Interpolating Leaf Quad Tree Image Compression

Author

Lincoln, Luke, Gonzalez, Ruben

Published

2013

Conference Title

2013 7TH INTERNATIONAL CONFERENCE ON SIGNAL PROCESSING AND COMMUNICATION SYSTEMS (ICSPCS)

Interpolating Leaf Quad Tree Image Compression

Luke Lincoln, Member, IEEE
Griffith University
Gold-Coast, Queensland (07) 5552 8100
Email: luke.lincoln@griffithuni.edu.au

Ruben Gonzalez, Member, IEEE
NICTA and Griffith University
Gold-Coast, Queensland (07) 5552 8100
Email: r.gonzalez@griffith.edu.au

***Abstract*—In the past, wavelet and transform based methods have dominated the field of image compression, however they bring computational complexity to the compression process and are typically more suited to high bit rate compression. The Quad Tree (QT) is a simple image compression method but is not as effective as transform based codecs, such as JPEG. This paper presents a novel variation of the QT called the Interpolating Leaf Quad Tree (ILQT). The ILQT uses QT decomposition, combined with bi-linear interpolation to better approximate image blocks. The ILQT is shown to outperform JPEG and other QT variations in Rate-Distortion (RD) tests at low bit rates, it also outperforms existing methods in terms of perceptual quality.**

I. Introduction

The QT is an effective image compression structure, it works by breaking up an image into a hierarchical data structure (a tree) where each leaf in the tree represents a section of the original image. Many variations on the QT have been proposed ([1], [2], [3]), these methods have been shown to have good Peak Signal to Noise Ratio (PSNR) vs. Bits Per Pixel (BPP) performance but often produce such results by sacrificing algorithm complexity. The ILQT method presented in this paper competes with these algorithms in terms of low bit rate RD performance whilst reducing algorithm complexity and improving perceptual quality over the generic QT.

This paper provides a background into some of the previous work performed in hierarchical image compression (section II), from there, it presents objectives for the ILQT, as well as which algorithms are suitable for comparison and why. Then in section III, the basic QT concept is explained for readers not familiar with this area, terms in this section are used throughout the paper. Next, in section IV, the ILQT is presented, and details of its: decomposition, bit representation and compression strategy are discussed. Section V contains various experiments showing the effectiveness of the ILQT method and section VI concludes the research paper.

II. Previous Methods

Many variations of the generic QT structure exist. These methods improve the QT by: optimizing decomposition, improving quadrant representation, joining quadrants, pruning the tree and tree structure compression. The following research bears certain similarities with the work presented here.

A. Shusterman et al.

Shusterman *et al.* [3] pointed out that the generic QT algorithm has poor RD performance compared to notable DCT based methods. They also make a note about the blocking effect produced by the QT. To alleviate these problems they have improved the choice of threshold chosen and altered the method of decomposition, they have also introduced a method for bit allocation in order to improve the effectiveness of quantization.

B. Quad-Binary Tree

Kassim *et al.* [1] introduce a hybrid codec called the Quad Binary (QB) tree, which is a cross between three hierarchical techniques: the QT, the binary space partitioning tree (BSP) and the wedgelet (WL). As QT decomposition is performed, the QB tree must search for the best leaf node representation out of the following methods: a single block of colour (QT), a block split by a plane containing two colours (WL), or a polygon split by a plane, also containing two colours (BSP). It is shown to outperform JPEG2000 and performs well at low bit rates, however, it increases algorithm complexity as it requires a large amount of computation per block [1].

C. ShadeTree

Gonzalez [4] came up with a technique called the Shadetree (ST), which also modifies the traditional QT structure. This tree representation stores the corner values of quadrants and performs interpolation between these corner values. Corner values can be shared by up to four tree leaves, this gives the ST better performance at higher bit rates. For decompression, the ST requires a search of the tree to find corner values for interpolation. The ST method is somewhat suitable for comparison with the ILQT method, however the ILQT is targeted at low bit rate compression and is less computationally complex than the ST as it does not require a search for corner values.

