1$; }

Procedure $QTree(T_{ij}, z, n, P)$ { // $z \ge 4$

- ***if*** T_{ij} is SIG ***do***:
  - output (1);
  - quad-tree partition T_{ij} into four subsets
  - $z = z / 2$;
  - ***if*** $z > 2$ ***do***:

    ∀ $O(T_{ij}) \in T_{ij}$ ***do***:
    - ***if*** *P* is Odd ***then*** add each $O(T_{ij})$ to the end of LISO;
    - Else add each $O(T_{ij})$ to the end of LISE;
  - ***else if*** $z = 2$ do:

    ∀ $O(T_{ij}) \in T_{ij}$ ***do***:
    - Add $O(T_{ij})$ to LPR;
    - Pass $O(T_{ij})$ to $Code(T_{ij}^r, n)$;
- ***else do***: // T_{ij} is ISIG
  - output (0);
  - ***if*** *P* is Odd ***then*** Add T_{ij} to the end of LISE;
  - ***else*** Add T_{ij} to the end of LISO; }

Like other set partitioning algorithms, the decoding process mirrors the encoding process. That is, the decoder duplicates the encoder's execution path as follows: replace the word 'output' by the word 'input' for all procedures, then at any execution point, when the decoder receives '1', this means that the corresponding set or pixel is SIG; otherwise it is ISIG. More precisely, if the corresponding set is SIG, it is partitioned, and if it is ISIG, it is simply bypassed. On the other hand, if the corresponding pixel is SIG, the decoder knows that it lies in the interval $[2^n, 2^{n+1})$, so it is reconstructed at the center of the interval which is equal to $\pm 1.5 \times 2^n$ depending on its sign bit. For example, assume that $n = 5$, so at the encoder a pixel with magnitude = +35 is SIG because $2^5 \le +35 < 2^6$. At the decoder it is reconstructed to $1.5 \times 2^5 = +48$. Every refinement (REF) bit updates the value of the pixel found SIG in previous bit-planes by adding or subtracting a value of (0.5×2^n) depending on its sign. More specifically, if the pixel is positive, a value of (0.5×2^n) is added to it if the REF bit is '1', and a value of (0.5×2^n) is subtracted from it if the REF bit is '0'. On the other hand, if the pixel is negative, a value of (0.5×2^n) is subtracted from it if the REF bit is '1' and (0.5×2^n) is added if the REF bit is '0'. For example, the pixel +35 which is reconstructed to +48 at $n = 5$, is updated to $48 - (0.5 \times 2^4) = 40$ at $n = 4$ as its REF bit is '0' at this bit-plane. In this way the values of reconstructed pixels approach the originals ones and hence the distortion decreases as more bits are decoded.

3.2 Codec2

The proposed codec2 algorithm has lower complexity than the Codec1. This is achieved by merging the first and third sub-passes. The adopted coding method doesn't differentiate between the NSIG and the PSIG pixels. In other words, there are only two types of pixels: ISIG and SIG. Therefore a single status bit per pixel is sufficient: $\sigma_{ij} = 0$ if q_{ij} is ISIG and $\sigma_{ij} = 1$ if q_{ij} is SIG. Codec2 works exactly as Codec1 except that LPR is processed in one sub-pass only as follows: a set T_{ij}^r in LPR that is yet marked ISIG ($\delta_r = 0$), it is passed to the $Code(T_{ij}^r, n)$ procedure for encoding as before. On the other

hand, if the set T_{ij}^{r} is marked SIG (δ_r = 1) at any one of the previous passes, this means that it may contain one or more pixels that are found SIG in the previous passes and may contains ISIG pixels. Thus each ISIG pixel (with σ = 0) is coded by the *Code_pixel(q_{xy},n)* procedure as given before while each SIG pixel (with σ = 1) is refined directly by sending its n_{th} MSB to the bitstream. The pseudo code of Codec2 is identical to that of Codec1 except that the first sub-pass (step 2.1) is modified as given below and the third sub-pass is eliminated because it is merged with the first sub-pass.

