PAPER

A Fast and Memory Efficient SPIHT Image Encoder

Zhong-Ho CHEN†[a], Nonmember and Alvin W. Y. SU†, Member

SUMMARY Set-partitioning in hierarchical trees (SPIHT) is one of the well-known image compression schemes. SPIHT offers an agreeable compression ratio and produces an embedded bit-stream for progressive transmission. However, the major disadvantage of SPIHT is its large memory requirement. In this paper, we propose a memory efficient SPIHT image coder and its parallel implantation. The memory requirement is reduced without sacrificing image quality. All bit-planes are concurrently encoded in order to speed up the entire coding flow. The result shows that the proposed algorithm is roughly 6 times faster than the original SPIHT. For a 512 × 512 image, the memory requirement is reduced from 5.83 Mb to 491 Kb. The proposed algorithm is also realized on FPGA. With pipeline design, the circuit can run at 110 MHz, which can encode a 512 × 512 image in 1.438 ms. Thus, the circuit achieves very high throughput, 182 MPixels/sec, and can be applied to high performance image compression applications.

key words: image coding, wavelet, set-partition in hierarchical trees (SPIHT), FPGA

1. Introduction

Set-Partitioning in hierarchical trees (SPIHT) [1] is one of the well-known image compression schemes. SPIHT offers agreeable compression ratio and produces embedded bit-stream for progressive transmission. In SPIHT image compression, an original image is wavelet transformed and then constructed as a spatial-oriented tree (SOT). Figure 1 shows an example of the SOT. The tree defines the spatial relationship on the hierarchy constructed with recursive four-subband splitting. Each SOT node corresponds to an individual wavelet coefficient and has either no offspring or four offspring, which are always grouped as a 2 × 2 adjacent block. Its offspring correspond to the four coefficients of the same spatial orientation in the next finer level of the pyramid.

A node of SOT is said to be significant if the coefficient magnitude is greater than a certain threshold. Otherwise, it is insignificant. The significance of nodes is important to achieve a better trade-off between rate and distortion, so SPIHT stores the information in three ordered lists, called list of insignificant sets (LIS), list of insignificant pixels (LIP) and list of significant pixels (LSP). The SPIHT algorithm encodes an image bit-plane by bit-plane. The coding flow of a bit-plane is composed of two passes, the sorting pass and the refinement pass. Details of the algorithm can be found in [1].

Although SPIHT can achieve good trade-off between rate and distortion, the price to be paid is its large memory requirement. For a 512 × 512 image, the capacity for LIP/LSP is 512 × 512 × 18 bits and the capacity for LIS is (512 × 512/4) × (16 + 1) bits. Therefore, it requires 5.83 Mb in total to maintain these three lists. Moreover, it is expensive to implement the insertion and deletion operation of the lists by hardware.

In this paper, we propose a memory efficient SPIHT image coder and its parallel implantation. Most importantly, the memory requirement is reduced without sacrificing image quality. All bit-planes are concurrently encoded in order to speed up the entire coding flow. The rest of this paper is organized as follows. Section 2 discusses related works. Section 3 describes the proposed algorithm. Section 4 illustrates the implementation on FPGA. Experiment results are shown in Sect. 5. Finally, the conclusion is given in Sect. 6.

2. Related Work

The memory requirement of SPIHT comes from two sources, buffer for storing transform coefficients and memory for maintaining three lists. First, the wavelet transformed coefficients of entire image should be available in memory. To reduce buffer size for these coefficients, one can apply strip-based wavelet image compression [2]–[4] or image partition [5]. Second, one can apply induction methods [7]–[9] or simplification methods [10]–[20] to reduce the memory requirement for maintaining three lists. In this paper, we focus on reducing the memory capacity for maintaining three lists.

In line-based wavelet image compression [2], image
data is read line by line and only the minimum required number of lines is kept in memory. However, it requires large memory to generate an embedded bit-stream. Strip-based coding [3] adopts the line-based wavelet transform and encapsulates an embedded coder. The transform coefficients are buffered in strip buffer for the following embedded coding. New SPIHT tree structures are proposed in [4] and it adopts the strip-based coding. Unfortunately, the method sacrifices coding efficiency when comparing with the original SPIHT.

An alternative approach to reduce the memory requirement for wavelet coefficients is to apply image partition [5]. However, it involves blocking artifact which has significant impact on the image quality.

The other memory requirement for SPIHT coding is temporary storage of three ordered linked-lists. The lists can be implemented with one memory, such as [6], but the size is still considerable. When investigating the SPIHT algorithm, the LIS only stores nodes with offspring and a node and its offspring aren’t stored in LIS at the same time. By utilizing the property, the size of LIS can be reduced [7], [8]. Because the nodes in the SOT are grouped as 2 × 2 blocks, the size of LIS can be further reduced in [9].