D. Varma et al.

Varma *et al.* [2] have introduced a novel variation of the QT, and as with the ST they have altered the way in which leaf nodes represent image blocks. They have focused on designing an image codec which performs well at low bit rates, thus this method has similar goals to the ILQT. Varma *et al.* use first order polynomials to represent quadrants in the QT. This involves calculating variables for use in a system of linear

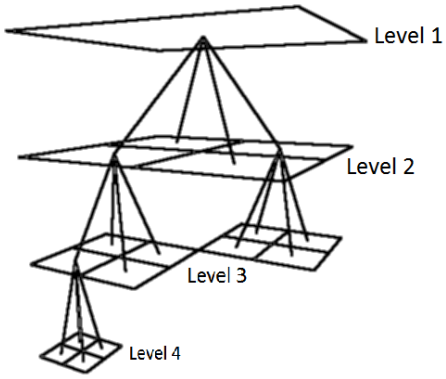

Fig. 1. Quad Tree Structure

equations, which, once solved, produce the coefficients of these polynomials. The coefficients are then quantized and stored in a bottom-up QT structure. This method has been shown to outperform JPEG at low bit rates. As mentioned, this algorithm has the most similar aims to the ILQT and so it is most suitable for comparison with the ILQT.

E. Aims

Through the ILQT, we aim to provide a novel image codec which is effective at low bit rates. Such a method should maintain the advantages associated with the QT, such as simplicity, whilst managing and preventing the disadvantages, such as poor RD performance and the known blocking effect.

III. Quad Tree

The QT is a hierarchical data structure used to decompose an image into a set of square blocks, each block is uniquely defined with a location, size and colour. Within the hierarchy, each node containing no sub-nodes is termed a leaf node, see figure 1 for an example of this structure. The node at level 1 is called the root node, this figure has leaves at levels 2, 3 and 4. Each node can have up to four leaves, called quadrants in reference to the image. QT decomposition can be viewed in figure 2, here, black lines are drawn on the borders of each quadrant for visualization purposes.

Each leaf represents a block of pixels in the image. A single value is used by the leaf to represent the entire block, this value is the average (mean) value of the pixels within the block. The average pixel value within a block is defined by the equation,

$$AVG = \frac{1}{S^2} \sum_{y=b}^{b+S-1} \sum_{x=a}^{a+S-1} f(x,y) \qquad (1)$$

where (a, b) is the location of the lower corner of the leaf node, and S is the width and height of the block represented by the leaf node. Using the average value, decomposition is halted when the error between the average value and the original pixel values is below a threshold, $error < T$, where T is a user defined threshold value. The error can be defined using

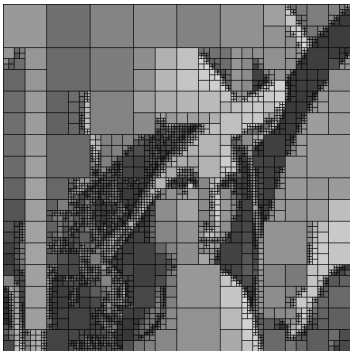

Fig. 2. Decomposition of Image using QT

any error metric, however the Mean Square Error (MSE) is most popular. The MSE is defined using the original image $f(x, y)$ and the leaf value, AVG. Below, the MSE is shown for use in QT decomposition, it uses the same variables as discussed with the mean pixel equation.

$$MSE = \frac{1}{S^2} \sum_{y=b}^{b+S-1} \sum_{x=a}^{a+S-1} (AVG - f(x,y))^2 \qquad (2)$$

An excellent introduction and detailed discussion of the QT can be found in chapter 1.4 of Hanan Samet's book, Foundations of Multidimensional and Metric Data Structures [5].

IV. Interpolating Leaf Quad Tree

A. Decomposition

The ILQT follows the decomposition strategy of a top-down QT, however instead of representing each leaf with the average value of a block, four values are used to interpolate between corners. These four values are chosen to be the average value of the quadrants which are closest to the corner. See the left image in figure 4 for an example of this, here the average values of the quadrants are A, B, C and D. Once these averages are calculated, they are interpolated to generate a shaded block, shown on the right in figure 4. This allows the ILQT to represent blocks more accurately than the generic QT, it also reduces the blocking effect. This interpolation feature is similar to the Shade Tree (ST) [4] which uses exact corner values for interpolation. Using this interpolation gives the ST an advantage at higher bit rates as corner values are shared by up to four leaves. The four colours stored using the ILQT cannot be shared by other leaves, however the ILQT approximates each block better, giving the ILQT an advantage at low bit rate compression. An image of Lena decomposed with the ILQT (using the same MSE threshold as in figure 2) is shown in figure 3.

