SPIHT IMAGE COMPRESSION WITHOUT LISTS

Frederick W. Wheeler and William A. Pearlman

Rensselaer Polytechnic Institute
Electrical, Computer and Systems Engineering Dept.
Troy, NY 12180, USA
wheeler@cipr.rpi.edu, pearlman@ecse.rpi.edu

ABSTRACT

A variant of the SPIHT image compression algorithm called No List SPIHT (NLS) is presented. NLS operates without linked lists and is suitable for a fast, simple hardware implementation. NLS has a fixed predetermined memory requirement about 50% larger than that needed for the image alone. Instead of lists, a state table with four bits per coefficient keeps track of the set partitions and what information has been encoded. NLS sparsely marks selected descendant nodes of insignificant trees in the state table in such a way that large groups of predictably insignificant pixels are easily identified and skipped during coding passes. The image data is stored in a one dimensional recursive zig-zag array for computational efficiency and algorithmic simplicity. Performance of the algorithm on standard test images is nearly the same as SPIHT.

1. INTRODUCTION

The EZW [1] and SPIHT algorithms [2] are fast and effective techniques for image compression. Both are spatial tree-based and exploit magnitude correlation across bands of the decomposition. Each generates a fidelity progressive bitstream by encoding, in turn, each bitplane of a quantized dyadic subband decomposition. Both use significance tests on sets of coefficients combined with set partitioning to efficiently isolate and encode high magnitude coefficients.

An important difference between EZW and SPIHT is in their set partitioning rules. SPIHT has an additional partitioning step in which a descendant (type A) set is split into four individual child coefficient sets and a granddescendant (type B) set.

EZW explicitly performs a breadth first search of the hierarchical trees, moving from the coarse to fine bands. Though it is not explicit, SPIHT does a roughly breadth first search as well. After partitioning a granddescendant set, SPIHT places the four new descendant sets at the *end* of the LIS, or list of insignificant sets. It is the appending to the LIS that results in the approximate breadth first traversal. Breadth first, as opposed to depth first, scanning improves performance because coefficients more likely to be significant are tested first.

This work was supported in part by the National Science Foundation under grant numbers NCR-9523767 and EEC-9812706, and by a fellowship from Sun Microsystems.

It is more complex to implement a zerotree codec that does a breadth first search for significant coefficients. The codec needs to determine whether each node of the tree encountered during the search must be tested and coded or can be skipped because it is a member of an insignificant set. It would be inefficient to repeatedly look up the tree for an ancestor that is the root of a zerotree. Other means have been developed.

SPIHT uses list structures to keep track of which sets must be tested. However, the use of lists in SPIHT causes a variable, data dependent, memory requirement, and the need for memory management as list nodes are added, removed and moved. At high rates, there can be as many list nodes as coefficients. This may be undesirable in hardware implementations.

We have developed a new image coder called No List SPIHT (NLS) that uses the set partitioning rules of SPIHT, and does an explicit breadth first search without using lists. State information is kept in a fixed size array that corresponds to the array of coefficient values, with about four bits per coefficient to enable fast scanning of the bitplanes.

In NLS, instead of searching up the tree to find *predictable insignificance*, special markers are placed in the state table on certain lower nodes of insignificant trees when the trees are created. These sparse markers are updated when new insignificant trees are formed by partitioning. With this sparse marking scheme, large sections of the image are skipped at once as the breadth first scan moves through the lower nodes of the spatial trees.

Lin *et al.* have developed listless zerotree codecs for images [3] and video [4] that also make use of fixed size state tables. However, in some passes, their codecs perform a depth first search of the trees.

In NLS, efficient skipping of blocks of insignificant coefficients is accomplished using a recursive zig-zag image indexing scheme. Instead of indexing the array of coefficients using two indices we move the two dimensional array to a one dimensional array. This particular format offers several computational and organizational advantages. Seetharaman *et al.* [5] have used this same linear coefficient ordering in image segmentation applications.

