Efficient Decoding of Compressed Data

Mostafa A. Bassiouni and Amar Mukherjee
Department of Computer Science, University of Central Florida, Orlando, FL 32816

In this article, we discuss the problem of enhancing the speed of Huffman decoding. One viable solution to this problem is the multibit scheme which uses the concept of *k*-bit trees to decode up to *k* bits at a time. A linear-time optimal solution for the mapping of 2-bit trees into memory is presented. The optimal solution is derived by formulating the memory mapping problem as a binary string mapping problem and observing that at most four different 4-bit patterns can occur within any 2-bit Huffman tree. In addition to reducing the processing time of decoding, the optimal scheme is storage efficient, does not require changes to the encoding process, and is suitable for hardware implementations.

Introduction

One of the popular data compression methods is the Huffman encoding technique (Huffman, 1952), which takes advantage of the skewness of the frequency of input symbols. Accordingly, the most frequent symbols are assigned to the shortest codes and all larger codes are constructed so that shorter codes do not appear as prefixes. Simply, the Huffman method builds a decode tree (i.e., a binary tree in which leaf nodes represent symbols) having minimal external path length. If the set of symbols is given by $\{A_1, A_2, \ldots, A_v\}$, the probability of occurrence of symbol A_k is p_k, and the distance from the root of the tree to the leaf node corresponding to symbol A_k is d_k, then the Huffman tree minimizes the quantity $\sum_{j=1}^{v} p_j * d_j$. Huffman encoding has been used in Liu and Yu (1991), along with dictionary compression for the efficient storage of large databases. Huffman compression is also used in the JPEG image compression standard to store the AC values obtained via DCT coding. Huffman encoding, arithmetic coding (Witten, Neal, & Cleary, 1987) and the LZW scheme (Welch, 1984) are used in conjunction with lossy schemes to improve the fidelity of compressed images at a given level of compression (Bassiouni, 1993). Efficient VLSI designs for Huffman compression is given in Mukherjee, Rangandthan, and Bassiouni (1991b).

Figure 1 gives an example Huffman tree for the 12 symbols, A, B, C, D, E, F, G, H, I, J, K, L, whose weights (frequencies of occurrence) are assumed to be 4, 3, 4, 1, 1, 2, 8, 2, 1, 1, 1, and 4, respectively. During decoding, the compressed file is processed serially (one bit at a time) and the Huffman tree is repeatedly traversed from its root to the leaf nodes. For example, the bit sequence "001" causes a movement from the root of the Huffman tree of Figure 1 to the leaf node of symbol B. In this article, we concentrate on the problem of improving the efficiency of the decoding (decompression) process of Huffman and other similar tree-based compression schemes.

Improving the decoding process is important for the following reasons:

(1) The decoding phase in tree-based compression schemes is bit-serial and is therefore inherently slow; bit-serial decoding can benefit the most from better implementations.
(2) Some applications use large volumes of static data that are retrieved repeatedly. While the encoding process is executed once to store these data in compressed form, the decoding routines are frequently invoked to decompress the stored information whenever the users issue read requests.

In the following sections, we shall discuss a scheme, called multibit or *k*-bit decoding, for reducing the time overhead of the bit-serial decoding operation and present the optimal solution for the 2-bit mapping problem; the terms "*k*-bit decoding" and "multibit decoding" will be used interchangeably throughout. The multibit method has different thrust and orientation than previous proposals. Choueka et al. (1985) proposed a method to reduce the time requirement by using a hierarchy of tables whose total size could grow exponentially with *k*, the number of bits per block to be decoded. They also presented a revised version that needs storage $O(n * \log(n))$ to $O(n^2)$, where *n* is the number of leaf symbols. For the storage of internal nodes, our multibit scheme needs sublinear storage for *k* greater than or equal to 2.

Received August 31, 1992; revised August 10, 1993 and November 19, 1993; accepted February 10, 1994.

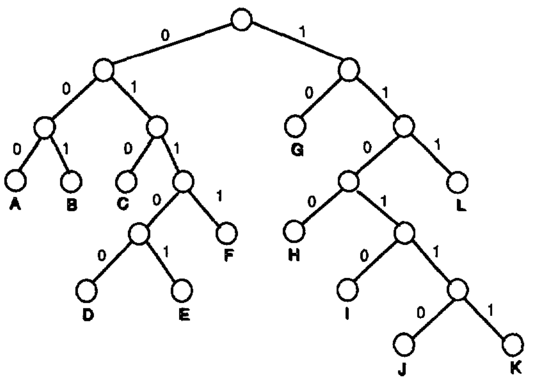

FIG. 1. An example Huffman tree.