B. Weighted Threshold

The ILQT exploits some basic assumptions about the tree data structure. The further a quadrant is decomposed,

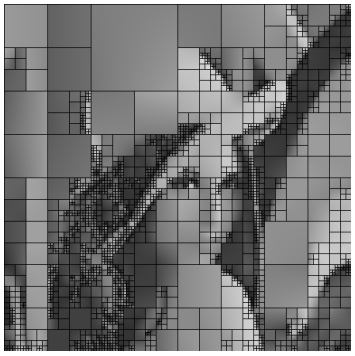

Fig. 3. Decomposition of Image using ILQT

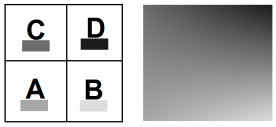

Fig. 4. ILQT Leaf

1) the more memory the structure requires,
2) the higher quality the image will be, and
3) the more memory efficiency is sacrificed for quality.

It is better to trade memory efficiency for quality as the algorithm progresses through decomposition. Besides the MSE threshold, the ILQT makes use of a level of decomposition threshold called the cut-off. Once decomposition reaches the pixel level, each pixel is represented using four times the amount of data compared to the basic QT algorithm. Therefore the cut-off threshold is used to stop decomposition early, without considering the MSE.

To decide when decomposition should halt, ILQT makes use of several pieces of information: MSE – the error between the original image and the ILQT quadrant (T the threshold for the MSE), x – the width of the quadrant and Q – the cut-off threshold. This cut-off threshold, Q, is defined as the width of a quadrant represented by the node. For example, if 4 is chosen to be this cut-off, once a node reaches a level where the quadrant has a width of 4, decomposition is halted. In order to exploit the assumptions explained earlier, two weights are used, these weights give the ILQT some extra leniency at earlier levels of decomposition. ILQT decomposition is halted if one of the following boolean functions becomes true.

$$\begin{gathered} MSE < T \\ x \leq Q \\ (x \leq (2Q) \wedge MSE \leq W_1 T) \\ (x \leq (4Q) \wedge MSE \leq W_2 T) \end{gathered} \tag{3}$$

Values for Q are typically chosen as $4, 8$ or 16. Weights are chosen such that, W_1 is smaller than W_2, this allows ILQT to stop at an earlier stage given the MSE is low enough.

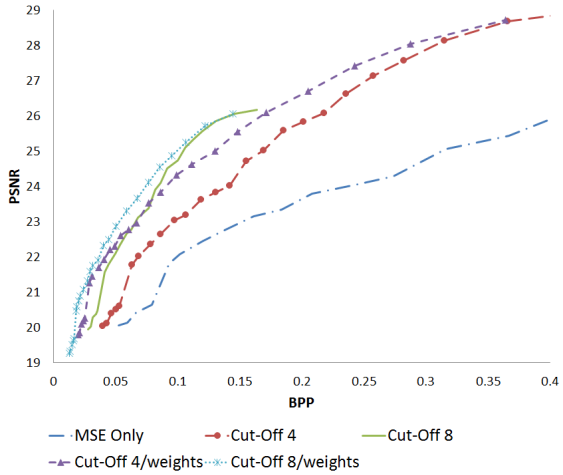

Fig. 5. ILQT Threshold Comparison

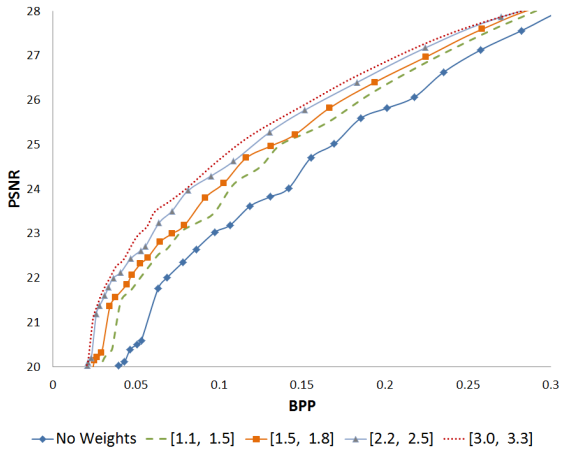

Fig. 6. Comparison of Weights [W_1, W_2]

The results of using of different cut-off thresholds (with and without weights), are shown in figure 5 and results using different weights are shown in figure 6. Each test in figure 5 used weight values, $W_1 = 3.0$ and $W_2 = 3.3$. It can be seen that RD performance suffers if no cut-off is used. With a cut-off, RD performance is gained, however, for a given cut-off, adding weights to the threshold for different decomposition levels further improves the ILQT in terms of RD performance. Results evaluated using different weights in figure 6 show that optimal selection of these weights is around: $W_1 = 3.0$ and $W_2 = 3.3$, values higher than this reduce RD-performance, and these weights perform similarly on a range of images.

