Combined Error Correcting and Compressing Codes

Extended Summary

Thomas Wenisch
Peter F. Swaszek
Augustus K. Uht[1]
University of Rhode Island, Kingston RI

Submitted to International Symposium on Information Technology 2001 – Washington DC
12 September 2000

1. Introduction

Traditionally, the two encoding operations of compression and error detection/correction are at odds with one another. Compression techniques reduce redundancy in a set of data. Error correction adds redundant information to a data stream so that errors can be detected and recovered from. In the cases where these two encoding operations are combined, they are generally done as two separate encoding steps[1]: first compress the data using, for example, Huffman coding, and then perform an error correction encoding on packets of compressed data, using a technique such as a Hamming coding.

This paper explores the possibility of performing both compression and error correction in a single coding step. This new Error Correcting and Compressing Coding (ECCC) encodes using codewords of varying lengths, but selects codewords such that a minimum Hamming distance between codewords is maintained, enabling error detection and correction.

ECCC codes differ from traditional Huffman codes in their treatment of the prefix property. In ECCC coding, it is not sufficient for codeword prefixes to simply differ, instead, they must be a minimum Hamming distance apart. For the single error correcting codes studied here, this minimum Hamming distance is 3. This minimum Hamming distance between valid codewords allows errors to be detected and corrected.

Thus far, no algorithm or construction has been discovered which finds the optimal ECCC code for a particular input data set as Huffman's algorithm does for non error-correcting codes. However, this paper presents four intermediate results regarding ECCC codes. First, the potential usefulness of such codes is discussed, with a sample code presented. Second, a generalization of the Kraft-McMillen inequality [1] for ECCC codes is presented, which places a lower bound on the existence of an ECCC code for a given set of codeword lengths. Third, the non-optimal techniques which have been successfully used to generate ECCC codes are explored. Of particular interest in this area is a heuristic approach to codeword selection that has performed well against exhaustive trial of all possible codewords. Finally, a construction is presented which demonstrates that many codes for larger alphabet sizes can be constructed by combining codes with smaller alphabets.

2. Motivation for Error Correcting and Compressing Coding

The ECCC coding method provides some potential benefits over a traditional two-step approach. With a Huffman – Hamming approach, a table lookup would be required to encode a symbol in a Huffman code, some memory would be required to accumulate Huffman codewords, and then a second table look up would be required to perform the Hamming coding. With a single code performing both functions, the second table lookup and the memory can be eliminated.

The performance of ECCC is quite good when compared to a Huffman – Hamming encoding, though optimum ECCC codes cannot yet be readily generated. There are two key performance measures that must be

[1] This work was partially supported by NSF Grant No. CCR-9708183, and by the URI International Engineering Program.

examined when evaluating codes which both correct errors and compress. The first measure is the maximum density of errors which can be corrected. ECCC corrects errors at a fixed rate of one bit error per symbol. A Hamming code corrects errors at a fixed rate of one bit error per *m* bits, where *m* depends on which Hamming code is selected. For ECCC codes, the average rate of errors corrected is 1 per *L* bits, where *L* is the average weighted symbol length of the ECCC code. To deduce errors per symbol in a Huffman-Hamming encoding, multiply the weighted average length of the Huffman code by the ratio of total bits to data bits in the Hamming code.

The most convenient measure of compression for comparing ECCC and Huffman-Hamming is the weighted average codeword length after error correction. For ECCC, this is simply the weighted average codeword length. For Huffman-Hamming, this reatio is obtained by multiplying weighted average length by the ratio of total bits to data bits for the Hamming Code.

In the first phase of this research, it was the goal to develop an ECCC code that performs comparably to Huffman-Hamming in both metrics for a real world data set. Since both Huffman coding and Hamming coding are optimal techniques within their respective fields, finding an ECCC code which performs well is not a trivial task. The "real world" data source chosen for this study was the Calgary Text Compression Corpus [2], in particular the obj2 file. Huffman codeword lengths generated based on the symbol frequencies in this file are available online in the appendixes to this summary [3]. The sample ECCC code from which the statistics below were taken is also available. Note that the Huffman code is optimal for this data source, but the ECCC code is not, and more efficient ECCC codes are likely to exist. The following table presents statistics about the two coding methods:

Compression statistics	
Source entropy:	6.26 bits / symbol
Optimal Huffman code:	6.29 bits / symbol
Average length after encoding in a 4-7 Hamming code:	11.01 bits / symbol
Average length after encoding in a 11-15 Hamming code:	8.58 bits / symbol
ECCC code:	10.56 bits / symbol
Error correction statistics	
Huffman – 4-7 Hamming errors per total bits:	1 error / 7 bits
Huffman – 4-7 Hamming errors per data bits:	1 error / 4 bits
Huffman – 11-15 Hamming errors per total bits:	1 error / 15 bits
Huffman – 11-15 Hamming errors per data bits:	1 error / 11 bits
Huffman – 4-7 Hamming errors per symbol:	1.57 errors / symbol
Huffman – 11-15 Hamming errors per symbol:	0.57 errors / symbol
ECCC errors per bits:	1 error / 10.56 bits
ECCC errors per symbol:	1 error / symbol

As can be seen from the table, the performance of the ECCC code exceeds the 4-7 Huffman-Hamming code in terms of compression, but falls short of the performance of the 11-15 Huffman-Hamming. However, it exceeds the rate of error correction of the 11-15 Huffman-Hamming code, while falling short of the error correction rate of the 4-7 Huffman Hamming. ECCC offers a new option of error correction vs. compression tradeoff.

3. Encoder and Decoder designs for ECCC codes

In order for ECCC codes to be useful in practical applications, efficient coding and decoding algorithms are needed. An encoder for ECCC can be trivially implemented using a lookup table. Once such a code table has been generated using some code-generating algorithm, lookups in the code table can be accomplished in O(1) time.

Decoding ECCC codes, however, is not so simple, because of the possibility of bit errors in the received codeword. Many implementations for decoders of variable length codes decode by traversing a code tree. The code

tree is a binary tree where each edge in the tree represents a 0 or 1 at a particular position in a codeword. The leaves of the tree represent codewords. The decoder operates by beginning at root node of the tree, and following either the "0" edge or the "1" edge as each bit is read from the input channel. Once a leaf node is reached, a codeword has been identified and the corresponding symbol is emitted. Then, the decoder begins again at the root node of the tree, and decodes the next symbol.

If bit errors are allowed in the input channel, a simple code tree is no longer enough to correctly identify ECCC codewords. A changed bit will result in a tree traversal that ends up at a "dead end", not landing on any of the leaf nodes that represent codewords.

One solution to this problem is to modify the decoding tree such that all leaf nodes are matched to some symbol. This tree can be constructed by adding leaf nodes for every bit string that differs from a valid codeword by a single bit. Thus, each symbol will be tied to 1 leaf node for itself, and an additional number of leaves equal to its length. Note that, because of the minimum Hamming distance requirements between codewords, these sets of leaf nodes are guaranteed to be disjoint. Any tree traversal ending in a leaf not associated with a symbol indicates at least two bit errors in the channel, and is undecodable. This decoder runs just as quickly as any other tree based decoding algorithm, with the time to decode a symbol proportional to the number of bits in the codeword. The disadvantage of this approach is the storage required for the decoding tree. Such trees become extremely large for even relatively small symbol alphabets.

A second approach to decoding ECCC codewords is to compare input bit strings to the codeword list and search for a match. This decoding algorithm operates by reading in a number of bits from the channel equal to the length of the shortest codeword. Then, the input string is compared to all codewords of this length. If there is an exact match, then the matching codeword has been received and the corresponding symbol can be output. If the input string differs from one of the codewords by a single bit, then a bit error occurred in the channel, and this "near match" is the correct codeword corresponding to the input string. If the input string differs by 2 or more bits from all codewords of this length, then the input string must be for a longer codeword. Additional bits are read from the channel and appended to the input string, and then the comparisons are again performed against the longer codewords. If the decoder compares the input string to the longest codewords and still fails to find a match or "near match", then 2 or more bit errors occurred in the channel. In this situation it is very likely that the decoder will lose synchronism and will be unable to decode the remainder of the input.

A comparison of the input string and a potential codeword is really a calculation of the Hamming Distance between two bit strings, which is most easily accomplished by an exclusive-or operation on the two bit strings, and then totaling the number of 1's in the result. In the general case, N comparisons (where N is the number of symbols) are required to decode a codeword. However, since the decoder can stop comparing as soon as it finds a match or "near match", the average case will require far fewer comparisons. If the symbol alphabet had a flat frequency distribution, the average number of comparisons would be N/2, and when the symbol alphabet is not evenly distributed, even fewer comparisons will be required. This improvement in the average case results because the decoder is always considering the shorter, more likely codewords first.