To further reduce the memory requirement, most researches apply simplification with respect to the original SPIHT algorithm. For example, the refinement pass is removed in [10], [11]. The bits are immediately outputted in the sorting pass, thus, LIP and LSP can be thrown away. No List SPIHT [12] adopts a state table and performs the sorting pass, thus, LIP and LSP can be thrown away. Since the lists are removed, the order of pixels is missing and the image quality is decreasing. Some researches adopt fixed-order search to remove the memory for performing breath-first search. Depth-first search approaches [13]–[15] implement a regular SPIHT-like coder but the coding efficiency also decreases. An alternative scan order based upon Morton Order is used in [16]–[20]. Though, they provide less trade-off between rate and image quality, loss in image quality is still a problem.

3. Proposed Methods

The original SPIHT algorithm encodes transform wavelet coefficients bit-plane by bit-plane and requires multiple accesses to a coefficient. These coefficients are usually stored in slow external memory and its access is time-consuming. In this section, we describe a parallel implementation of SPIHT algorithm, all bit-planes are encoded concurrently. A coefficient is only accessed twice. Moreover, we adopt one priority queue rather than three lists to maintain the order of coefficients. The following sets defined in the original SPIHT algorithm are also used to present our algorithms:

- \(O(n)\): set of all offspring of node \(n\);
- \(D(n)\): set of all descendents of the node \(n\);
- \(H\): set of coordinates of all spatial orientation three roots;
- \(L(n) = D(n) - O(n)\);

Figure 2 (a) shows an example of a three-level SOT and Fig. 2 (b) is the SPIHT encoded bit-stream. According to the threshold, the bit-stream is divided into multiple bit-planes. In a bit-plane, the bits contributed from entries in LIP are outputted before those from entries of LIS and LSP. In this paper, a sequence of bits in a bit-plane contributed from a specific list is called a bit-slice. Bit sequences contributed from entries in LIP, LIS and LSP are called NZ, ZT and RF bit-slices, respectively. A node may be inserted into three lists and contributes bits to multiple bit-slices during the coding process. In Fig. 2 (b), the bits depicted in bold face are assigned to the related bit-slices. To apply such encoding flow, the offsets of all bit-slices should be calculated before encoding. Table 1 is the bit-slice offsets of various bit-planes in Fig. 2 (b). In the next sub-section, we describe how to calculate the offsets by analyzing the life cycle of a node with respect to three lists.

3.1 Life Cycle Analysis

Thus, calculation of bit-slice offsets has to be considered in order to apply the concurrent SPIHT coding. The offset of each bit-slice can be obtained by analyzing the life cycle, which describes when a node is inserted into lists, when it is removed from lists and how it contributes bits to bit-slices.
First of all, the specific bit-plane when a node turns significant is defined as:

\[
sig(n) = \begin{cases} 
0, & \text{if } c(n) = 0 \\
\left\lceil \log_2(|c(n)|) \right\rceil + 1, & \text{Otherwise} 
\end{cases}
\]

where \(c(n)\) is the coefficient value of the node \(n\). Without loss of generality, only the integer coefficients are considered. We can use the above defined bit-plane to present the life cycle of a node. \(\text{MaxBP}\) represents the maximum bit-plane of nodes, and it is defined as \(\text{MaxBP} = \max_{n \in \phi} \{\text{sig}(n)\}\). A node is born when it is inserted into LIS. A node is mature when one of its offspring turns significant. A node is dead when one of its offspring turns significant or it has no offspring. If a node becomes mature and then dies under the same threshold, it is removed from LIS. Otherwise, it is moved to the end of the list and marked as type B. The following variables are introduced to describe the life cycle of a node:

- \(\text{born}(n) = \begin{cases} 
\text{MaxBP} + 1, & \text{if } n \in H \\
\text{dead}(\text{parent}(n)), & \text{otherwise} 
\end{cases}\)
- \(\text{mature}(n) = \max_{n' \in \phi(n)} \{\text{sig}(n')\}\);
- \(\text{dead}(n) = \begin{cases} 
\max_{n' \in \phi(n)} \{\text{sig}(n')\}, & \text{if } L(n) \notin \phi \\
\text{mature}(n), & \text{otherwise} 
\end{cases}\)