C. Bit Representation

1) Tree Data: The ILQT stores the tree in depth-first order, in a top-down fashion. In the tree structure, each node is stored with a single bit, each bit represents the existence of sub-nodes indexed in counter-clockwise order. If a node requires decomposition, a binary 1_2 would be used, conversely a binary 0_2 would indicate the node is a leaf node.

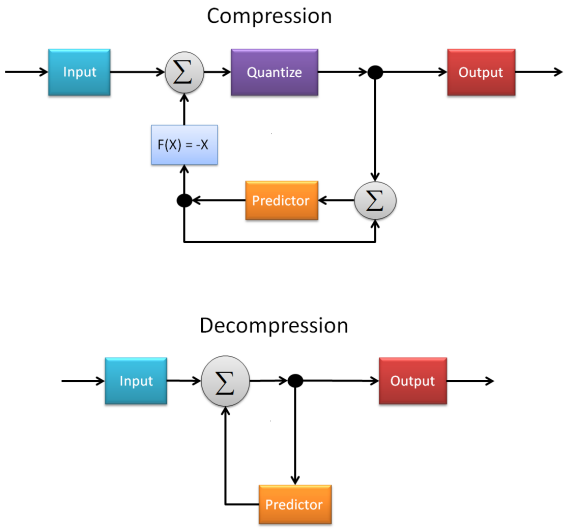

Fig. 7. DPCM Compression Algorithm

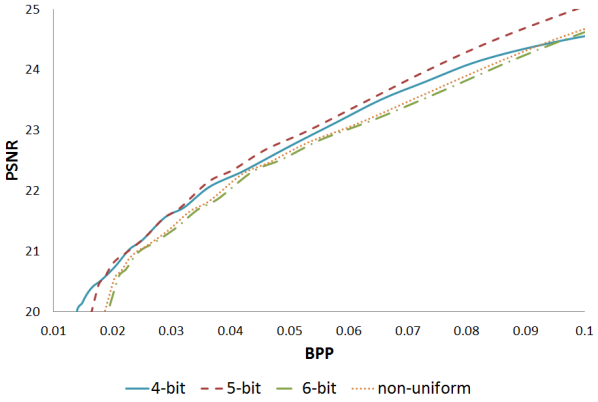

Fig. 8. Effectiveness of DPCM at different levels of Quantization

This bit is regarded as a decision bit by the decoder, it decides if further decomposition is required or if leaf information must be read and decomposition should be halted.

2) Tree Compression: Colour data is stored in a separate data file to the tree structure, each colour is stored in depth-first order as it appears in the tree. The colour data takes up the bulk of the data size, so Differential Pulse Code Modulation (DPCM) is used to decorrelate the data before further compression. Figure 7 shows the DPCM algorithm, the effectiveness of DPCM's usage in ILQT was evaluated and results are shown in figure 8. These results show that quantizing the data to 5 bits produced the best overall RD performance, other quantization levels were found to be more effective than no DPCM compression. The Non-Uniform quantization tested, was based on Weber's law.