4. Lower Bound on the Existence of ECCC Codes

It is desirable to have an analytical approach whereby a list of integers representing codeword lengths can be tested to see if they could be the codeword lengths of a valid ECCC code. With an analytical approach, such

integer lists could be quickly tested to see if they represent a possible ECCC code. The output of such a test could then be used to direct and speed up a code generation algorithm, by "showing the algorithm where to look". For traditional (non error correcting) variable length codes, a geometric approach can be used to derive such a bounding condition. If one takes the longest codeword length L in the list l_i, one can build an L dimensional binary space S where each point in S is labeled with a unique binary coordinate string of length L. In order for the codeword lengths in l_i to exist as a valid code, these codeword lengths must "fit" into the space S. Each codeword of length L corresponds to the coordinate of one point in S. Shorter codewords correspond to sets of 2^{L-li} points in S for which that codeword is a prefix. Thus, we can construct the following inequality:

$$\sum_{i=1}^{q} 2^{L-l_i} \leq 2^L \tag{1}$$

where the right side of the equation represents the total number of points available in space S and the left totals up how many points in this space are "occupied" by each of the q members of the list l_i. By manipulating this inequality slightly and replacing 2 with an arbitrary radix r for higher order codes (non-binary), one arrives at the Kraft-McMillen inequality:

$$\sum_{i=1}^{q} r^{-l_i} \leq 1 \tag{2}$$

By applying a similar geometric approach, a bounding equation such as Kraft-McMillen can be found for ECCC codes. Again, one starts by taking the total number of available points, 2^L on the right hand side of an inequality. However, now, in addition to the space directly taken up by each codeword an additional "bounding sphere" surrounds each codeword, the additional points that can be reached by single bit errors. In order to provide error correction, a bounding sphere from one codeword may not overlap the bounding sphere of another. For a desired number of errors corrected E_c the total space taken by each codeword and its bounding sphere is:

$$2^{L-l_i} \sum_{j=0}^{E_c} \binom{l_i}{j} \tag{3}$$

These error bounding spheres include the point in S whose coordinates are prefixed by a particular codeword, and all other points which can be reached with E_c bit errors. The summation term counts the total number of prefixes (the selected codeword itself for $j = 0$, all prefixes with 1 bit different for $j = 1$, 2 for $j = 2$ and so on). Each prefix then occupies a set of 2^{L-li} points in S. Thus, the total number of points in S occupied by a codeword is given by equation 3. Using equation 3, we can now pack these bounding spheres in S, by summing the number of points over the q codewords and ensuring that the total number of points is less than the 2^L points which exist in S. As an inequality, this can be expressed (in a form similar to Kraft-McMillen):

$$\sum_{i=0}^{q} \left(2^{-l_i} \sum_{j=0}^{E_c} \binom{l_i}{j} \right) \leq 1 \tag{4}$$

Unfortunately, this new inequality has a key weakness when compared with Kraft-McMillen which limits its usefulness. While this new inequality does establish a necessary condition for a set of codeword lengths to represent an actual code, this condition is not sufficient. It is clear that it is a necessary condition that there are at least as many points in S as are occupied by the codewords. However, simply because there are enough points for

the bounding spheres to fit *S* does not necessarily imply that these spheres will actually fit. Sphere packing has its own complicated geometry, which this counting approach does not consider.

Numerous examples can be found which meet the requirements of equation 5, but do not represent possible ECCC codeword lengths. For example, the codeword lengths {3,4,5} and $E_c = 1$ result in a value of exactly 1, however, no such code can exist. For any choice of the 3 bit codeword, this leaves only one possible choice for the first 3 bits of the other two codewords. With the first 3 bits of these two codewords identical, a Hamming distance of 3 between them is impossible.

5. Current Techniques for Generating ECCC Codes

In order to obtain a valid ECCC code, it is essential that a generation algorithm ensure that each symbol receive a codeword that is at least Hamming distance 3 from the prefixes of all other codewords. The code generating techniques employed in this study operated by maintaining a list of all codeword prefixes that are currently available for selection, and, upon selecting a codeword from the list, removed all others from the list that did not meet the minimum distance requirement with the selected codeword. This list of possible codeword prefixes is called the *Free Space* (*FS)* list. At the start of a generation algorithm, *FS* contains all eight 3-bit combinations.

As generation proceeds, codewords are selected from *FS,* and *FS* is updated to reflect the new list of possible choices. Codeword selections which would completely exhaust the *FS* list are not allowed. If no choice from *FS* satisfying this condition is possible, *FS* is *grown* by lengthening the prefixes listed in *FS*. Each existing member is replaced with two new members, one appending a 0, the other a 1. Thus, the total size of *FS* doubles. This grow operation can be repeated until *FS* is sufficiently large that a valid choice exists.

If the generation technique just described is applied, the *comma code* will be generated. In this code each codeword is 3 bits longer than the previous one; save the last two codewords, which are of equal length. This code represents one extreme of the ECCC codes that are efficient for some source probability distribution.