2. LINEAR INDEXING

The linear indexing technique uses a single number to represent the index of a coefficient instead of two. Let $R = C =$

	0	1	2	3	4	5	6	7
0	0	1	4	5	16	17	20	21
1	2	3	6	7	18	19	22	23
2	8	9	12	13	24	25	28	29
3	10	11	14	15	26	27	30	31
4	32	33	36	37	48	49	52	53
5	34	35	38	39	50	51	54	55
6	40	41	44	45	56	57	60	61
7	42	43	46	47	58	59	62	63

Figure 1: Linear indexing with $R = 8$, $C = 8$, and two subband levels with bands delimited by thick lines.

2^N be the number of rows and columns of the square image, and let r and c be zero-based row and column indices. Represent the row index in binary $r = [r_{L_r-1},\ldots,r_1,r_0]$, where each of the r_n is a bit, and do the same for the column index. For an index (r, c) the linear index is defined by $i = [r_{L_r-1}, c_{L_r-1}, \ldots, r_1, c_1, r_0, c_0]$. The bits of r and c are simply interleaved. The linear index i ranges from 0 to $I-1$, where $I = RC$. Fig. 1 shows an example of this indexing scheme. Notice that the children on a spatial tree have 4 consecutive indices, the grandchildren have 16 consecutive indices, and so on. We will take advantage of this property when skipping past lower nodes of insignificant trees.

Another important property of the linear index is that it efficiently supports the operations on coefficient positions needed for tree-based algorithms with one operation instead of two, assuming the usual subband data arrangement of Fig. 1. Also, the linear index naturally facilitates a breadth first search of the hierarchical trees.

Given either coordinates (r, c) or i, suppose we must find the index of the first child (indicated with subscript c) in the spatial tree. Using row and column indices, we need $r_c = 2r$, and $c_c = 2c$. For the linear index the single operation is $i_c = 4i$. To find the location of the parent coefficient (indicated with subscript p) using row and column indices, we need $r_p = \lfloor r/2 \rfloor$, and $c_p = \lfloor c/2 \rfloor$. With the linear index, the single required operation is $i_p = \lfloor i/4 \rfloor$. Iterating over four siblings in a tree is another common operation. With row and column indices, the iteration can be represented by $r_n = r, r+1$ and $c_n = c, c+1$. Using the linear index, only a single level of iteration is needed, using $i_n = i, i+1, i+2, i+3$. All multiplication here (and throughout) is by integral powers of 2 and can be implemented by bit shifting.

Image data is usually stored in raster order, so index conversion must be done for each coefficient once by the encoder and once by the decoder. The interleaving and deinterleaving required to convert between index modes is trivial in custom hardware, but not directly supported by general purpose CPUs. For an efficient software conversion, we use a fixed 256 entry lookup table that maps one byte to two bytes, with zero bits padded between the original bits. This bit spreader table combined with shift and bitwise OR operations makes the conversion simple and fast [5].

3. NO LIST SPIHT

NLS uses the same set structures and partitioning rules as SPIHT. The trees are tested for significance breadth first. Significance tests are made in a different order than SPIHT because SPIHT performs significance tests roughly breadth first, while NLS performs the tests strictly breadth first. Because the set splitting rules are the same, each coder produces the exact same output bits, though in a different order. For the coefficients in the 8 by 8 example image given in [1], NLS and SPIHT happen to produce identical bitstreams.

There are three passes per bitplane. First, the insignificant pixel (IP) pass which corresponds to SPIHT's LIP pass tests each lone insignificant pixel for significance. The significant set (IS) pass, like SPIHT's LIS pass, tests each multiple-pixel set for significance, splitting and repeating as needed. Finally, the refinement (REF) pass, like SPIHT's LSP pass, refines pixels found significant in previous bitplanes.

3.1. Storage

The number of coefficients in the DC band is $I_{dc} = R_{dc}C_{dc}$, where $R_{dc} = R\,2^{-L}$, $C_{dc} = C\,2^{-L}$, and L is the number of subband decomposition levels. The coefficients are stored in a single array of length I. For convenience, we will refer to the magnitude part with the array `val` and the sign part with array `sign`. The state table is an array, named `mark`, of length I, with 4 bits per coefficient. There is a one-to-one correspondence between `val` and `mark`.