2.1 ***The first sub-pass***

$\forall$ entry $r \in$ LPR ***do***:

- ***if*** $\delta_r = 0$ ***then*** $Code(T_{ij}^{r}, n)$; // T_{ij} is still ISIG
- ***else do***: // T_{ij} *is* SIG
  - $\forall (x, y) \in T_{ij} : x = i, i+1, y = j, j+1$, ***do*:**
    - ***if*** $\sigma_{xy} = 0$ ***then*** *Code_pixel(q_{xy},n)*; // q_{xy} is still ISIG.
    - ***else*** *output* the n_{th} MSB of q_{xy}. //q_{xy} is SIG to be refined

Notice that with Codec2, LPR is scanned and processed once per bit-plane coding pass, while in Codec1 it is scanned and processed twice per pass. Therefore, it has lower complexity than Codec1. The price to be paid is a very slight decrement in the embedding performance. This is mainly due to merging the first sub-pass that is dedicated to test and code the ISIG pixels with the third sub-pass that is dedicated to refine the PSIG pixels. This will lead to deteriorate the embedding ordering slightly [2]. However, the performance decrement is very small as will be demonstrated experimentally in the next section.

4. SIMULATION RESULTS

In this section, the proposed Codec1 and Codec2 algorithms are evaluated and tested using the standard 8 bpp gray-scale (512×512) pixels 'Lena', 'Barbara', 'Mandrill', and 'Goldhill' test images. Each image is transformed using the bi-orthogonal 9/7 Daubechies 2D-DWT with L = 5 dyadic decomposition levels and the wavelet coefficients are rounded to the nearest integer prior to coding. The initial set size for Codec1 and Codec2 was (128×128) pixels (Q = 8). The algorithms were implemented by MATLAB 7.10 under Window 7, Intel Core 2 Duo CPU with 2.3 GHz speed and 2GB of RAM. The algorithms were compared with SPECK [7], and LEBP [13] algorithms. The test includes the algorithm's rate distortion performance, its memory requirements, and its computational complexity.

4.1 Rate Distortion Performance

The rate distortion performance is represented by the Peak Signal to Noise Ratio (PSNR) vs. bit-rate (the average number of bits per pixel (bpp) for the compressed image). The PSNR is given by [2, 22]:

$$\boldsymbol{PSNR = 10\, log_{10} \frac{255^2}{MSE}\ dB} \qquad (4)$$

where MSE is Mean-Squared Error between the original image I_o and the reconstructed image I_r each of size M×N pixels defined as:

$$\boldsymbol{MSE = \frac{1}{M \times N} \sum_{i=1}^{M} \sum_{j=1}^{N} [I_o(i,j) - I_r(i,j)]^2} \qquad (5)$$

Table 1 illustrates the algorithms' PSNR vs. bit-rate for the four test images. For each bit-rate, the best PSNR is bolded. No arithmetic coding was used in the significance test or any symbols produced by the algorithms for these results, i.e. the output bits were written directly to the bitstream. Using contexts-based modeling arithmetic coding generally improves PSNR by about 0.5 dB but at the same time increases the algorithm's complexity [6].

Table 1. PSNR vs. bpp for test images

	Bit-rate (bpp)	SPECK	LEBP	Codec1	Codec2
Lena	0.0156	23.25	**23.71**	23.25	23.35
	0.0313	25.38	**25.67**	25.35	25.24
	0.0625	27.73	**27.90**	27.72	27.75
	0.125	30.36	30.56	**30.65**	30.44
	0.25	33.56	33.54	**33.62**	33.42
	0.5	**36.79**	36.62	36.77	36.57
	1	**39.98**	39.75	39.97	39.72
	2	44.37	-	**44.39**	43.77
Barbara	0.0156	21.38	**21.68**	21.45	21.48
	0.0313	22.51	**22.63**	22.49	22.50
	0.0625	23.68	23.64	**23.67**	23.57
	0.125	24.70	25.05	**25.16**	25.14
	0.25	27.48	27.43	**28.02**	27.86
	0.5	31.26	31.48	**31.97**	31.51
	1	36.18	36.72	**37.05**	36.51
	2	42.22	-	43.12	**43.21**
Goldhill	0.0156	**23.59**	-	23.57	23.46
	0.0313	24.92	-	24.90	**24.97**
	0.0625	**26.39**	-	26.37	26.35
	0.125	28	-	**28.07**	27.90
	0.25	30.13	-	**30.15**	29.91
	0.5	**32.71**	-	32.59	32.40
	1	**36.01**	-	35.88	35.69
	2	**41.13**	-	41.02	40.64
Mandrill	0.0156	19.42	**19.52**	19.40	19.38
	0.0313	19.85	**19.94**	19.84	19.84
	0.0625	20.54	**20.95**	20.53	20.51
	0.125	**21.45**	21.40	21.44	21.43
	0.25	22.87	**22.90**	22.87	22.90
	0.5	25.08	24.77	**25.08**	25.07
	1	**28.63**	28.30	28.60	28.27
	2	**34.11**	-	34.07	33.40