In order to generate other codes, grow operations have to be performed "early" – that is, they must be performed at times when *FS* isn't about to be exhausted. This is called *biasing*. By biasing a codeword, its length is sacrificed – is made longer – in order to make further codewords shorter. If the very first codeword is sufficiently biased, the other extreme case, the *balanced code*, can be generated. In the balanced code, all codewords are of equal length. The two extreme cases are easy to generate, but do not perform well for typical data sets. Optimum codes for a particular data set lie somewhere between the comma code and the balanced code.

For small alphabet sizes, the space between the comma code and balanced code can be exhaustively searched, by in turn trying all possible choices from *FS* at each codeword selection, and all possible combinations of biasing between codeword selections. Exhaustive results have been obtained for alphabet sizes up to 9 symbols. Empirically, the search time for these exhaustive searches increases 12 fold with each increase of 1 in the alphabet size. This growth rate results in unacceptable performance for large symbol alphabets.

For codeword selection, a heuristic has been developed to replace the exhaustive trial and error of all possible *FS* choices. This heuristic performs remarkably well – using it, all codes for alphabet sizes up to 8 that are found by the exhaustive search can be generated, and for 9 symbols, only 2 of 65 possible codes are missed.

The heuristic operates by first selecting the subset of the *FS* list for which a maximal number of *FS* members will remain following codeword selection. That is to say, the subset of *FS* for which each member is distance 3 from the most other members of *FS* (there will almost invariably be several codewords which are each

distance 3 from an equal number of other codewords). Then, for each member of this subset, the average distance between the "leftover" *FS* members is calculated. The leftover *FS* members are those *FS* members that are distance 3 from the codeword being considered. The codeword for which this metric is highest is then selected and assigned.

This heuristic selection algorithm runs in $O(n^2)$ time with respect to the number of members in the *FS* list, as each member of *FS* needs to be compared to every other at least once to determine their Hamming distances. In the worst case, if all codewords leave an equal number of "leftover" *FS* members, n average distance calculations of n elements each are required. However, despite these high costs, the heuristic runs orders of magnitude faster than a truly exhaustive search of the code space.

A similar heuristic for when and how to bias code generation has not yet been developed. The codes used in preparing this paper were generated using trial-and-error of a variety of biasing schemes, with the goal of incrementally improving the performance of a generated code by slightly modifying the biasing scheme, and in this way zero in on an optimum code.

6. Constructing ECCC Codes

After examining the output of the exhaustive search algorithm described above, it can quickly be seen that many patterns exist in the codes that are discovered. Indeed, many of the codes can be "built" by combining two codes for lower numbers of symbols. Consider the 3-7-8-8 code. This code is as follows, as it is discovered by the exhaustive search algorithm. A quick check will show that each codeword is at least distance 3 from every other. However, if a line is drawn after the third bit in this code, then two other codes can be seen.

A	000	
B	111	0000
C	111	01110
D	111	10111

To the right of the line, the suffixes of codewords B, C, and D form a 4-5-5 code. To the left of the line, if the codewords B, C, and D are collapsed into a single codeword, a 3-3 code is formed. Using these two codes, the 3-7-8-8 code can be constructed as follows:

Given two valid ECCC codes C_1 and C_2 with symbol counts of m and n, respectively:

1. Take the last codeword in C_1 and duplicate it n-1 times.
2. Then, to each occurrence of this codeword, append a codeword from C_2.
3. The result will be a valid ECCC code with $m + n - 1$ symbols.

Codes that can be constructed from other codes can now be identified using an addition operation. For example, the 3-7-8-8 code discussed previously is 3-3 + 4-5-5. For N=9, of the 65 codes which exist, only 6 cannot be described using this addition operation of two smaller codes. These codes, which cannot be constructed by this method are called *basis* codes. Even in the basis codes, there are patterns in the way bit strings appear, however, a simple construction such as the one described here cannot generate these codes.

7. Conclusions

This paper has shown that codes can be developed which provide error correction and compression capabilities in a single coding step. However, the geometry of these codes, due to the error correcting "spheres" that must surround each codeword, makes these codes very difficult to generate directly from a set of symbol probabilities, as can be done for regular Huffman codes. This paper discusses a general approach to generating ECCC codes including an exhaustive approach, and heuristic methods which sacrifice accuracy for enormous time gains. This paper also shows how ECCC codes are highly patterned, and many codes can be generated from smaller

codes using a simple construction. It is hoped that with further study, a method for directly generating the ideal ECCC code for a particular set of symbol probabilities can be found.

8. References

[1] R. W. Hamming, *Coding and Information Theory*. Englewoods, NJ: Prentice Hall 1980.

[2] T. C. Bell, J. G. Cleary, I. H. Witten, *Text Compression*. Englewoods, NJ: Prentice Hall 1990.

[3] Available online at www.ele.uri.edu/~iota/eccc.html