First, we describe life cycle of an entry in LIS. A node inserted into LIS would generate three kinds of symbols which contribute to \(ZT\) bit-slices in different bit-planes. We named these three symbols as \(ZO, NZO\) and \(LZ\). For a node \(n\), \(ZO\) symbols are generated from bit-plane \(\text{born}(n)\) to bit-plane \(\text{mature}(n) + 1\), and each would output “0”, indicating that its offspring are insignificant. In case that \(\text{born}(n)\) is equal to \(\text{mature}(n)\), this symbol is never generated. \(NZ\) Symbols are generated in bit-plane \(\text{mature}(n)\). A NZ Symbol is composed of a “1”, the significance and the sign bits, \(\text{sign}(n)\), of the direct descendants. \(LZ\) Symbols are generated from bit-plane \(\text{mature}(n) - 1\) to bit-plane \(\text{dead}(n)\). Each LZ symbol outputs a bit to indicate the significance of \(L(n)\). In case that \(\text{mature}(n)\) is equal to \(\text{dead}(n)\), LZ symbols are not generated, either. Figure 3 shows life cycles of the four nodes around the top left corner of Fig. 2a. The node, 23, is taken as the example. It produces \(NZ\) symbols from bit-plane 6 to bit-plane 5. In bit-plane 4, it produces a \(NZ\) symbol, whose size is 9. The \(LZ\) symbol is produced at bit-plane 3.

The life cycle of each entry in LIP and LSP is analyzed in the similar way, but it requires the information of its parent. All nodes in the same \(2 \times 2\) block have the identical parent. Here, two variables are introduced:

- \(\text{PM}(b) = \begin{cases} 
\text{MaxBP} + 1, & \text{if } b \in H \\
\text{mature} (\text{parent}(b)), & \text{otherwise} 
\end{cases}\)
- \(\text{PD}(b) = \begin{cases} 
\text{MaxBP} + 1, & \text{if } b \in H \\
\text{dead} (\text{parent}(b)), & \text{otherwise} 
\end{cases}\)

Each node \(n\) in block \(b\) would output “0” to \(NZ\) bit-slice from bit-plane \(\text{PM}(b) - 1\) to bit-plane \(\text{sig}(n) + 1\), indicating that the node is still insignificant. In bit-plane \(\text{sig}(n)\), the node outputs “1” and its sign bit. If \(\text{PM}(b)\) is less than or equal to \(\text{sig}(n)\), the node never outputs any bits to \(NZ\) bit-slices. A node is inserted into LSP at bit-plane \(\text{sig}(n)\), and it outputs most significant bits from bit-plane \(\text{sig}(n) - 1\) to the end of the coding flow.

3.2 Priority Queue

In the proposed algorithm, the dynamic order of nodes is maintained by a priority queue. Nodes are inserted into the queue with a certain priority level in the way described in the following context. And, the entry with the highest priority level is encoded first. Because the granularity of a SOT is a \(2 \times 2\) block, we can just maintain the dynamic orders of blocks rather than of individual nodes. An entry in the priority queue is either Type A or Type B. A block \(b\) is inserted into the priority queue as an entry of type A when its parent node dies. Thus, the priority of the block is set as \(\text{PD}(b)\). A block \(b\) is inserted into the priority queue as a type B entry when its parent node becomes mature. Thus, the priority level of the block is set as \(\text{PM}(b)\). In case that \(\text{PM}(b)\) is equal to \(\text{PD}(b)\), the block only yields type A entry. It is noted that leaf blocks are never inserted into the queue. Thus, the size of coordinate to represent a block in priority queue is less than that to represent a node. For an \(N \times N\) image, LIS can be replaced with the priority queue whose memory capacity is \(\frac{1}{8} N^2 \times (2 \log_2 \frac{N}{4} + 1)\) bits, while \(\frac{1}{8} N^2 \times (2 \log_2 \frac{N}{3} + 1)\) bits are needed if the original SPIHT is used. Furthermore, one can implement the priority queue with a 1-D array, if the number of blocks of each priority level is known.

If we encode nodes in the order of the priority queue, the \(ZT\) bit-slices can be guaranteed to be identical to those of the original SPIHT because the order maintained by the priority queue is identical to the order of the original LIS.
Unfortunately, it is not the case for NZ and RF bit-slices. To avoid introducing additional lists, we further divide NZ and RF bit-slices. Because an entry is inserted into LIP when its parent node becomes mature, each NZ bit-slice in one bit-plane is further divided into MaxBP pieces. Similarly, a RF bit-slice is further divided into MaxBP × MaxBP pieces because an entry is inserted into LSP when the node turns significant and its parent node become mature. A ZT bit-slice is indexed by bp. A NZ bit-slice is indexed by (bp, pm) and a RF bit-slice is indexed by (bp, sig, pm). Figure 4 shows an example organization of the SPIHT encoded bit-stream and its relation to the bit-slices when MaxBP is 15.