The table depicts the following:

- For Lena, the proposed Codec1 algorithm and the original SPECK have close PSNR at all bit-rates. SPECK outperforms Codec1 by at most 0.03 dB at 0.0313 bpp. On the other hand, Codec1 outperforms SPECK by at most 0.29 dB at 0.125 bpp. LEBP has slightly higher PSNR than Codec1 at very low bit-rates between 0.0156-0.0625 bpp. On the other hand, Codec1 perform better at bit-rates between 0.125-1 bpp, which is the most useful bit-rate range.
- For Barbara, which has high frequency content, Codec1 has higher PSNR than SPECK at most bit-rates with maximum PSNR difference of 0.9 dB at 2 bpp. Again, LEBP has slightly higher PSNR at bit-rates between 0.0156 and 0.0313 bpp only.
- For Goldhill, Codec1 and SPECK have very close PSNR.
- For Mandrill, which is the most complex image, all algorithms have also very close PSNR.
- Finally, the proposed Codec2 is also competitive with SPECK and Codec1 for all images.

These results demonstrate that the proposed Codec1 and Codec2 have competitive PSNR efficiency with the other state-of-art algorithms. The advantages of these algorithms are the reduced memory and complexity as shown in the next subsections.

4.2 Memory Requirements

The algorithm's memory Requirements is represented by the memory size required by its auxiliary lists. Notice that SPECK uses LSP, LIP and the array LIS_z, z = 2, 4 …Z. Most authors assume that since each individual pixel can be marked as either significant of insignificant, so the total number of entries of LIP and LSP is equal to the number of image pixels [9, 12, 13]. As mentioned before, the list size cannot be preallocated because it depends on the image size, image type and on the compression bit-rate. Therefore, this assumption can be realized only if the slow dynamic memory allocation technique is employed leading to increase the complexity greatly [10]. The more practical solution is to pre-allocate the size of each list to the worst case which occurs when the compression bit-rate is at full bit-rate. Table 2 depicts the actual size of all LIS_z, z = 2, 4…Z, LIP, and LSP used by SPECK and the actual size of LPR, LISE, and LISO used by the proposed Codec1 for the four test images compressed at full bit-rate. The size is with respect to the number of image pixels (M×N).

Table 2. The memory size of the auxiliary lists of SPECK and the proposed Codec1 at full compression bit-rate

Image	SPECK			Codec1		
	LIS_z	LIP	LSP	LPR	LISE	LISO
Lena	0.22	0.37	0.73	0.25	0.15	0.08
Barbara	0.16	0.30	0.74	0.25	0.10	0.06
Goldhill	0.18	0.33	0.81	0.24	0.11	0.08
Mandrill	0.16	0.30	0.91	0.25	0.08	0.10
Max.	0.22	0.37	0.91	0.25	0.15	0.10

As shown, the size of the lists varies with the aforementioned parameters. So, for algorithm reliability, the maximum size values must be adopted. Table 2 also depicts that the maximum size of LIP and LSP is about 1.3 the image size. In addition, the maximum size of LIS_z is 0.22 the image size and the maximum size of both LISE and LISO is 0.25 the image size. This means that the cost of implementing linked list LIS with random access as two 1-D arrays with FIFO access is very negligible which is 0.03 of memory increment with respect to image size.

Each entry of these lists must have *c* bits to store the i and j coordinates of the corresponding pixel. For an image with (M×N) pixels, $c = \lceil \log_2(M) \rceil + \lceil \log_2(N) \rceil$, where $\lceil x \rceil$ is the nearest integer $\geq x$ [12, 21]. Furthermore, for LIP and LIS which are implemented as linked lists, each entry must be provided by a pointer to point out to the next entry [8, 11]. Thus, the maximum memory requirement of SPECK is equal to:

$$\boldsymbol{MEM_{SPECK}^{max} = \{c \times 0.91MN + [(c + p_{LIP}) \times 0.37MN] + [(c + p_{LIS}) \times 0.22MN]\} / 8 \quad Bytes} \qquad (6)$$