Hirschberg and Lelewer (1990) proposed a series of methods that are suitable when the amount of memory available during the decode operation is limited but the encoder is allowed substantial storage and computational resources (to construct coded text which the space-limited decoder can translate efficiently). The optimal scheme presented in this study, on the other hand, does not require any change in the encoding operation and is applicable to other tree-based codes (for a discussion of these codes and their properties, the reader is referred to the review given in Lelewer & Hirschberg [1987]). In addition, the optimal scheme offers more parallelism, is also storage efficient, and is more suitable for hardware implementations.

k-Bit Decoding

In Huffman decoding, the compressed bit stream is processed serially one bit at a time. Basically, the decoding operation produces data characters (symbols) by repeatedly traversing the Huffman tree, from the root to the leaf node, under the control of the input bits; a bit value of 1 initiates a visit to the right child, whereas a value of 0 results in a visit to the left child. This process is inherently slow and, because of its strict sequential nature, is not amenable to elegant parallel implementations. The availability of a large number of processors within a parallel machine, for example, may be used to decode simultaneously several files (or records) that were encoded separately, but the sequential decoding of each file needs only one processor at a time and gains no appreciable improvement by the increased scale of parallel hardware. A viable approach to improve the speed, however, can be based on a different concept, namely, using k bits at a time in each step of the decoding process. The problem of "multibit" or "k-bit" decoding has been motivated in an earlier conference paper (Mukherjee et al., 1991a) which also presented a high-level VLSI design for a basic k-bit encoder/decoder. In this article, we give a new formulation for the k-bit decoding problem, and present the optimal memory mapping for 2-bit Huffman trees.

Consider the Huffman tree shown in Figure 1. The first step is to obtain the corresponding k-bit tree. Each edge in this tree corresponds to the encoding of a maximum of k bits of the code. If the length of the code-word is n bits, it is represented by a sequence of $\lceil(n/k)\rceil$ edges in the unique path from the root to the leaf node; only the last edge leading to the leaf node could possibly have a label with less than k bits. The tree of Figure 1 can be viewed as a 1-bit tree; the corresponding 2-bit tree is shown in Figure 2 (labels inside nodes in Fig. 2 represent the id or node number of each node). The two trees of Figure 1 and Figure 2 are equivalent; any one of these two trees can be easily and uniquely constructed from the other. The code of a character in a k-bit tree is obtained by concatenating the labels read from the root to the leaf node of that character.

The *k*-Bit Decode Table

The purpose of the k-bit scheme is to achieve faster decoding by processing k bits at a time. To maximize the benefit obtained by this scheme, the overhead associated with processing the k-bit sequences should be minimized. In particular, the k-bit decode table must be carefully designed to allow for fast lookup and tree traversal. Below, we discuss a design approach that, in addition to being suitable for efficient software implementation, is quite attractive and very suitable for hardware/VLSI technology.

Consider a k-bit Huffman tree whose nodes are numbered 0, 1, . . . , N (assume 0 is the index of the root as shown in Fig. 2). A table of size $M \geq N + 1$ is used to store appropriate information about the nodes of the k-bit tree (we call this table the k-bit decode table). A node j in this tree, $0 \leq j \leq N$, is mapped to a unique entry r in the table, $0 \leq r \leq M - 1$. To enhance the speed and reduce the number of memory references of the decoding process, a single pointer value (called the "base") is

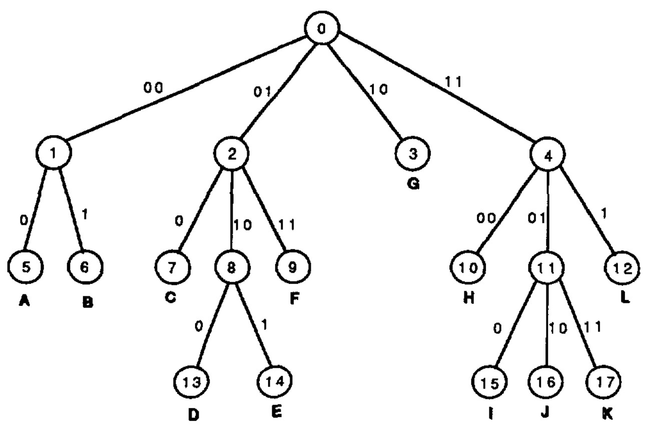

FIG. 2. A corresponding 2-bit Huffman tree.

stored in the entry of each parent node and is used to access all the children of that node. If the edge connecting a child node to its parent has a label of value L and the base value stored in the parent is B, then the child should be stored in the table entry whose index is given by $B + L$. This mapping yields a simple and efficient decoding logic. In each decoding step, one or two input bits are read by the decoder and are used to compute the value of the label associated with the next node, L. The base address, B, is fetched from the "current node" and the child node located in entry $B + L$ is chosen to replace its parent as the "current node" of the decoding operation. Based on this simple logic, the properties of the desired mapping can be described as follows:

(1) If a node, say node v, has the maximum fan-out (i.e., it has 2^k children), all the children of this node are mapped to contiguous table entries. The children are ordered according to the labels of the edges connecting them to their common parent v. Thus, the child with label "00 . . . 0" occupies the entry having the lowest address in the contiguous block (the index of this entry is the base value B stored in the entry of node v). Similarly, the child with label "11 . . . 1" occupies the entry having the highest address.
(2) If a node has less than 2^k children, the mapping of these children must preserve the same relative positions that would have been obtained if the node had maximum fan-out. For example, if $k = 3$ and a node has three children whose edges have the labels "001," "011," and "110," then the three children should be mapped to entries $B + 1$, $B + 3$, and $B + 6$, respectively, for some integer base value B. Notice that entries B, $B + 2$, $B + 4$, $B + 5$, and $B + 7$ are not set aside for the children of this node and may therefore be utilized for other nodes.
(3) Only the mapping of nodes having a common parent need to obey the above rule. There is no restriction imposed on the relative locations of the two entries to which a node and its child are mapped or to which two unrelated nodes are mapped. Because the root node is the only node in a Huffman tree that does not have siblings, it can therefore be stored arbitrarily in any free entry in the table.
(4) To optimize the design (especially for VLSI and associative memory), the size M of the decode table must be minimized, that is, M should be as close to $N + 1$ as possible.

The above mapping of tree nodes into entries of the decode table will enable us to construct an efficient decoding operation with simple logic as discussed before. In what follows, we shall elaborate on the decoding design and present the optimal mapping for 2-bit Huffman trees. First, we shall formulate the design of the k-bit decode table as a binary string mapping problem. The formulation is applicable to any tree-based codes (e.g., Shannon–Fano codes [Shannon & Weaver, 1949; Fano, 1949]; Fibonacci codes [Fraenkel & Klein, 1985; Lelewer & Hirschberg, 1987]; and Huffman codes [Huffman, 1952], etc.).

Descendent Strings

For each nonleaf node in the k-bit tree, we associate a binary string (called the descendent string) of length 2^k. A bit in this string is set to 1 only if the index (position) of this bit is equivalent to the label of the edge leading to a child of this node. If the edge label for a child has less than k bits, extra zeros are appended to this label, at the least significant (rightmost) positions, to obtain a k-bit field that can be used as an index into the descendent string.

For example, if $k = 3$ and a node has four children with edge labels "0," "100," "101," and "11," the corresponding 8-bit descendent string is "10001110." Notice that the short labels "0" and "11" are first extended to become "000" and "110" and then used to set the two bits at positions 0 and 6 of the string.

Remarks: Descendent strings are used to ensure that the solution obtained for the decode-table mapping problem preserves the correct relative positions of the children of a common parent. Based on the above method of construction, these descendent strings satisfy the following:

(1) Each child node corresponds to a unique 1 in the descendent string of its parent. Notice that appending zeros to short labels at the rightmost position (rather than the leftmost position) preserves this uniqueness.
(2) The total number of 1's in all descendent strings is equal to N, the number of nonroot nodes in the k-bit Huffman tree. Because the root of the tree is not accounted for in any of these descendent strings, we shall add a special 1-bit string of value "1," called the root string, to take care of the mapping of the root node. Since leaf nodes do not have children, they all have identical descendent strings of the form "00 . . . 0." Soon it will be clear that these descendent strings (all zeros) will not need to be considered in our search for the optimal mapping.

Mapping of Descendent Strings

In this section, we present the binary string mapping problem which is equivalent to the problem of designing the k-bit decode table. Simply, the set of descendent strings is used to construct a binary string, W, which satisfies certain properties. In general, the string W is not unique, but its properties ensure that the problem of finding the optimal mapping for the k-bit decode table is equivalent to the problem of finding the string W which, when not counting any leading or trailing zeros, has the minimum length. Below, we discuss the binary string mapping problem in detail.

Given the descendent strings of the nonleaf nodes of a k-bit Huffman tree augmented with the 1-bit root string, a binary string W is constructed such that:

(1) Each descendent string is mapped to 2^k consecutive

bits in W. The root string is mapped to a single bit in W.

(2) Overlapping of descendent strings or the root string within W is permitted provided that no two bits having value 1 are mapped to the same position in W.

(3) Each bit in W is covered by at least one descendent string or the root string, that is, for each bit in W, there is at least one descendent-string bit or the root string that is mapped to it.