The number of bit-slices is dependent on MaxBP. The number of ZT bit-slices is MaxBP. For a block b, it never generate bits to NZ bit-slices in bit-planes higher than PM(b). Thus, the number of the NZ bit-slices is 

\[ \text{NZ} = \sum_{bp=1}^{\text{MaxBP}} \left( \text{MaxBP} - bp + 1 \right) \]

For a node \( n \) in block \( b \), the PM(b) is always greater than or equal to sig(n). A node \( n \) only generates bits to RF bit-slices in the bit-planes lower than sig(n). Therefore, the number of RF bit-slices is

\[ \text{RF} = \sum_{bp=1}^{\text{MaxBP}} \sum_{\text{sig} = \text{bp} + 1}^{\text{MaxBP}} \left( \text{MaxBP} - \text{sig} + 2 \right) \]

Table 2 shows the number of bit-slices for different MaxBP.

| MaxBP | ZT bit-slices | NZ bit-slices | RF bit-slices | Total |
|-------|---------------|---------------|---------------|-------|
| 14    | 14            | 105           | 546           | 665   |
| 15    | 15            | 120           | 665           | 800   |
| 16    | 16            | 136           | 770           | 922   |
| 17    | 17            | 153           | 875           | 1045  |
| 18    | 18            | 171           | 980           | 1169  |

3.3 Accumulation Phase

As mentioned above, the offset of bit-slices and the number of entries in each priority level should be obtained before encoding process starts. Thus, there are two phases of the proposed algorithm: accumulation phase and encoding phase. The following algorithm describes the accumulation phase:

1) For all blocks \( b \) in SOT in reverse order:
   1.1) Let \( pm = PM(b) \) and \( pd = PD(b) \);
   1.2) If the block is not a leaf:
      * Accumulate \( Pcnt \):
         - \( Pcnt[pm] + 1 \);
         - If \( pm = pd \) then \( Pcnt[pd] + 1 \);
      * Accumulate \( ZTcnt \):
         - For \( pd \leq k \leq pm \): \( ZTcnt[k] + 1 \);
         - For each node \( n \) in \( b \):
            - If \( \text{mature}(n) \leq k \leq pd \): \( ZTcnt[k] + 1 \);
   1.3) For each node \( n \) in the block \( b \):
      * Accumulate \( NZcnt \):
         - If \( \text{sig}(n) \geq pm \): \( NZcnt[pm][k] + 1 \);
         - If \( pm - 1 \geq \text{sig}(n) \) then \( NZcnt[pm][\text{sig}(n)] + 2 \);
      * Accumulate \( RFcnt \):
         - \( RFcnt[pm][\text{sig}(n)] + 1 \);

2) Initialize all head pointers of bit-slices: \( Phead, ZThead, NZhead, RFbase \) and \( RFhead \).

The algorithm scans all blocks in reverse order, i.e. from leaves to roots. Based on the definitions of \( PM(b) \) and \( PD(b) \), to obtain \( PM(b) \) and \( PD(b) \) requires examining all coefficient values of its descendants. Thus, we store \( \text{mature}(n) \) and reuse it to obtain \( PM(b) \) and \( PD(b) \). The definition of \( PM(b) \) and \( PD(b) \) in Sect. 3.1 can be modified as:

\[
PM(b) = \begin{cases} 
\text{MaxBP} + 1, & \text{if } b \in H \\
\max_{\text{sig}(n) \in b} \text{mature}(n), & \text{otherwise} 
\end{cases}
\]

\[
PD(b) = \begin{cases} 
\text{MaxBP} + 1, & \text{if } b \in H \\
\max_{\text{sig}(n) \in b} \text{mature}(n), & \text{otherwise} 
\end{cases}
\]

The first step of the above algorithm is to accumulate the bit-count of each bit-slice and the number of entries in each priority level. \( Pcnt \) is used to accumulate the number of entries of each priority level. The bit-count of each ZT bit-slice is stored in a 1-D array, \( ZTcnt \), and the bit-count of each NZ bit-slice is stored in a 2-D array, \( NZcnt \). The bit-count of each RF bit-slice can be stored in the 2-D array, \( RFcnt \), instead of a 3-D structure because an entry in LSP generates the same number of bits in all bit-planes. The bit-count of ZT bit-slices and the number of entries in each priority level are only accumulated for non-leaf blocks. The second step is to initialize the head pointers of all bit-slices and number of entries of all priority levels. The variable, \( RFbase \), is introduced to avoid a 3-D array for RF bit-slices. \( RFbase \) stores the offset information of RF bit-slices in a bit-plane. The offset of a RF bit-slice can be then calculated from \( RFbase \) and \( RFhead \).
3.4 Encoding Phase

After head pointers of bit-slices are initialized, the encoding process starts. The encoding algorithm uses a block as the granularity to encode the roots and the leaves in the order of the priority queue. There are three operations of the algorithm, \textit{opA}, \textit{opB}, and \textit{opC}. \textit{OpA} is responsible for handling type A entries and \textit{opB} is responsible for handling type B entries. The \textit{opC} process a node and outputs the corresponding bits in \textit{NZ} bit-slices and \textit{RF} bit-slices. The encoding algorithm is shown as follows:

1) For all blocks in \(H\):
   1.1) Apply \textit{opC} on four nodes;
   1.2) Apply \textit{opA} on three offspring blocks;
2) For all entries in the Priority Queue:
   2.1) If the entry is of type A, apply \textit{opA} on four offspring blocks;
   2.2) If the entry is of type B, apply \textit{opB} on the block;

\textit{OpA} is used to handle a block. A block contains four nodes and the bits contributed to the \textit{NZ} and \textit{RF} bit-slices are generated by applying \textit{opC}:

1) \textit{opC}(n):
   1.1) For bit-planes from \(PM(b) - 1\) to \(sig(n) + 1\):
      * Outputs ‘0’ to \textit{NZ} bit-slices;
   1.2) If \(sig(n) > 0\):
      * If \(PM(b) - 1 \geq sig(n)\), then output ‘1’ and \(sign(n)\) to the \textit{ZT} bit-slice;
   1.3) For bit-planes, \(bp\), from \(sig(n) - 1\) to \(1\):
      * Output the \((bp - 1)\)-th most significant bit of \(n\) to the \textit{RF} bit-slice;

In the original SPIHT, an entry in LIS contributes to the \textit{ZT} bit-slices but it requires the sign bits of its offspring. In our algorithm, the entry is replaced with the block formed by its offspring in priority queue. After outputting bits, the entry may insert elements into the priority queue. Typically, the block \(b\) is inserted into the priority queue with priority \(PM(b)\) as a type \(B\) entry. In case that \(PM(b)\) is equal to \(PD(b)\), the four offspring blocks are inserted into the priority queue. \textit{OpA} and \textit{opB} operations are shown below:

1) \textit{opA}(b):
   1.1) Apply \textit{opC} on four nodes of block \(b\);
   1.2) Output bits to \textit{ZT} bit-slices by \textit{EncodeA}(b);
   1.3) If the block is not a leaf:
      * If \(PM(b) = PD(b)\), insert four direct descendant blocks to priority queue with priority \(PM(b)\) and set them as type \(A\);
      * Otherwise, insert its parent to priority queue with priority \(PM(b)\) set it as type \(B\);
2) \textit{opB}(b):
   2.1) Output bits to \textit{ZT} bit-slices by \textit{EncodeB}(b);
   2.2) If the block is not a leaf block:
      * Insert four direct descendant blocks into priority

queue with priority \(PD(b)\) as type \(A\);

3) \textit{EncodeA}(b):
   3.1) For bit-planes from \(PB(b)\) to \(PM(b) + 1\):
      * Output ‘0’;
   3.2) For the bit-plane \(PM(b)\):
      * Output ‘1’;
      * For each node \(n\) in \(b\):
         - If \(sign(n) = PM(b)\), then output ‘1’ and \(sign(n)\);
         - Otherwise, output ‘0’;
      * If \(b\) is not a leaf:
         - If \(PM(b) = PD(b)\), then output ‘1’;
         - Otherwise, output ‘0’;
4) \textit{EncodeB}(b):
   4.1) For bit-planes from \(PM(b) - 1\) to \(PD(b) + 1\):
      * Output ‘0’;
   4.2) For the bit-plane \(PD(b)\):
      * Output ‘1’.

4. FPGA Implementation

The proposed algorithm is also implemented on field programmable gate array (FPGA). FPGA is a reconfigurable device that offers massive parallelism, and designers can develop hardware solution without time-consuming fabrication design flow. Figure 5 shows the block diagram of the proposed architecture. We assume the coefficients and the bit-stream are stored in external memory. The COEF RAM supplies coefficients of a \(2 \times 2\) block in one cycle and the width of the bit-stream RAM is 32-bits. The Mature RAM is used to store \textit{nature}(n) of all non-leaf nodes. PQ RAM realizes the proposed priority queue. Bit-slice buffers align the output bits to 32-bit words. The arbiter chooses the most full bit-slice buffer and writes its data to the bit-stream RAM. It requires querying the head pointer before writing it to the bit-stream RAM.

Figure 6 shows the encoder of \textit{ZT} bit-slices. The encoder is composed of a LUT, and numbers of multiplexers. Figure 6(a) is the look-up table (LUT) that encodes the size and data of \textit{NZO} symbol. Figure 6(b) shows the organization of multiplexers which are responsible for choosing the correct symbols that will be outputted to the bit-slice \textit{ZT}(bp). Depending on the \(PB(b)\) and \(PM(b)\), an entry of type A would generate a \textit{ZO}, \textit{NZO} symbol or none. An en-
try of type B would generate \( LZ \) symbol or none, depending on \( PM(b) \) and \( PD(b) \). The bits of \( LZ \) symbol is also depending on the \( PD(b) \).