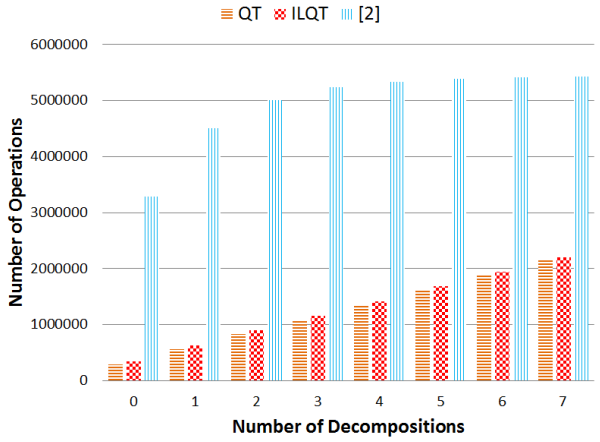

Fig. 9. No. of Operations vs. No. of Decompositions

D. Complexity & Efficiency

The ILQT requires little more operations than the QT, efficiency is defined here as the amount of operations required to calculate a quadrant's inner value. For the QT this is the average value, for the ILQT, it is the average of the four quadrants' values and for Varma *et al.* [2] this is a first order polynomial. These functions depend on the width (w) of the quadrant, for the QT, this requires w^2 additions and 2 multiplications. For the ILQT this requires $4w^2$ additions and 5 multiplications. The case of the first order polynomial was also assessed, assuming that the coefficients are optimally calculated and a non-general systems solver is used, [2] requires, $20w + 36$ additions and $4w + 69$ multiplications. Kassim *et al.* [1] mention that, without speed ups, the QB tree requires assessing $6(w-1)^2$ different partitions, each of which requires around w^2 operations, equating to $6w^4 - 12w^3 + 6w^2$ operations which puts it in another league to the ILQT, QT and [2] in terms of complexity.

The complexity functions are multiplied by the amount of quadrants at a particular level of decomposition, generating a worst case scenario evaluation at different levels of decomposition. Figure 9 shows the efficiency of these algorithms. These graphs are cumulative and assume an image of 512×512 is split up into a maximum of 7 decompositions, where each quadrant at the lowest level is 4×4, this is the limit of [2] and one of the limits used with the ILQT. ILQT uses a top-down approach and so does the implementation of QT, Varma *et al.* have used a bottom up approach which they suggest provides a RD performance advantage, it also means that when compressing at low bit rates it requires much more operations.

In a worst case scenario, the ILQT and QT require a relatively constant amount of addition and multiplication whilst [2] requires a large amount of operations at the first few levels of decomposition, once the 3rd level of decomposition is reached [2] requires less additions than QT and ILQT, but even for total decomposition, the ILQT and QT never reach as many operations as [2].