Zerotree encoders can optionally trade memory for computation by precomputing and storing the maximum magnitude of all possible descendant and granddescendant sets [6]. For NLS, the precomputed maximum descendant magnitude array, `dmax`, has length $I/4$, and the maximum granddescendant magnitude array, `gmax`, has length $I/16$. These arrays can be eliminated at the expense of repeated searching over insignificant trees for significant coefficients.

If each subband coefficient is stored in W bytes, the total bulk storage memory needed is: RCW for the subband data, $RCW/4$ for the maximum descendant array, $RCW/16$ for the maximum granddescendant array, and $RC/2$ (half byte per pixel) for the state table. For a 512 by 512 image using 2 bytes per coefficient, and using the optional precomputed maximum descendant arrays, this is 800k bytes, or 56% more than is needed for the image alone. This is all of the significant memory needed for this algorithm. The amount of memory required is fixed given the size of the image. We are not counting the memory required for the subband transform, but this part of the system can be handled efficiently by a rolling line-based transform.

3.2. State Table Markers

The following markers are placed in the 4 bit per coefficient state table, `mark`, to keep track of the set partitions. Each element of `mark`, if set, indicates something about the corresponding element in the `val` image array. Each marker and its meaning is listed below.

M*P Each of these first four markers are for a lone pixel.

- MIP The pixel is insignificant or untested for this bitplane.
- MNP The pixel is newly significant so it will not be refined for this bitplane.
- MSP The pixel is significant and will be refined in this bitplane.

MCP Like MIP, but applied during partitioning in the IS pass so the new pixel set will be tested for significance immediately during the same IS pass, while MIP pixels are skipped.

MD The pixel is the first (lowest index) child in a set consisting of all descendants of its parent.

MG The pixel is the first (lowest index) grandchild in a set consisting of all granddescendants of its grandparent pixel, but not including the grandparent or the children of the grandparent.

MN* The following markers are used on the leading node of each lower level of an insignificant tree. As the image is scanned, these markers indicate that the next block of pixels is insignificant.

- MN2 The pixel is the first grandchild of a MD set. This pixel and its 16 (4 by 4) 1st cousins can be skipped.
- MN3 The pixel is the first great grandchild of a MD or MG set. This pixel and its 64 (8 by 8) 2nd cousins can be skipped.
- ...
- MN6 The pixel is the first 6th generation descendant of a MD or MG set. This pixel and its 4096 (64 by 64) 5th cousins can be skipped.

The markers MD and MG are similar in meaning to the SPIHT set types A and B. These set markers are associated directly with a pixel that is in the set, while in SPIHT, types A and B are associated with the root pixel of a set, which is not actually in the set.

The edge markers, MN*, are used to keep track of which pixels are members of insignificant sets with roots at higher levels of the tree. With linear indexing, when a MN2 marker is reached during a scan we simply advance the scan index by 16. When a MN3 marker is reached we advance 64, etc.

3.3. Initialization

A dyadic subband transform is performed on the image with about $L = 5$ levels, and the floating point transform coefficients are quantized to integers. Next, the image is read into the linear array `val`. For each (r, c), find i by bit interleaving and move the coefficient. The mean of the DC band is removed and transmitted.

If encoding, the two maximum descendant magnitude arrays are computed. The maximum magnitude of the descendant set rooted at coefficient i is `dmax`$[i]$, and the maximum magnitude of the granddescendant set rooted at coefficient i is `gmax`$[i]$. This data is computed in advance by scanning the first quarter of the linear array backwards and using the equations `gmax`$[i] = \max($`dmax`$[4i]$, `dmax`$[4i+1]$, `dmax`$[4i+2]$, `dmax`$[4i+3])$ and `dmax`$[i] = \max($`val`$[4i]$, `val`$[4i+1]$, `val`$[4i+2]$, `val`$[4i+3]$, `gmax`$[i])$. Zero is substituted for `gmax`$[i]$ when $i \geq I/16$ because these nodes have no granddescendants. These arrays are actually computed via bitwise OR instead of the max function.