Figure 7 (a) shows encoders for \( NZ \) bit-slices. Since we have \( \sum_{bp=1}^{MaxBP} (MaxBP - bp + 1) \) \( NZ \) bit-slices, there are the same numbers of encoders which are LUTs with the parameters \( pm \) and \( bp \). The bits of \( LZ \) symbol is also depending on the \( PD(b) \).

Figure 7 (b) shows the encoders for \( RF \) bit-slices, which are LUTs with the parameters \( pm \), \( sig \) and \( bp \). When encoding a block \( b \), only those encoders with the parameter \( pm \) that equal to \( PM(b) \) are active. Because, all encoders concurrently generate bits to bit-slices in parallel, the proposed architecture could achieve very high throughput.

Figure 8 shows the finite state machine of the encoder and the activated components in each state. The first three states realize the accumulation phase of the proposed algorithm. In initialization state, the bit-counts and the bit-slice buffers are reset. In the second state, all coefficients are scanned in the reverse order and the Mature RAM is initialized. The bit-counts of all bit-slices are also accumulated. In the accumulation state, the head pointers of bit-slices are calculated from the bit-count information.

The last three states realize the encoding phase of the proposed algorithm. In the pre-align state, the bit-slice buffers are zero-stuffed in order to align with the head pointers. In the encoding state, blocks are encoded in the order maintained by the priority queue. Finally, the unaligned data in bit-slice buffer are flushed to the bit-stream RAM.

Because of the FPGA architecture, it may introduce a lot of delay in table lookup and wire routing. The pipeline architecture, shown in Fig. 9, is carefully designed in order to achieve high clock rate. There are seven major pipeline stages and each may take more than one cycle. In look-ahead stage, it takes the entry with the highest priority in PQ RAM. The address of the block is generated according to the type and coordinate of the entry in address generation stage. A block of coefficients is fetched in the fetch stage. The encoding stage takes three cycles to generate the bits contributed to bit-slices. In the write back stage, the
bit-count and the mature RAM are updated. The entries are also inserted into the priority queue in this stage. The generated bits are also written to the bit-slice buffers. The arbiter queries the head pointers in the abstraction stage, and writes the data to bit-stream RAM in the output stage.

The pipeline design increases the performance of the circuit, but it also introduces data hazards. In the proposed architecture, PQ RAM is read in the look-head stage, but block coordinates are written to the PQ RAM in the write-back stage. It causes the read after write (RAW) hazard. Additional control circuits are developed to detect and handle the hazards. We introduce the queue count, $Q_C$ to detect such data hazards. $Q_C$ indicates the address of the entry in PQ RAM. If an entry tries to write another entry to the PQ RAM which is close to itself, it would generate a branch event. And those pipeline stages before the write-back stage should be flushed. Fortunately, such hazards rarely happen and thus have only a little impact on the performance.

The bit-width of the bit-stream RAM is 32-bit, but a block may generate more than 32-bits to the RAM. Figure 10 shows the number of generated bits per cycle in the encoding stage. Though one may use the larger bit-slice buffers to solve the problem, it is not economic. Another solution is to introduce stalls in certain pipeline stages, but it also complicates the design. In our implementation, the encoder generates bubble cycles in the look-ahead stage when the bit-slice buffers are nearly full. It can reduce the bandwidth of the bit-stream RAM efficiently.

5. Experiment Results

5.1 Software Performance

Because all bit-planes are concurrently encoded in the proposed algorithm, the encoding process can be much faster. We compare the proposed algorithm with the original software version of SPIHT provided on the SPIHT website [21]. The comparison was made without arithmetic coding. Table 3 shows the total execution time to encode a $512 \times 512$ image. The execution environment is Intel Q6600 CPU run at 2.40 GHz with 3 GB RAM running Microsoft Windows

| Table 3 | Execution time. |
|---------|-----------------|
| bpp     | Original SPIHT (ms) | Proposed Algorithm (ms) |
| 3.57    | 110             | 18.31              |
| 3.74    | 110             | 18.84              |
| 4.14    | 130             | 20.13              |
| 4.59    | 130             | 21.39              |
| 6.08    | 160             | 25.66              |

| Table 4 | Memory requirement comparison. |
|---------|-------------------------------|
| Memory Requirement for $N \times N$ Images | $\frac{3}{16} \times N^2 \times (2\log_2 \frac{N}{2} + 1)$ | $2N^2 + 2 \frac{N^2}{4}$ | $\frac{37}{16} N^2$ | $4N^2$ | $\frac{1}{8} \times N^2 \times (2\log_2 \frac{N}{4} + 1)$ |
| Memory Requirement for $512 \times 512$ Images | 737 Kb | 655 Kb | 606 Kb | 1048 Kb | 491 Kb |
| Quality Lose | Y | Y | Y | N |