(4) The value of each bit in W is obtained by performing the bitwise OR operation on all the bits that are mapped to it (notice that at most one of these bits is allowed to have a value of 1).

If W is constructed as above, then the number of 1's in W is equal to the number of nodes $N + 1$, that is, each node in the k-bit tree is associated with a unique 1 in W. Assume that after truncating any leading and trailing zeros from W, the resulting string, say S, is of size M bits. A decoding table of size M entries is then constructed. A node in the k-bit tree is mapped to the entry of the decode table whose index is equal to the index of the unique 1 associated with this node in S. To optimize the design, the value of M should be minimized.

The *k*-Bit Contiguous Binary Superstring (CBS) Problem

We now summarize the mapping problem discussed above. The problem is a different version of the one posed in Mukherjee et al. (1991a). The formulation is applicable to any tree-based codes (e.g., Shannon–Fano codes, Fibonacci codes, universal codes of Elias [1975], Huffman codes, etc.). In this article, we shall concentrate on solving the problem for Huffman codes.

Instance: A collection of binary strings (called descendent strings) of length 2^k bits each, and a single 1-bit string (called the root string) whose value is "1." Let N be the total number of 1-valued bits in all the descendent strings.

Problem: Find a binary string $W = 0^*S0^*$ where S is a minimum-length binary string, with no leading or trailing zeros, which has exactly $N + 1$ 1-valued bits such that every descendent string and the root string can be positioned (aligned) within W so that each 1-valued bit in these strings corresponds (is mapped) to a unique 1 in S.

In the above definition, the notation 0^* is used to denote an all-zero string of arbitrary length (possibly empty), and 0^*S is used to denote the concatenation of the two strings 0^* and S.

Definition

The vacancy ratio of the solution for the CBS problem is defined as the ratio $(M - N - 1)/M$ and the expansion ratio is defined to be $(M - N - 1)/(N + 1)$, where M is the size of the resulting string S and N is the number of 1-valued bits in all the descendent strings (N corresponds to the number of nonroot nodes in the 2-bit tree).

Example 1

For $k = 3$, consider the three descendent strings

$D1$ = "01010000" $D2$ = "01001000"

$D3$ = "10000100"

and the root string "1." The value of N in this case is 6 and the string S will have a size of at least 7 bits. The optimal solution in this case is S = "1111111" and $W = 0S00$ = "0111111100." The starting positions (which we call the CBS indexes) of the strings $D1$, $D2$, and $D3$ within W are 2, 0, and 2, respectively. The CBS index of the root string is 6. The 7-bit string S implies that the optimal solution achieved the lower bound implied by the total number of nodes in the tree.

If the third descendent string in the above example is changed to $D3$ = "10000101," then the value of N becomes 7 and the optimal solution in this case is $W = 0S$ = "0111111101." The CBS indexes remain unchanged. The 9-bit string S implies that we need to use a decoding table of $M = 9$ entries. One entry in this table (the one before the last) is not used to store decoding information, but may be freely used to store any other data. The fraction of table entries that are not used is given by the vacancy ratio which, in this case, is equal to 1/9. The expansion ratio of 1/8, on the other hand, gives the ratio between the extra (nonused) space to the original number of nodes in the tree.

Notice that the descendent strings need not be distinct since several nodes (e.g., nodes 1 and 8 in Fig. 2) may have the same pattern of descendent edges. Notice also that one might opt to store the entry of the root node in a separate register to speed up its access. If this is the case, the 1-bit root string is no longer needed because the root node will not occupy any entry in the decode table. By removing the root node from the decode table, the CBS problem stated earlier remains the same except that the required mapping is applied only to the descendent strings of the nonleaf nodes (i.e., the root string is omitted). This modification does not change the complexity of the CBS problem nor change the optimal solution presented later in this section. The values of the vacancy ratio and the expansion ratio for the modified problem are given by $(M - N)/M$ and $(M - N)/N$, respectively. In general, solving the CBS problem seems to require an exhaustive search, but suboptimal solutions can be obtained using a variety of fast heuristic algorithms. The CBS problem has the flavor of some compute-bound string matching problems (e.g., the multiple string alignment problem [Sankoff, 1985] and the superstring problem [Tarhio & Ukkonen, 1988]) but is quite distinct from them. We conjecture that the general k-bit CBS problem is NP-hard. Proving this conjecture is posed as

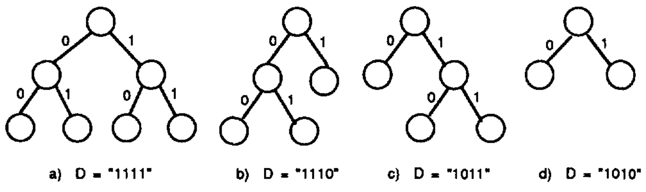

FIG. 3. Valid topologies for 1-bit Huffman trees of height 1 or 2.