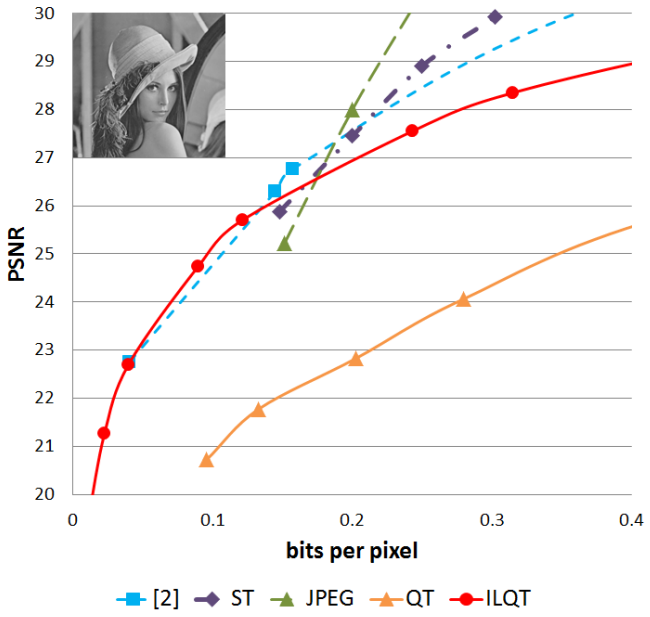

Fig. 10. PSNR vs. BPP with the Lena image

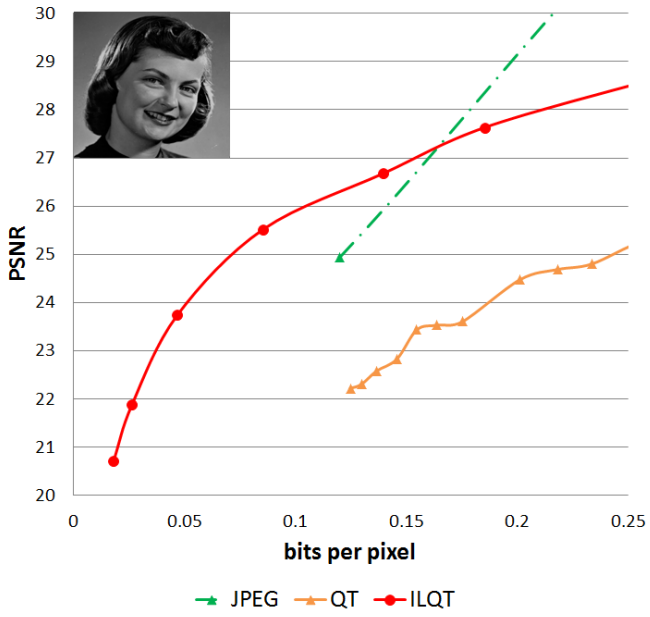

Fig. 11. PSNR vs. BPP with the Young-Girl Image

The ILQT is more efficient than [2], as finding the average value of a quadrant is more simplistic than calculating coefficients and solving a system of equations. Other algorithms such as the QB tree [1], require even more computation than Varma *et al.* and this should be taken into consideration when comparing ILQT with the QB tree.

V. Experiments and Discussion

Experiments are conducted on a 512×512 grey scale version of the Lena image, as well as a 512×512 version of the young girl image, both images are shown inside their RD graphs in figures 10 and 11. Algorithms tested include: a generic version of the QT, the low bit rate QT codec from [2], the ST [4] and transform codec, JPEG. Results have been obtained through corresponding research papers. Experiment results shown include: RD performance, perceptual visual quality and PSNR vs. Compression Ratio tests. PSNR comparisons are calculated as,

$$PSNR = 20\log_{10}\left(\frac{255^2}{\sqrt{MSE}}\right) \tag{4}$$

and compression ratios are taken the same way as in [2],

$$Compression\ Ratio\ =\ \frac{Original\ File\ Size}{Compressed\ File\ Size} \tag{5}$$

In each experiment, weights are chosen to be, $W_1 = 3.0$ and $W_2 = 3.3$], the cut-off is defined as either 2, 4, 8 or 16 and all file sizes exclude the size of the dictionary created by the Huffman coder.

A. Rate-Distortion Performance

The RD performance results from the Lena image are shown in figure 10. These results show that ILQT competes well against [2] at low bit rates, it also bests JPEG at bit rates below 0.2 bpp, ILQT improves upon the generic QT and also the ST [4] with bit rates below the 0.2 bpp mark. It should be noted that ILQT is also more efficient than [2] at low bit rates, it also does not require a search of the tree like ST does during decompression.

RD performance of the 512×512 young girl image shows again that the ILQT is effective at low bit rates, the RD performance is shown in figure 11. The original QT performed poorly on this particular image, however the ILQT outperformed JPEG at bit rates below .15 bpp, and up to a PSNR of $27db$.

B. Compression Ratio Performance

ILQT is shown to outperform the QT method by Varma *et al.* as well as the JPEG method in compression ratios above 70. Figure 12 shows these results, JPEG is easily beaten at these high compression ratios, with ILQT clearly surpassing the results provided by Varma *et al.* This further highlights ILQT's practicability at low bit rate compression.

C. Perceptual Quality Analysis

Figure 13 shows four images compressed with the Varma *et al.* QT and the ILQT, it is easy to see the visual improvement of ILQT over the method presented in [2]. The top row shows that for a much higher compression ratio, the ILQT has much better quality. Compare the hair and lips in these two images, the ILQT has much higher detail in these two areas, and in particular, Lena's face is less distorted in the

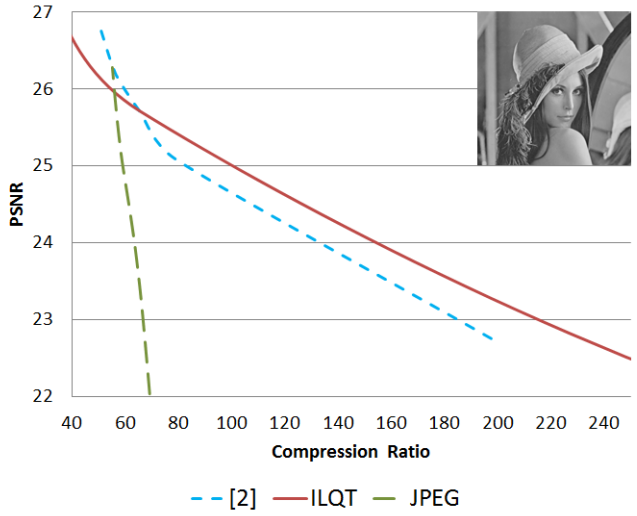

Fig. 12. Varma *et al.* QT vs. ILQT, PSNR vs. Compression Ratio with the Lena image