where the three terms correspond to the maximum memory of LSP, LIP, and LIS_z respectively, and p_{LIP} and p_{LIS} are the pointer size for LIP and LIS respectively. Lamsrichan [8] assume that the size of the pointer (*p*) is 4 bytes = 32 bits per entry. However, a more precise calculation for *p* is that since every list entry may be the next entry, so, *p* is equal to $\lceil \log_2(\text{max. list size}) \rceil$ bits per entry. For LIP, $p_{LIP} = \lceil \log_2(0.37MN) \rceil$, and for LIS, $p_{LIS} = \lceil \log_2(0.22MN) \rceil$. For instance, For an image with (512×512) pixels, $c = \lceil \log_2(512^2) \rceil = 18$ bits, $p_{LIP} = \lceil \log_2(0.37 \times 512^2) \rceil = 17$ bits, and $p_{LIS} = \lceil \log_2(0.22 \times 512^2) \rceil = 16$ bits. Thus the maximum memory requirement of SPECK is equal to:

$$MEM_{SPECK}^{max} = \{18 \times 0.91 \times 512^2 + (18 + 17) \times 0.37 \times 512^2 + (18 + 16) \times 0.22 \times 512^2\}/8 \cong 1218\, KB$$

For Codec1 and Codec2, LPR, LISO and LISE are implemented as simple 1-D arrays. So, they don't need to use the pointer *p*. However, each one of LPR entries must have *c*+1 bit, where the additional bit is for set's significance marker (δ_r). In addition, each entry in LISO and LISE must have *c*+s bits, where the additional *s* bits are used to store the size *z* of its corresponding set. It is worth noting that *s* depends on the size of the initial Q sets (*Z*×*Z*). However, as *Z* is a power of two, the set can be represented as $(2^t \times 2^t)$ pixels, where $Z = 2^t$. So, $t = \lceil \log_2(Z) \rceil$ and $s = \lceil \log_2(t) \rceil$. Therefore, the maximum memory requirement of Codec1 and Codec2 is equal to:

$$\boldsymbol{MEM_{Codec1,2}^{max} = \{(c + 1) \times 0.25MN + (c + s) \times 0.25MN\} / 8\, Bytes} \qquad (7)$$

where the two terms correspond to the maximum memory requirements of LPR, and LISO + LISE respectively. For an image with (512×512) pixels and Q = 4, i.e. the initial set size is (256×256) = $(2^8 \times 2^8)$ pixels. So $s = \log_2(8) = 3$ bits, and c = 18 bits. Thus the maximum memory requirement of Codec1 is equal to:

$$MEM_{Cod1}^{max} = \{19 \times 0.25MN + 21 \times 0.25MN\}/8 \cong 328\, KB$$

The LEBP algorithm uses a list of length (M×N) to store the magnitudes of all wavelet coefficients. Thus each list entry must have W bits. In addition, each coefficient is provided with a 4 bits status marker. Thus the total memory required for LEBP is given by [13]:

$$MEM_{LEBP} = MN \times W + MN/2 + (2L + 2)\ Bytes \quad (8)$$

For a (512×512) pixels image, and L = 5, the total memory required for LEBP is equal to $512^2 \times 2 + 512^2/2 + 12 \cong 655\ KBytes$.

As it can be seen, the memory requirements of the proposed algorithms are about 27% and about 50% of SPECK and LEBP respectively. The size of an (512×512) pixels gray-scale image with 1 byte per pixel is equal to $512^2 \cong 262$ Kbytes. Thus SPECK and LEBP need additional memory of about 4.5 and 2.5 times the image size respectively. On the other hand, the proposed algorithms need additional memory of 1.25 times the image size. More importantly, memory reduction is accompanied with the reduction of the algorithm's complexity as shown in the next subsection.

4.3 Computational Complexity

The algorithm's computational complexity is represented by the execution time needed to encode/decode the image against bit-rate. The proposed algorithms and SPECK were evaluated using MATLAB 7.10.0 under Window 7, Intel Core 2 Duo CPU with 2.3 GHz speed and 2 GB of RAM. Only the coding and decoding times were reported because the wavelet transform consumes the same time for all algorithms. Tables 3 reports the coding and decoding times (measured in seconds) vs. the bit-rate for Lena image. For LEBP, the coding and decoding times were taken from [13] knowing that it was implemented in MATLAB 7.10.0 under Window XP, Intel Core 2 Duo CPU with 3GHz speed and 4GB of RAM which is nearly the same computer environment used by the proposed algorithms.