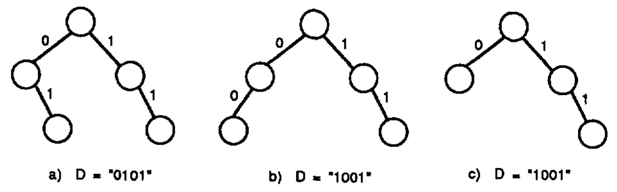

FIG. 5. Some invalid topologies/codes for the 1-bit Huffman case.

an open problem. Fortunately, the special case of Huffman decoding offers some useful properties that help tackle the CBS problem. For example, descendent strings in 2-bit Huffman trees have only four valid patterns (out of 16 distinct ones) and those in 3-bit Huffman trees have only 25 valid patterns (out of 256 distinct ones). In addition, we are particularly interested in the case of $k = 2$, because it represents the most suitable and viable value for hardware implementations. We shall therefore concentrate on solving the CBS problem for 2-bit Huffman trees.

Lemma 1

For 2-bit Huffman trees, the descendent strings of nonleaf nodes have only four 4-bit patterns: "1111," "1110," "1011," and "1010."

Proof of this lemma is based on the simple observation that every nonleaf node in a 1-bit Huffman tree must have two children. Consider a subtree rooted at any nonleaf node within a 1-bit Huffman tree such that this subtree contains all descendent nodes that are within a distance of two from the root of the subtree. Figure 3 lists all valid topologies for such a subtree along with their descendent strings. The corresponding topologies for the equivalent 2-bit Huffman trees are shown in Figure 4.

As illustrated in Figure 4, every nonleaf node in a 2-bit Huffman tree must have either four children (corresponding to the descendent string D = "1111" as in case a), three children (D = "1110" as in case b or D = "1011" as in case c) or two children (D = "1010" as in case d). Other patterns, for example, "0101" and "1001" as shown in Figure 5, are not possible because of the sibling property of Huffman trees and the method used to append zeros to short labels during the construction of descendent strings (see Descendent Strings subsection). Consider, for example, the string "1001." One can easily verify that the two topologies shown in Figure 5b and 5c are the only 1-bit tree topologies that produce this string. Because both topologies are not valid Huffman trees, the string "1001" cannot be a descendent string of any node in a Huffman tree.

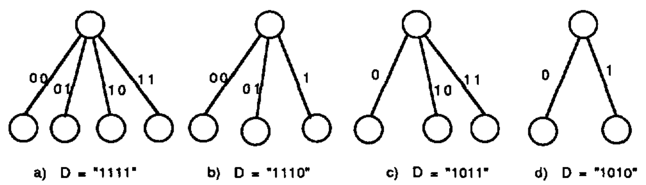

FIG. 4. Corresponding topologies for the 2-bit Huffman trees.

Remarks:

(1) For 2-bit Huffman trees, Lemma 1 implies that there are no leading zeros in the string W used in the definition of the CBS problem. Consequently, the CBS indexes obtained by the optimal solution are the same as the "base" indexes stored in the decode table.
(2) Several tree topologies may have the same descendent string, for example, cases b and c of Figure 5. Consider, for example, adding a single child with label 0 to the leftmost leaf node of the 1-bit Huffman tree shown in Figure 3d. The resulting tree is not a valid Huffman tree, but it has the same descendent string "1010" of the Huffman tree of Figure 3d. All we need to emphasize at this point is that the four patterns listed in Lemma 1 are the only descendent strings that can be encountered when dealing with 2-bit Huffman trees.

Based on Lemma 1, we can now construct an optimal algorithm for solving the 2-bit CBS problem for Huffman trees. The idea is to cluster the descendent strings into four groups. Group G1 contains all strings of value "1111." These strings are simply placed consecutively, one after the other, on 4-bit contiguous fields. The second group, G2, contains strings of the form "1110." Again, we map these strings onto 4-bit fields such that the first bit of a field overlaps with the last bit of the previous field. The third group, G3, and the fourth group, G4, contain strings of the form "1011" and "1010," respectively. First, we repeatedly try to pair one string from G4 with one string from G3 (shifted one bit to the right) and map them onto a 5-bit field. Next, we repeatedly try to pair two G4 strings onto the 4-bit field "1111" (this is done by shifting one string one bit to the right and then discarding its trailing zero). Finally, any remaining strings are mapped separately onto 4-bit consecutive fields, except possibly a single remaining G4 which is mapped to 3 bits. The root string is mapped to any 0-valued bit or, if there are no such bits, is mapped to a new entry at the end of the table. A high-level description of the mapping algorithm is given below. The notation $|G|$ represents the size of the set G.