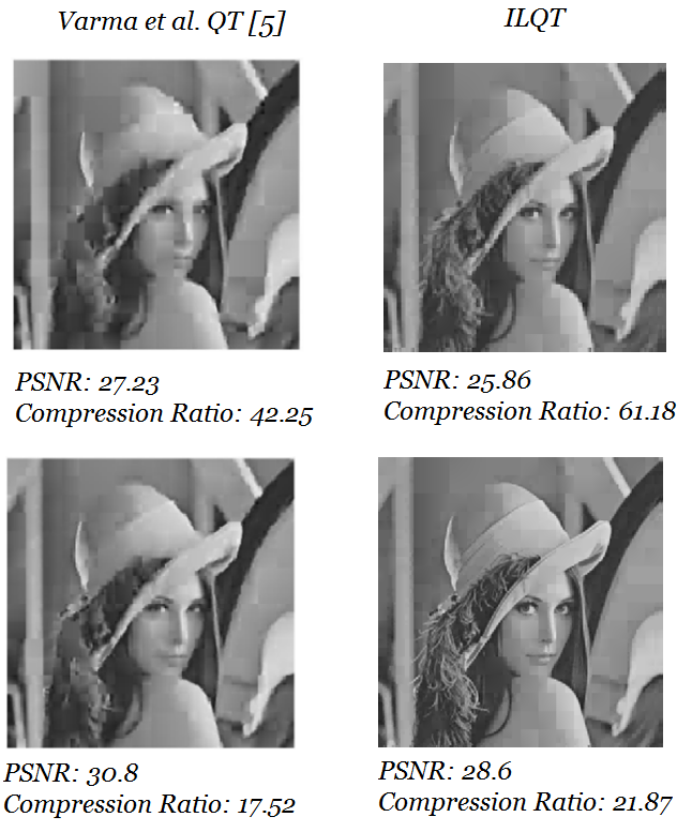

Fig. 13. Varma *et al.* QT vs. ILQT, visual quality comparison, with 512×512 Lena Image

image compressed using the ILQT method. The bottom row further shows that the PSNR metric does not give enough credit to the perceptual quality improvement produced by the ILQT. Here the ILQT has a higher compression ratio like the top row, but notice that the PSNR value is lower for the

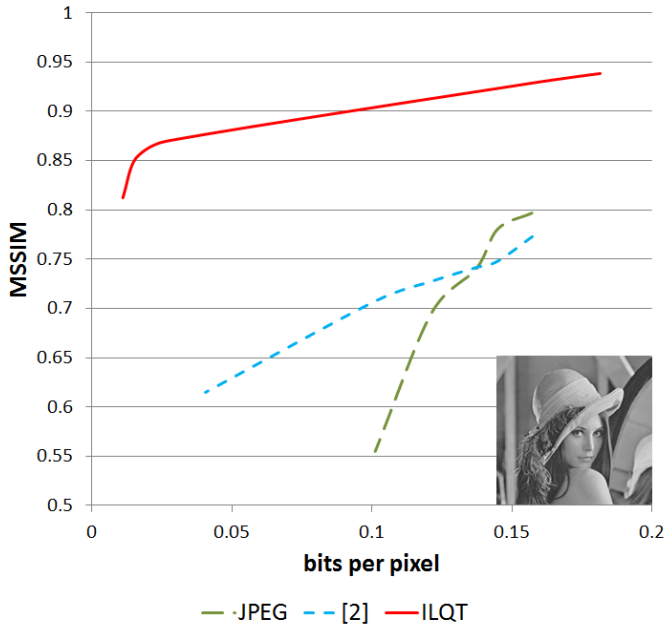

Fig. 14. $MSSIM$ vs. BPP with 512×512 Lena Image

ILQT image, yet the quality is much higher. Notice, also that the ILQT represents edges much more sharply than the QT, which often produces blurred images.

Varma *et al.* also make use of a perceptual metric called the Structural similarity index method ($SSIM$), which is calculated on patches of 8×8, this value is averaged over the image size to calculate what is called the $MSSIM$. The $MSSIM$ metric ranges from -1 (zero similarity to the uncompressed image), to $+1$ (identical to the uncompressed image). Figure 14 shows the rate distortion graph using the $MSSIM$ perceptual quality metric, this graph supports the interpretation of figure 13, that is, the ILQT outperforms Varma *et al.* in terms of perceptual quality. The graph also shows that ILQT outperforms JPEG in terms of perceptual quality by a fair margin.