Table 3. The coding and decoding time (in seconds) vs. bit-rate of the algorithms for Lena image

	Bit-Rate (bpp)	SPECK	LEBP	Codec1	Codec2
Coding time (seconds)	0.0156	0.065	0.11	0.056	0.032
	0.0313	0.075	0.14	0.068	0.066
	0.0625	0.135	0.24	0.077	0.075
	0.125	0.260	0.42	0.145	0.145
	0.25	0.485	0.79	0.164	0.165
	0.5	0.843	1.47	0.310	0.294
	1	1.330	2.87	0.527	0.500
Decoding time (seconds)	0.0156	0.025	0.03	0.003	0.003
	0.0313	0.040	0.05	0.008	0.008
	0.0625	0.010	0.11	0.020	0.017
	0.125	0.184	0.23	0.037	0.034
	0.25	0.387	0.45	0.081	0.071
	0.5	0.622	0.90	0.122	0.105
	1	1.133	1.78	0.252	0.219

As clearly shown, the coding time of Codec1 is about two times faster than that of SPECK especially at bit-rates (0.0625-1) bpp. This is achieved due to the many memory simplifications adopted by the proposed coding method. In contrast, LEBP runs by about two times slower than SPECK. This means that Codec1 is about four times faster than LEBP. Finally, Codec2 is the fastest among all algorithms because of the adopted sub-pass merging which led to scanning LPR one time per bit-plane instead of two. However, as shown in subsection 4.1, Codec2 has slightly lower PSNR than Codec1 due to not preserving the information ordering according to the embedding principle. On the other hand, for all algorithms, the decoding time is shorter than the coding time because that the decoder doesn't need to scan and test the sets' pixels at every bit-plane to see if the set becomes SIG. However, the decoding time of Codec1 is also about two times faster than that of SPECK while LEBP runs by about 1.1-1.5 times slower than that of SPECK. Finally, Codec2 is also the fastest among all algorithms. However, the speed improvement is less than that of the coding time as there is no set testing at the decoder side.

5. CONCLUSION

In this paper, we presented two wavelet-based rate scalable algorithms. As shown from the experimental performance results, the proposed algorithms have competitive PSNR, have about 75% lower memory and run by about two times faster as compared to SPECK. In addition, they have lower complexity, lower memory and better PSNR as compared with other low memory algorithms (e.g., LSK and LEBP). These factors make the proposed algorithms very attractive because with current technology, memory is not a limiting factor for most applications. The low memory and complexity of the proposed algorithms makes them very suitable for multispectral, hyper-spectral, 3D image compression, and real-time applications such as video transmission, where compression speed is vital. Furthermore, a fast algorithm requires short execution time and hence it consumes less power. Consequently the proposed algorithms can be very useful for wireless sensor networks (WSNs), mobile phones, digital cameras, etc. which have limited resources in terms of power, processing speed, and memory. Another advantage of the algorithms is their asymmetric property as its decoding time is much faster than its encoding time. This property is very valuable with scalable image compression as the image is compressed only once and may be decompressed many times. Finally, the proposed algorithms have weak list memory dependency due to using simple 1-D arrays with sequential FIFO access [3]. This interesting feature permits to upgrade them easily to be highly scalable algorithms that produce a compressed bitstream which is both rate and resolution scalable [23]. That is, an image at several resolutions and bit-rates can be obtained from the compressed bitstream by a simple scaling process. This flexibility makes the highly scalable bitstream very useful for modern heterogeneous users that have diverse bit-rate and resolutions requirements interconnected via heterogeneous wired and wireless networks that have diverse bandwidths.

6. REFERENCES

[1] Goyal V.K, "*Theoretical Foundations of Transform Coding"*, IEEE Signal Processing Magazine, Vol. 18, No.9, pp. 9-21, Sep.2001.

[2] Rabbani, M. and Joshi, R., "*An Overview of the JPEG 2000 Still Image Compression Standard",* Signal Processing: Image Communication, Vol. 17, No. 1, pp. 3-48, Jan. 2002.

[3] Taubman D., Weinberger M., Seroussi G., Ueno I., and Ono F., "*Embedded Block Coding in JPEG2000*," Signal Processing: Image Communication Vol. 17, No. 1, pp. 49-72, Jan. 2002.

[4] Ordentlich E., Weinberger M. and Seroussi G., "*A Low-Complexity Modeling Approach for Embedded Coding of Wavelet Coefficients"*, Proc. IEEE Data Compression Conference, DCC 98, Snowbird, pp. 408-417, March 1998.

[5] Li J. and Lei S., "*An Embedded Still Image Coder with Rate-Distortion Optimization"*, IEEE Trans. on Image Processing, Vol. 8, No. 7, pp. 913-924, July 1999.

[6] Said, A. and Pearlman, W.A., "*A New, Fast, and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees"*, IEEE Trans. on Circuits & Systems for Video Technology, Vol. 6, No. 3, pp. 243-250, 1996.

