Preprint
UCRL-JC-150989

Merged Source Word Codes for Efficient, High-Speed Entropy Coding

J. Senecal, K. Joy, and M. Duchaineau

This article was submitted to
Data Compression Conference, Snowbird, UT, March 25-27, 2003

December 5, 2002

Merged Source Word Codes for Efficient, High–Speed Entropy Coding

Joshua Senecal Ken Joy
Center for Image Processing and Integrated Computing
University of California, Davis
{*jgsenecal,kijoy*}*@ucdavis.edu*

Mark Duchaineau
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
duchaine@llnl.gov

Abstract

We present our work on fast entropy coders for binary messages utilizing only bit shifts and table lookups. To minimize code table size we limit our code lengths with a novel type of variable–to–variable (VV) length code created from source word merging. We refer to these codes as *merged codes*. With merged codes it is possible to achieve a desired level of efficiency by adjusting the number of bits read from the source at each step. The most efficient merged codes yield a coder with an inefficiency of 0.4%, relative to the Shannon entropy, in the worst case. On one of our test systems a current implementation of coder using merged codes has a throughput of 35 Mbytes/sec.

1 Introduction

With the rapid progression of computing technology high–performance computing is becoming more available. More complex problems are being examined at greater levels of detail, and these computations are creating greater amounts of data that must be stored for analysis. For example, a simulation of a Richtmyer–Meshkov instability and turbulent mixing was performed at a resolution of 2048 x 2048 x 1920 on the IBM Sustained Stewardship TeraOp system located at Lawrence Livermore National Laboratory [7]. The simulation run generated over three terabytes of data which needed to be compressed and stored. Instead of tying up valuable disk space by temporarily storing data which will be later compressed, we would like to add to simulation codes the ability to encode data on the fly, before it is written to disk. A coder used in this way must be fast, so as to not degrade the overall speed of the simulation.

We present here our work on developing a fast entropy coder that performs only bit shifts and table lookups during the coding process. To place a limit on the sizes of our code tables, our coder uses merged source word codes (hereafter referred to as "merged codes"), which are a type of variable–to–variable length (VV) code. Each code has a source window of W bits, reading up to W bits per coding iteration, and outputs a codeword that is up to C bits in length. When referring to a merged code created with a specific (W, C) we call it a (W, C) merged code.

We treat all data sources as binary and memoryless. Each bit b_i in a source has a probability of being either a 0 or a 1, denoted respectively by the pair (p_i, q_i). We assume without loss of generality that the more probable symbol is 0, that is, $q \leq 0.50$. For a binary memoryless source with probabilities (p, q) the entropy H of the source is defined as [9]:

$$H(p, q) = -(p \log_2 p + q \log_2 q) \tag{1}$$

From this we define the *coding inefficiency* I of a coder K at (p, q) as

$$I_K(p, q) = \frac{R_K(p, q) - H(p, q)}{H(p, q)} \times 100 \tag{2}$$

where $R_K(p, q)$ is the coding rate of coder K at (p, q). In a memoryless source, a word of some length l has a weight $w = p^z q^y$, where z is the number of zeros in the word and y is the number of ones. If there are s possible words in a source, $R_K(p, q)$ for a coder is computed empirically as

$$R_K(p, q) = \frac{bitsout}{bitsin} \tag{3}$$

and theoretically as

$$R_K(p, q) = \frac{\sum_{i=1}^{s} c_i w_i}{\sum_{i=1}^{s} l_i w_i} \tag{4}$$

where c_i is the length of the prefix–free codeword assigned to source word i. *Theoretical inefficiency* refers to I computed with a theoretical R, while *empirical inefficiency* refers to I computed with an empirical R. Our goal is to create a coder that has a theoretical $I \leq 1\%$ when coding a given source. We call a coder K inefficient if $I_K > 1\%$. We later show that a coder using the most efficienct of the merged codes created in our study would have a theoretical inefficiency of less than 0.4%.

2 Prior Work

Given a binary memoryless source a common way of encoding it is through run–length encoding, usually the more efficient Golomb code [4]. A plot of the Golomb Codes' inefficiency in Figure 1 shows that while the codes are mostly efficient, there are regions where the inefficiency is well above 1%. A more efficient method is to use Block–to–Variable (BV) Huffman coding [5], in which W bits are read at a time and encoded. A plot of the inefficiency of 10-bit block Huffman coding in Figure 2 shows that BV Huffman coding can perform much better in the regions where Golomb coding is inefficient. BV coding creates a problem, however. When using table lookups for encoding and decoding, the number of entries in the tables must be a power of two of the block and longest code lengths, respectively. If q is small enough the longest code length may force a decoding table size that is larger than desired. To enforce a maximum allowable decode table size we need to limit the code lengths. Rather than use previous methods of limiting code lengths that adjust code

lengths at the expense of efficiency [12] we choose to limit code lengths by shortening those source words that are least likely. Source word merging (described in section 3) performs this shortening, taking a BV Huffman code and transforming it into a VV code.