The most significant non-zero bitplane, B, is found by scanning the DC band and a small section of `dmax`, and transmitted.

The function `push`, in pseudo-code below, is used to insert MN* markers where needed to the lowest tree level after a new descendant set is created.

```
define push(i)
  mark[4i] = MN2; mark[16i] = MN3; ···
```

A 5 level descendant tree has 1364 pixels, but only 4 need to be marked by the `push` function. When scanning the image, only the top level MD marker and 4 MN* markers associated with the tree will be encountered. So, a great number of predictably insignificant coefficients are skipped with little processing.

The state table is initialized by marking each DC coefficient with MIP, and each full-size spatial tree with MD. For each initial tree and each decomposition level, the leading pixel is marked with an appropriate MN*, as shown in the initialization pseudo-code below. Very few of the coefficients are marked during initialization.

```
for i = 0, ..., I_dc − 1
  mark[i] = MIP
for i = I_dc, ..., 4I_dc
  mark[i] = MD; push(i)
```

A small constant lookup table, `skip`, tells how many coefficients to skip if a marker is encountered and no processing needs to be done for that section in this pass. Values of `skip` are `skip`[M*P] = 1, `skip`[MD] = 4, `skip`[MG] = 16, `skip`[MN2] = 16, `skip`[MN3] = 64, `skip`[MN4] = 256, `skip`[MN5] = 1024, etc. Another table, `isskip`, is used in the IS pass. It is the same as `skip`, except for `skip`[MIP] = `skip`[MNP] = `skip`[MSP] = 4. Because of the partitioning rules, if a pixel with one of these markers is encountered during the IS pass, each of its siblings can be skipped.

3.4. Main Algorithm

The main encoder algorithm in pseudo-code below is performed for each bitplane, b, starting with B and decrementing to 0, or until a bit budget is met. The significance level for each bitplane is $s = 2^b$. Significance checks are always done with bitwise AND.

The decoder follows the same overall procedure as the encoder with some low-level changes. To decode, use `input` instead of `output`, and set the bits and signs of coefficients with bitwise OR instead of testing them with bitwise AND. The decoder performs midtread dequantization for coefficients that are not fully decoded.

In each pass, the coefficient array `val`, and the precomputed maximum descendant magnitude arrays are examined in the linear array order. The `push` function necessarily sets elements of `mark` ahead of the index.

Though the pseudo-code below for the main algorithm and above for the `push` function use multiplication and division for clarity, all multiplication and division operations are by integral powers of 2 and are implemented by bit shifting. Further, the addition operations needed during set partitioning are done with bitwise OR. Actual addition is only needed when advancing the main index i.

INSIGNIFICANT PIXEL PASS

```
i = 0, while i < I                          start the IP pass
  if mark[i] = MIP                          insignificant pixel
    output(d = (val[i] AND s))              send bit
    if d
      output(sign[i])                       send the sign
      mark[i] = MNP                         pixel now newly significant
  i = i + 1                                 move to next pixel
```

```
else
  i = i + skip[mark[i]]                          move past set/block
INSIGNIFICANT SET PASS
i = 0, while i < I                                start the IS pass
  if mark[i] = MD                              a set of descendants
    output(d = (dmax[⌊i/4⌋] AND s))
    if d                                    if the set is significant
      mark[i] = mark[i + 1] = MCP           split into 4 children
      mark[i+2] = mark[i+3] = MCP   (will test these next)
      mark[4i] = MG                 and the granddescendants
      no increment here so new MCP pixels processed next
    else
      i = i + 4              move past the siblings in this set
  elseif mark[i] = MG              a set of granddescendants
    output(d = (gmax[⌊i/16⌋] AND s))
    if d                                    if the set is significant
      mark[i] = mark[i + 4] = MD                 split into 4 sets
      mark[i+8] = mark[i+12] = MD  (will test these next)
      mark the borders of these new sets
      push(i), push(i + 4), push(i + 8), push(i + 12)
      no increment here so new MD sets processed next
    else
      i = i + 16             move past the cousins in this set
  elseif mark[i] = MCP                   an insignificant pixel
    output(d = (val[i] AND s))
    if d                                  if the pixel is significant
      output(sign[i])               send the sign of the pixel
      mark[i] = MNP         the pixel is now newly significant
    else
      mark[i] = MIP                   the pixel is insignificant
    i = i + 1                               move past this pixel
  else
    i = i + isskip[mark[i]]                    move past set/block
REFINEMENT PASS
i = 0, while i < I                        start the refinement pass
  if mark[i] = MSP                                significant pixel
    output(val[i] AND s)                           refine the pixel
    i = i + 1                                  move past the pixel
  elseif mark[i] = MNP                      newly significant pixel
    mark[i] = MSP                  significant pixel in next plane
    i = i + 1                                  move past the pixel
  else
    i = i + skip[mark[i]]                      move past set/block
```