Algorithm CBS_2H

```
Input:    A collection of 4-bit descendent strings (not necessarily distinct)
          and the 1-bit root string.

Output:   The CBS index of each string and the size of the decode table.

Method:   Cluster the input strings by pattern in 4 groups
          I = 0 ; R = 0; /* initialization */
          For j = 1 to | G1 | do        /* pattern is "1111" */
                { map the jth member of G1 to I; I = I + 4 }
          For j = 1 to | G2 | do        /* pattern is "1110" */
                { map the jth member of G2 to I ; I = I + 3 }
          For j = 1 to min {| G3 |, | G4 |} do
                { map the jth member of G4 to I;        /* "1010" */
                  map the jth member of G3 to I+1;      /* "1011" */
                  I = I + 5 }
          If( | G3 | > | G4 | ) then
                  for j = | G4 | + 1 to | G3 | do        /* "1011" */
                        { map the jth member of G3 to I ; R=I+1; I = I + 4 };
          If( | G4 | > | G3 | ) then
                {for j = | G3 |+2 to | G4 | step 2 do /* "1010" */
                        { map the (j−1)th member of G4 to I ;
                          map the jth member of G4 to I+1 ; I = I + 4}
                  If( | G4 | − | G3 | is odd) then
                        {map the last member of G4 to I; R=I+1 ; I = I + 3}
                }
          If(R=0) {R=I ; I=I+1 } ;
          Map root node to R
          M = I   /* M is the size of the decode table */
```

The variable R in the above code is used to remember whether there is at least one 0-valued bit that can be allocated to the root string.

Example 2

For the 2-bit Huffman tree of Figure 2, the number of nonroot nodes is $N = 17$ and the descendent strings of the 6 nonleaf nodes are as follows:

Node 0:	D_0 = "1111"
Node 1:	D_1 = "1010"
Node 2:	D_2 = "1011"
Node 4:	D_4 = "1110"
Node 8:	D_8 = "1010"
Node 11:	D_{11} = "1011"

In the previous example, algorithm CBS_2H produces a string S of 18 consecutive 1's and generates six CBS indexes that map D_0 to 0, D_4 to 4, D_1 to 7, D_2 to 8, D_8 to 12, and D_{11} to 13. The root node is mapped to location 17. The vacancy ratio of this mapping is zero.

Lemma 2

For 2-bit Huffman trees, the linear-time algorithm CBS_2H is optimal, that is, it produces a string S of minimum length.

Proof of the above lemma can be established by considering the four patterns of valid descendent strings in 2-bit Huffman trees. Notice that the algorithm produces compact mapping (without any expansion) for patterns "1111" and "1110" as well as for "1011"/"1010" and "1010"/"1010" string pairs. The only expansion introduced by the algorithm is due to either (i) a single (left-over) string of value "1010;" or
(ii) strings of value "1011" which are in excess of their "1010" counterparts.

Notice that at most one type of expansion (i or ii) can occur for any given 2-bit Huffman tree. It is easy to see that if such expansion occurs, it is indeed necessary. In other words, any valid mapping will produce a string S with a number of 0's equal to or greater than the number of left-over strings causing the expansion.

Lemma 3

The upper bound for the vacancy ratio of algorithm CBS_2H is 0.25 (corresponding to a worst case expansion ratio of 1/3).

It is easy to see that this worst case ratio can only be obtained if all descendent strings have value "1011." Notice that, in this case, there is only one valid mapping that can be used to solve the CBS problem. This worst case is, however, highly unlikely. Typical values of the vacancy ratio for the optimal mapping are usually much smaller (and are often zero) due to the pairing of strings in $G3$ and $G4$, and the freedom of allocating any vacant entry to the root node.

The Optimal 2-Bit Decoding Process

For the tree of Figure 2, a decode table of 18 entries (numbered 0 through 17) is used to store the nonroot nodes of the tree. The mapping of a nonroot tree node into an entry in the decode table is obtained by adding the following two components: (i) the CBS index of the descendent string of the parent of this node; and (ii) the label of the edge connecting this node to its parent. As explained before, labels of length one bit are extended to 2 bits by appending a rightmost zero. For example, node 6 (symbol B) in Figure 2 is mapped to entry 9 in the decode table; this is obtained by adding the CBS index of string D_1 obtained in Example 2 (i.e., decimal value 7) to the extended label "10" (decimal value 2).