VI. Conclusion

In this paper, the ILQT, an effective low bit rate image codec was presented. This method is effective at low bit rate image compression because of its ability to accurately approximate leaf colours at higher levels in the tree. Results show that this algorithm improves upon other QT compression methods at low bit rates in terms of efficiency, RD performance and perceptual quality. The ILQT algorithm has similar complexity to QT, making it a simple but effective competitor to other QT variations, it also improves image quality by reducing the QT blocking effect. A pseudo code implementation of the ILQT without weights or a cut-off is presented in Appendix VII-A. Some possible future work could be done in further compressing colour data, as well as exploring other tree

structure variations.

ACKNOWLEDGMENT

The authors would like to thank Dr. Xin-Wen Wu.

REFERENCES

[1] A. Kassim, W. S. Lee, and D. Zonoobi, "Hierarchical segmentation-based image coding using hybrid quad-binary trees," *Image Processing, IEEE Transactions on*, vol. 18, no. 6, pp. 1284–1291, 2009.

[2] K. Varma, S. Parthasarathy, and P. Sankaran, "Application of quad tree for low bitrate compression," in *Computational Intelligence and Signal Processing (CISP), 2012 2nd National Conference on*, 2012, pp. 104–108.

[3] E. Shusterman and M. Feder, "Image compression via improved quadtree decomposition algorithms," *Image Processing, IEEE Transactions on*, vol. 3, no. 2, pp. 207–215, 1994.

[4] R. Gonzalez, "Shadetree image compression for embedded computing," *10th International Symposium on DSP and Communication Systems, DSPCS'2007*, December 2007.

[5] H. Samet, *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

VII. APPENDIX A

A. ILQT Algorithm

```
main function
begin
  im := readImage()                          read in an image
  Node := newNode()                          set-up root node
  Node.pos := (0, 0)
  Node.wid := width(im)
  call buildTree(im, Node, Threshold)call recursive function
end
```

```
average(im, pos, hw)
begin
  avg := 0
  for y := pos.y to pos.y + hw step 1 do
      for x := pos.x to pos.x + hw step 1 do
          avg+ = im.xy
      od
  od
  return avg/hw^2
end
```

```
getColours(im, Node, halfwidth)
begin
  col := array(4)
  col[0] := average(Node.pos, halfwidth)
  Node.pos.x+ = halfwidth
  col[1] := average(Node.pos, halfwidth)
  Node.pos.y+ = halfwidth
  col[3] := average(Node.pos, halfwidth)
  Node.pos.x− = halfwidth
  col[2] := average(Node.pos, halfwidth)
end
```

```
mse(im, Node)
begin
  colIncrement1 := (Node.cols[2] − Node.cols[0])/Node.wid
  colIncrement2 := (Node.cols[3] − Node.cols[1])/Node.wid
  mse := 0
  y1 := Node.cols[0]
  y2 = Node.cols[1]
  for y := Node.pos.y to Node.pos.y + Node.wid step 1 do
      v := y1
      colIncrement3 := (y2 − y1)/Node.wid
      for x := Node.pos.y to Node.pos.y + Node.wid step 1 do
          mse+ = (im.xy − v)^2
          v+ = colIncrement3
      od
      y1+ = colIncrement1
      y2+ = colIncrement2
  od
  Return mse/Node.wid^2
end
```

```
getpos(pos, i, halfwidth)
begin
  if i = 1 → pos.x+ = halfwidth
  □ i = 2 → pos.y+ = halfwidth
  □ i = 3 → pos.x+ = halfwidth, pos.y+ = halfwidth fi
  return pos
end
```

```
buildTree(im, Node, Threshold)
begin
  halfwidth := Node.wid/2
  Node.cols := getColours(im, Node, halfwidth)
  if mse(im, Node) ∨ Node.wid = 8
    then
          WriteNode(Node)
    else
          for i := 0 to 4 step 1 do
              NewNode := ()
              NewNode.pos := getpos(Node.pos, i, halfway)
              NewNode.wid := halfwidth
              call BuildTree(im, NewNode, Threshold)
          od
  fi
end
```