| Table 5 | FPGA synthesis results. |
|---------|-------------------------|
| FPGA Device | Altera EP2S180 |
| Maximum Clock Rate | 110 MHz |
| Total ALUTs | 36,340 |
| Total Registers | 12,422 |
| Internal Memory Bits | 194,362 |
| Execution Time for $512 \times 512$ Image | 1.438 ms |
| Throughput | 182 MPixels/sec |

| Table 6 | Throughput comparison. |
|---------|------------------------|
| Device/Process | Altera APEX 20K | Xilinx Virtex 2000E | Xilinx XC2V1000 | TSMC 0.35um | Xilinx XC4VLX15 | Altera EP2S180 |
| Clock Rate (MHz) | 33 | 56 | 133 | 100 | 41 | 110 |
| Throughput (Mpixels/sec) | 3.38 | 224 | 4.82 | 4.63 | 32 | 182 |
| Quality Lose | Y | Y | Y | Y | Y | N |
XP. The result shows that the proposed algorithm is around 6 times faster than the original SPIHT.

5.2 Memory Requirement

Another advantage of the proposed algorithm is its small memory requirement. It generates bit-stream identical to that of the original SPIHT algorithm, so no image quality loss is induced. Table 4 compares the memory requirements of various works. The memory requirement of the proposed algorithm is the minimum.

5.3 FPGA Performance

Our FPGA implementation is targeted to Altera EP2S180 Device. Table 5 shows the synthesis results. COEF, Mature, PQ and bit-stream RAMs are not synthesized as the FPGA internal memory. The internal memory bits are consumed in bit-slices buffers. The maximum clock of the circuit is 110 MHz. It takes many ALUTs and registers, because it concurrently encodes all bit-planes. Around 1.438 ms is required to encode all bit-plane of a 512×512 image. Thus, the throughput of the proposed design is about 182 MPixels/sec.

Table 6 compares the throughputs of various works. The memory requirement of the proposed algorithm is around 1.438 ms. Thus, the throughput of the proposed design is about 182 MPixels/sec. Table 6 compares the throughputs of various works.

6. Conclusion

In this paper, we propose a memory efficient SPIHT algorithm and its parallel implementation. Instead of using three original lists of SPIHT, we adopt a priority queue and used a block as the granularity to reduce the memory requirement without sacrificing image quality. For a 512×512 image, the memory requirement is reduced from 5.83 Mb to 491 kb. Moreover, all bit-planes are encoded concurrently in order to reduce the execution time. The results show that the software encoder of the proposed algorithm is 6 times faster than the original SPIHT encoder [21]. The proposed algorithm is also realized on the FPGA. With careful pipeline design, the circuit can run at 110 MHz, which can encode a 512×512 image in 1.438 ms. Thus, its throughput is as high as 182 MPixels/sec and it can be applied to high performance image compression applications.

Acknowledgement

This work is partially supported by National Science Council, Taiwan, under grant 98A106.

References

[1] A. Said and W.A. Pearlman, “A new, fast, and efficient image codec based on set partitioning in hierarchical trees,” IEEE Trans. Circuits Syst. Video Technol., vol.6, no.3, pp.243–250, 1996.

[2] R.K. Bhattar, K.R. Ramakrishnan, and K.S. Dasgupta, “Strip based coding for large images using wavelets,” Signal Process., Image Commun., vol.17, pp.441–456, 2002.

[3] L.W. Chew, L.-M. Ang, and K.P. Seng, “New virtual SPIHT tree structures for very low memory strip-based image compression,” IEEE Signal Process. Lett., vol.15, pp.389–392, 2008.

[4] C. Chrysafis and A. Ortega, “Line-based, reduced memory, wavelet image compression,” IEEE Trans. Image Process., vol.9, no.3, pp.378–389, 2000.

[5] J. Ritter and P. Molitor, “A pipelined architecture for partitioned DWT based lossy image compression using FPGAs,” Proc. 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays Monterey, pp.201–206, California, United States, 2001.

[6] P. Corsonello, S. Perri, P. Zicari, and G. Cocorullo, “Microprocessor-based FPGA implementation of SPIHT image compression subsystems,” Microprocessors and Microsystems, vol.29, pp.299–305, 2005.

[7] J. Ritter, G. Fey, and P. Molitor, “SPIHT implemented in a XC4000 device,” in Circuits and Systems, 2002. MWSCAS-2002. The 2002 45th Midwest Symposium, vol.1, pp.J-239–242, 2002.

[8] M.S. Wegmueller, D. Perels, T. Blaser, S. Senn, P. Stadelmann, N. Felber, and W. Fichtner, “Silicon implementation of the SPIHT algorithm for compression of ECG records,” IEEE International Midwest Symposium on Circuits and Systems, vol.2, pp.381–385, 2006.

[9] J. Jyotheshwar and S. Mahapatra, “Efficient FPGA implementation of DWT and modified SPIHT for lossless image compression,” J. Syst. Archit., vol.53, no.53, pp.369–378, 2007.