A field in the decode table, called "base," is used to store the CBS index for each nonleaf node. Recall that for the tree of Figure 2, these CBS indexes are as follows

Node #	0	1	2	4	8	11
CBS index	0	7	8	4	12	13

In the case of leaf nodes, the "base" field is used to store the output code of the corresponding symbol. We assume that the value of "base," or a flag bit in it, can be used to determine whether the corresponding node is a terminal (symbol) node or not. Alternatively, a separate Boolean flag can be used for this purpose. The basic loop of the decoding process proceeds as follows: (a) read two bits from the compressed file; (b) add these two bits, treated as a 2-bit integer, to the value of the "base" field of the current node; and (c) the result gives the index of the node to be visited next.

There is now one last issue that needs to be solved, namely, handling short labels. The last edge leading to a leaf node may have a label of length one bit (rather than 2). In this case, we should only use the first input bit (appended with 0) to complete the current decoding process. The other (nonused) input bit should be attached to a new bit from the compressed file and the resulting two bits are then used to start a new decoding operation from the root of the 2-bit tree. To accomplish this, two Boolean flags f_0 and f_1 in the decode table are used to indicate short labels as follows: if the next input bit has a value j and flag f_j has a value of 1 (True), then an edge with a short label is encountered. For example, the value of the two flags (f_0,f_1) for nodes 2, 4, and 8 of Figure 2 are (1,0), (0,1), and (1,1), respectively. The two flags are not needed for leaf nodes; their values in this case are immaterial. Table 1 shows the decode table for the tree of Figure 2 based on the optimal mapping obtained in Example 2. The decode table has 18 entries (numbered 0 through 17); each entry contains the three fields: base, f_0, and f_1. For clarity, Table 1 also gives the sequential index of each table entry as well as the index (node #) of the tree node assigned to that entry. These latter two fields are included for the purpose of clarification; they are not actually stored in the decode table.

Algorithm Decode_2H below gives a high-level description of the 2-bit Huffman decoding algorithm. The

TABLE 1. Optimal decode table for the tree of Figure 2.

Node # (see Fig. 2)	Entry index	Base	f_0	f_1
1	0	7	1	1
2	1	8	1	0
3	2	G		
4	3	4	0	1
10	4	H		
11	5	13	1	0
12	6	L		
5	7	A		
7	8	C		
6	9	B		
8	10	12	1	1
9	11	F		
13	12	D		
15	13	I		
14	14	E		
16	15	J		
17	16	K		
0 **root**	17	0/* zero*/	0	0

algorithm uses a decode table denoted DT. The auxiliary variables Base, $F[0]$, and $F[1]$ are used to store the base, f_0, and f_1 fields, respectively, of the current node, while the variables $v[0]$ and $v[1]$ are used to hold the two input bits currently being processed.

```
Algorithm Decode_2H

/* 2-bit Huffman decoding */
while (not end of file) do
  initialize Base, F[0], and F[1] from root entry
  Repeat
    store one input bit into v[1]
    if( F[v[1]] = 1 )
    then v[0] := 0 /* short label */
    else store another input bit into v[0] endif;
    offset := integer {v[1]v[0] } /* form a 2-bit integer */
    Next := Base + Offset
    Base := DT[Next].base
    if(Base is not a symbol)
    then {F[j] := DT[Next].f_j   for j=0,1}
    else {Output the symbol stored in Base} endif;
  Until (Base is a symbol)
endwhile;
```

Other Considerations

Although our primary concern is reducing the decoding time, the multibit approach is also storage efficient. Notice that if n is the number of leaf nodes (symbols in the alphabet), then the number of internal nodes is reduced from $n - 1$ in the case of standard Huffman to $n/2$ in the case of 2-bit Huffman trees (see Figs. 1 and 2). The space overhead for all (internal and leaf) nodes in our 2-bit scheme is less than $1.5*n*(1 + E)*A$ bytes where $0 \leq E < \frac{1}{3}$ is the expansion ratio (see Lemma 3), n is the number of leaf nodes, and A is the size of a decode table entry (2 bytes or less). In many of the 2-bit trees that we have experimented for practical applications, the total space overhead was around 2.2*n bytes.

The optimal approach presented in the previous sections can be easily extended to incorporate variations that are sometimes used in special applications. For example, Huffman compression can be improved by using the concept of m-grams (Reghbati, 1981), that is, frequently occurring sequences of m characters, where $m \geq 2$. Extending the multibit scheme to include m-grams does not change our optimal mapping results nor the decoding approach presented in this study. Using different locality sets and providing a capability to switch between these sets is another example of techniques aiming at improving the compression of special data files (e.g., Bassiouni & Tzannes, 1993; Cormack, 1985; Hazboun, 1982). Rather than using a single (compromise) Huffman tree for all characters of the different fields/records of the input text (Huffman, 1952) or trying to locally adapt this tree based on the encountered text (Bentley et al., 1986; Gallagar, 1978), several Huffman trees are constructed and applied appropriately, one at a time, to these data fields. The optimal algorithm presented in this article can also be adapted, using minor changes, to the case of multiple locality sets. Discussion of these extensions, however, is beyond the scope of this article.