4. RESULTS AND CONCLUSIONS

Coding results for the 512 by 512 grayscale lena, goldhill and barbara images are plotted in Fig. 2. For these experiments, we used a five level decomposition with the 9/7 biorthogonal wavelet [7]. The solid lines show rate vs. distortion performance for SPIHT without arithmetic coding. The dash-dot line shows the performance of NLS without arithmetic coding. All decoded images for each curve were recovered from a single fidelity embedded encoded file, truncated at the desired rate.

NLS and SPIHT produce bitstreams with the same bits in different order. SPIHT's ordering offers slightly better performance at certain points in the bitstream during the set significance (IS/LIS) passes. During refinement the results are practically identical. At the end of each bitplane, the results are perfectly identical.

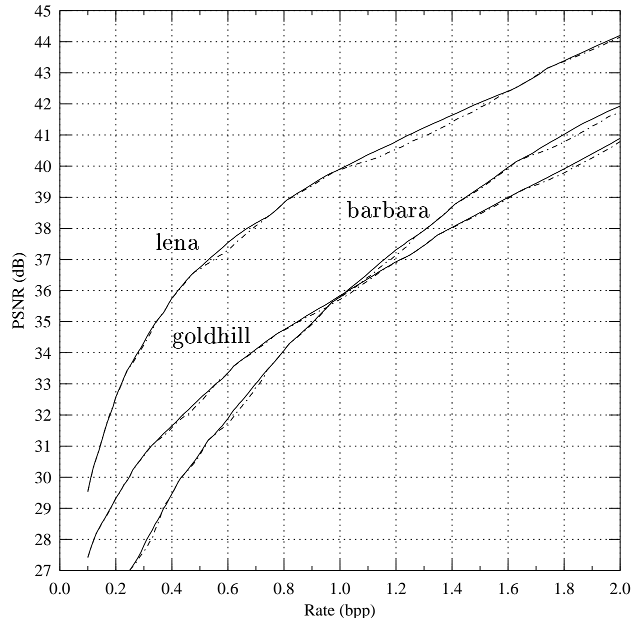

Figure 2: Coding performance for binary uncoded SPIHT and NLS on several images.

No arithmetic coding was used on the significance test or any symbols produced by the SPIHT or NLS algorithms for these results. Back-end arithmetic coding using contexts and joint encoding generally improves SPIHT by about 0.5 dB. We expect that improvement for NLS as well.

5. REFERENCES

[1] J. M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," *IEEE Trans. on Signal Processing*, vol. 41, pp. 3445–3462, December 1993.

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

[3] W.-K. Lin and N. Burgess, "Listless zerotree coding for color images," in *Proc. of the 32nd Asilomar Conf. on Signals, Systems and Computers*, vol. 1, pp. 231–235, November 1998.

[4] W.-K. Lin and N. Burgess, "3D listless zerotree coding for low bit rate video," in *Proc. of the International Conf. on Image Processing*, October 1999.

[5] G. Seetharaman and B. Zavidovique, "Z-trees: Adaptive pyramid algorithms for segmentation," in *Proc. of the International Conf. on Image Processing*, 1998.

[6] J. M. Shapiro, "A fast technique for identifying zerotrees in the EZW algorithm," in *Proc. of the International Conf. on Acoustics, Speech and Signal Processing*, pp. 1455–1458, May 1996.

[7] 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.