[10] Y. Sun, H. Zhang, and G. Hu, “Real-time implementation of a new low-memory SPIHT image coding algorithm using DSP chip,” IEEE Trans. Image Process., vol.11, no.9, pp.1112–1116, 2002.

[11] J.R. Ding and J.F. Yang, “A simplified spith algorithm,” J. Chinese Institute of Engineers, vol.31, pp.715–719, 2008.

[12] F.W. Wheeler and W.A. Pearlman, “SPIHT image compression without lists,” in Acoustics, Speech, and Signal Processing, 2000. ICASSP ’00. Proc. 2000 IEEE International Conference, vol.4, pp.2047–2050, 2000.

[13] L.-M. Ang, H.N. Cheung, and K. Eshraghian, “A dataflow-oriented VLSI architecture for a modified SPIHT algorithm using depth-first search bit stream processing,” in Circuits and Systems, 2000. Proc. ISCAS 2000 Geneva. The 2000 IEEE International Symposium, vol.1, pp.291–294, 2000.

[14] L.-M. Ang and H.N. Cheung, “Hardware implementation of the depth first search bit stream SPIHT system,” in Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium, vol.4, pp.518–521, 2001.

[15] T.-H. Chiang and L.-H. Dung, “A VLSI progressive coding for wavelet-based image compression,” IEEE Trans. Consum. Electron., vol.53, no.2, pp.569–577, 2007.

[16] V.R. Algazi and R.R. Estes, Jr, “Analysis-based coding of image transform and subband coefficients,” in International Society for Optical Engineering, p.11, 1995.

[17] W.-B. Huang, Y.-J. Chang, A.W.Y. Su, and Y.-H. Kuo, “VLSI design of a DWT/modified efficient SPIHT based image codec,” in Information, Communications and Signal Processing, 2003 and the Fourth Pacific Rim Conference on Multimedia. Proc. 2003 Joint Conference of the Fourth International Conference, vol.1, pp.248–252, 2003.

[18] T.W. Fry and S.A. Hauck, “SPIHT image compression on FPGAs,” IEEE Trans. Circuits Syst. Video Technol., vol.15, no.9, pp.1138–1147, 2005.

[19] W.-B. Huang, A.W.Y. Su, and Y.-H. Kuo, “VLSI implementation of a modified efficient SPIHT encoder,” IEICE Trans. Fundamentals, vol.89-A, no.12, p.3613, 2006.

[20] M. Akter, M.B.I. Reaz, F. Mohd-Yasin, and F. Choong, “Hardware implementations of an image compressor for mobile communications,” J. Communications Technology and Electronics, vol.53, pp.899–910, 2008.

[21] SPIHT Image Compression: Properties of the Method. [Online]. Available: http://www.cipr.rpi.edu/research/SPIHT/spiht1.html

[22] M. Lanuzza, S. Perri, P. Corsonello, and G. Cocorullo, “An efficient wavelet image encoder for FPGA-based designs,” in Signal Processing Systems Design and Implementation, 2005. IEEE Workshop on, pp.652–656, 2005.
[23] A.V. Nandi and R.M. Banakar, “Hardware modeling and implementation of modified SPIHT algorithm for compression of images,” in Industrial and Information Systems, 2007. ICIIS 2007. International Conference, pp.329–334, 2007.

Zhong-Ho Chen was born in Taichung, Taiwan, in 1981. He received the B.S. and M.S. degrees in Computer Science and Information Engineering from National Cheng-Kung University, Taiwan, in 2003 and 2005, respectively. He is currently working for the Ph.D degree in Computer Science and Information Engineering from National Cheng-Kung University, Taiwan. His research activities include Digital Signal Processing, VLSI Design, Low-power Design and Computer Architecture.

Alvin W.Y. Su was born in Taiwan, in 1964. He received the B.S. degree in control engineering from National Chiao-Tung University, Hsinchu, Taiwan, in 1986, and the M.S. and Ph.D. degrees in electrical engineering from Polytechnic University, Brooklyn, NY, in 1990 and 1993, respectively. From 1993 to 1994, he was with the Center for Computer Research in Music and Acoustics (CCRMA), Stanford University, Stanford, CA. From 1994 to 1995, he was with Computer Communication Laboratory, Industrial Technology Research Institute, Taiwan. In 1995, he joined the Department of Information Engineering and Computer Engineering, Chung-Hwa University, Taiwan, where he serves as an Associate Professor. He joined the Department of Computer Science and Information Engineering, National Cheng-Kung University, Tainan, Taiwan, in 2000. His research interests cover the areas of digital audio signal processing, physical modeling of acoustic musical instruments, human computer interface design, pattern recognition, data compression, image/video signal processing, and VLSI signal processing.