Conclusions

The multibit scheme reduces the time overhead of the bit-serial decoding operation. The case of 2-bit decoding is quite attractive for practical implementation; this study presented an optimal solution for the 2-bit CBS problem. In addition to reducing the processing time of decoding, the optimal algorithm is storage efficient, does not require changes to the encoding process, and is suitable for hardware implementations. Further research is needed to extend the optimal solution to higher order k-bit decoders and to determine the value of k for which a k-bit decoder represents the best trade-off between the speed of decoding and logic (or hardware) complexity.

Acknowledgment

This work has been supported by: DoD/STRICOM (Simulation, Training, and Instrumentation Command) under Contract No. N61339-92-C-0016, by NSF Equipment Grant Nos. CDA-9122006 on software-based compression and CDA-9222798 on hardware-based compression; by a grant from FHTIC (Florida High Technology and Industry Council); and by an in-house grant from UCF-DSR. The views and conclusions herein are those of the authors and do not represent the official policies of the funding agencies or the University of Central Florida.

References

Bassiouni, M., & Tzannes, N. (1993). Integrated lossless/lossy schemes for image compression. *Journal of Optical Engineering, 32,* 1848–1853.

Bassiouni, M. (1985). Data compression in scientific and statistical databases. *IEEE Transactions on Software Engineering, SE-11,* 1047–1058.

Bentley, J., Sleator, D., Tarjan, R., & Wei, V. (1986). A locally adaptive data compression scheme. *Communications of ACM, 29,* 320–330.

Choueka, Y., Klein, S., & Perl, Y. (1985). Efficient variants of Huffman codes in high level languages. *Proceedings of the ACM-SIGIR,* Montreal, pp. 122–130.

Cormack, G. (1985). Data compression in a data base system. *Communications of ACM, 28,* 1336–1342.

Elias, P. (1975). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory, 21,* 194–203.

Fano, R. (1949). *Transmission of information.* Cambridge, MA: MIT Press.

Fraenkel, A., & Klein, S. (1985). *Robust universal complete codes as alternatives to Huffman codes* (Tech. Rep. CS85-16). Department of Applied Mathematics, The Weizmann Institute of Science.

Gallagar, R. (1978). Variations on a theme by Huffman. *IEEE Transactions on Information Theory, IT-24,* 668–674.

Hazboun, K., & Bassiouni, M. (1982). A multigroup technique for data compression. *Proceedings of the ACM SIGMOD Conference on Management of Data,* pp. 284–292.

Hirschberg, D., & Lelewer, D. (1990). Efficient decoding of prefix codes. *Communications of ACM, 33,* 449–459.

Huffman, D. (1952). "A method for the construction of minimum redundancy codes. *Proceedings IRE, 40,* 435–447, 1098–1101.

Lelewer, D., & Hirschberg, D. (1987). Data compression. *ACM Computing Surveys, 19,* 261–296.

Liu, C., & Yu, C. (1991). Data compression using word encoding with Huffman code. *Journal of the American Society for Information Science, 42,* 685–698.

Mukherjee, A., Bheda, H., Bassiouni, M., & Acharya, T. (1991, April). Multibit decoding/encoding of binary codes using memory based architectures. *Proceedings of IEEE Data Compression Conference,* pp. 352–361.

Mukherjee, A., Ranganathan, R., & Bassiouni, M. (1991). Efficient VLSI designs for data transformation of tree-based codes. *IEEE Transactions on Circuits and Systems, 38,* 306–314.

Reghbati, H. (1981). An overview of data compression techniques. *Computer, 14,* 71–75.

Sankoff, D. (1985). Simultaneous solution of the RNA folding, alignment and photo-sequence problems. *SIAM Journal of Applied Mathematics, 45,* 810–825.

Shannon, C., & Weaver, W. (1949). *The mathematical theory of communication.* Champaign, IL: University of Illinois Press.

Tarhio, C., & Ukkonen, E. (1988). A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science, 57,* 131–145.

Welch, T. (1984). A technique for high-performance data compression. *Computer, 17,* 8–19.

Witten, I., Neal, R., & Cleary, J. (1987). Arithmetic coding for data compression. *Communications of ACM, 30,* 520–540.