[7] Pearlman, W.A, Islam, A., Nagaraj, N. and Said, A., "*Efficient, Low Complexity Image Coding with a Set-Partitioning Embedded Block Coder"*, IEEE Trans. on Circuits &Systems for Video Technology, Vol. 14, No. 11, pp. 1219-1235, Nov. 2004.

[8] Lamsrichan P., "*A fast algorithm for low-memory embedded wavelet-based image coding without list"*, the 8th Int. Conf. on Electrical Engineering/Electronics, Computer, Telecom. & Information Technology (ECTI-CON), pp. 979-982, Khon Kaen, May 2011.

[9] Jianjum, W. and Bo Liu, "*Modified SPIHT Based Image Compression for Hardware Implementation"*, IEEE Computer Society, Second Int. Workshop on Computer Science and Engineering, Qingdao, Vol. 2, pp. 572-576, Oct. 2009.

[10] Chrysafis C., Said A., Drukarev A., Islam A., & Pearlman, W.A, "*SBHP-A Low Complexity Wavelet Coder*," IEEE Int. Conf. Acoustic., Speech and Sig. Proc. (ICASSP2000), Istanbul, Vol. 4, pp. 2035-2038, June 2000.

[11] Berman, A.M., "*Data Structures via C++: Objects by Evolution"*, 1st edition, Oxford University Press, New York, USA, 1997.

[12] Latte M., Ayachit N. and Deshpande D., "*Reduced memory listless SPECK image compression"*, Elsevier Science, *Digital Signal Process.* Vol. 16, No. 6, pp. 817–824, Nov. 2006.

[13] Senapati R. and Mankar P., "*Improved Listless Embedded Block Partitioning Algorithms for Image Compression"*, International Journal of Image and Graphics, Vol. 14, No. 4, pp. 1–32, Dec. 2014.

[14] Hsiang S. and Woods J., "*Embedded Image Coding Using Zeroblocks of Subband/Wavelet Coefficients and Context Modeling"*, the 2000 IEEE Int. Symp. on Circuits and Systems (ISCAS2000), Geneva, Vol. 3, pp. 662–665, May 2000.

[15] Zhang Y.Z, Chao X., Wen T. W., and Liang B. C., "*Performance Analysis and Architecture Design for Parallel EBCOT Encoder of JPEG2000*," IEEE Trans. on Circuits & Systems for Video Technology, Vol. 17, No. 10, pp. 1336-1347, Oct. 2007.

[16] Pearlman, W.A., "*Trends of Tree-Based, Set-Partitioning Compression Techniques in Still and Moving Image Systems"*, Proceedings Picture Coding Symposium (PCS-2001), Seoul, Korea, 25-27, pp. 1-8, April, 2001.

[17] Al-Janabi A.K, "*Low Memory Set-Partitioning in Hierarchical Trees Image Compression Algorithm*," International Journal of Video & Image Processing and Network Security IJVIPNS-IJENS, Vol. 13, No. 2, pp. 12-18, April 2013.

[18] Al-Janabi A.K, "*Ultrafast and Efficient Scalable Image Compression Algorithm"*, Journal of ICT Res. Appl., to be published.

[19] Sakalli M., Pearlman W.A, and Farshchian M., "*SPIHT algorithms using Depth First Search Algorithm with minimum memory usage"*, IEEE 40th Annual Conference on Information Sciences and Systems, pp. 1158-1163, Princeton, NJ, 22-24 March 2006.

[20] Wern L., Minn A. and Seng K, "*Reduced Memory SPIHT Coding using Wavelet Transform with Post-Processing*", IEEE Inter. Conf. on Intelligent Human-Machine Systems and Cybernetics, IHMSC '09, pp. 371-374, Hangzhou, Zhejiang, Aug. 2009.

[21] Arora H., Singh P., Khan E., and Ghani F., "*Memory Efficient Set Partitioning in Hierarchical Tree (MESH) for Wavelet Image Compression",* ICASSP 2005, pp. 385-388, Mar. 2005.

[22] Salomon, D., "*Data Compression: the Complete Reference"*, 3rd ed., Springer, New York, USA, 2004.

[23] Al-Janabi, A.K., "*Highly Scalable Single List Set Partitioning in Hierarchical Trees Image Compression"*, IOSR Journal of Electronics and Communication Engineering, Vol. 9, No. 1, pp. 36-47, 2014. DOI: 10.9790/2834-09133647.