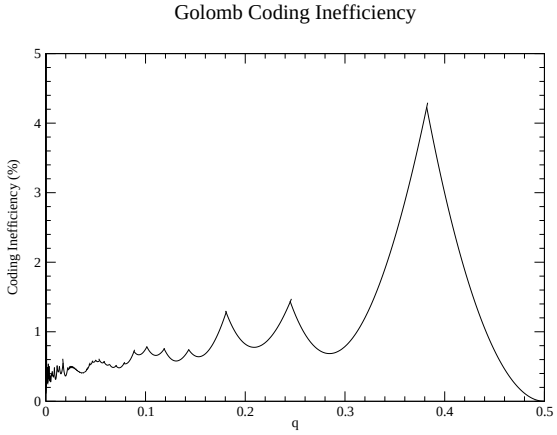

Figure 1: Inefficiency of Golomb Codes.

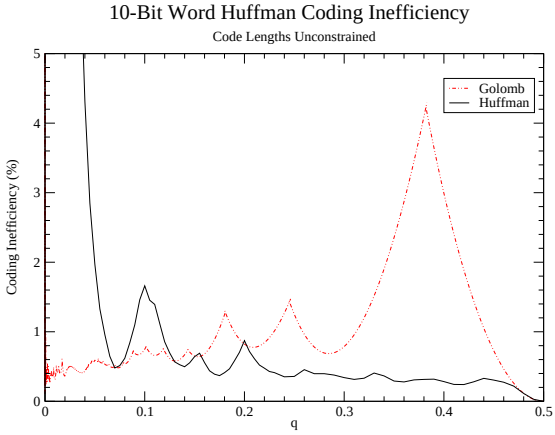

Figure 2: Inefficiency of Huffman Code.

VV codes are typically described in the literature as being created through an *extension* process similar to that of the Tunstall variable–to–block (VB) algorithm (reviewed in [1]). For a source with a given alphabet of β symbols the extension process is to first create a parse tree with β leaves, where each leaf s_i is at depth 1 and has probability $P(s_i)$. Then, pick a leaf s on the tree according to some criteria and extend it, turning it into an internal node with β children (all leaves) with probabilities $P(s)P(s_1)$, $P(s)P(s_2)$, ..., $P(s)P(s_\beta)$. The process repeats until some termination criteria are met. Prefix–free codes are generated for the source words, and the resulting word–code pairs are used to encode the source. Most work in VV code algorithms focuses on selecting a leaf for extension such that the resulting parse tree is optimal.

VV codes generally have not received a lot of attention. The reason for this is perhaps that the VV codes are difficult to analyze, and there are as yet no known algorithmic methods for creating optimal codes. Currently the only way to find an optimal VV code is through an exhaustive search. Fabris [2], Stubley [11], and Freeman [3] studied this problem, and although they developed methods that can be used to find codes that are near optimal the conclusion in the literature appears to be that a finding an efficient algorithm that generates optimal codes is an unsolved problem. Part of the difficulty is that the information mismatch (*divergence*) between a source parse tree and its associated code fluctuates with each leaf extension [3], and that even given an optimal tree the next larger optimal tree cannot always be obtained by extension [10].

3 VV Codes from Source Word Merging

Merged codes provide a way to limit the encoding and decoding table sizes. Source word merging stems from the observation that if we read a binary source W bits at a time then a parse tree for a binary bitstream has 2^W leaves ($W \geq 1$) and is *complete*—the parse tree will parse any bitstream (we assume the end of the bitstream is padded if needed). From this it is simple to observe that *any* parse tree with no null branches is also complete. Given a complete parse tree, by repeatedly merging the least probable leaf together with it's sibling into their parent we maintain the tree's completeness while reducing the number of source words, and thereby shorten the code lengths.

In essence the merging process transforms a BV code into a VV code. Given (p, q) and (W, C) the merging algorithm is shown in figure 3. Table 1 gives the effective number of source words for $(10, C)$ merged codes, created for various values of C and q.

Generate all 2^W source words
Compute each source word's weight
Compute a code length c_i for each word
while ($\exists\ c > C$) {
    select word with longest code length c
    merge with sibling into parent
    compute weight of new, shorter word
    reassign code lengths to all words
}
Generate prefix–free codes

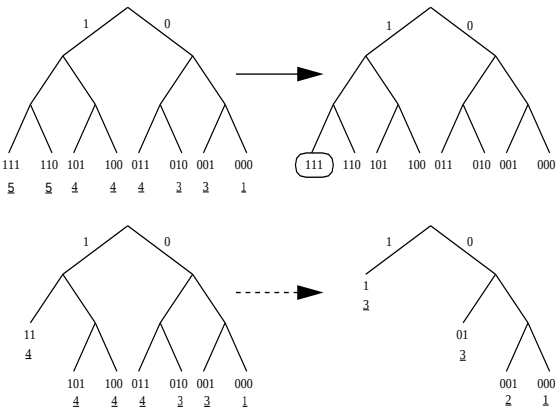

Figure 3: Example of merging leaves on the source parse tree. Underlined numbers are code lengths.

	Code Length Constraint			
q	7	8	9	10
0.12	56	57	68	176
0.20	60	89	144	192
0.30	84	155	263	444
0.40	90	224	367	731

Table 1: Effective number of source words for some $(10, C)$ merged codes. Original number of words was 1024.

4 Theoretical Inefficiency of Merged VV Codes

We examined merged VV codes created with an input bit window size W, $7 \leq W \leq 15$, and an output code length limit C, $7 \leq C \leq 13$. For each value of q, at a

granularity of 0.01 we created coding table with the parameters (W, C) and measured the theoretical and empirical inefficiency. Figures 4 and 5 show plots of the empirical inefficiency of merged codes where the W equals 10 and 15 bits, and $C = 7, 10, 11$.

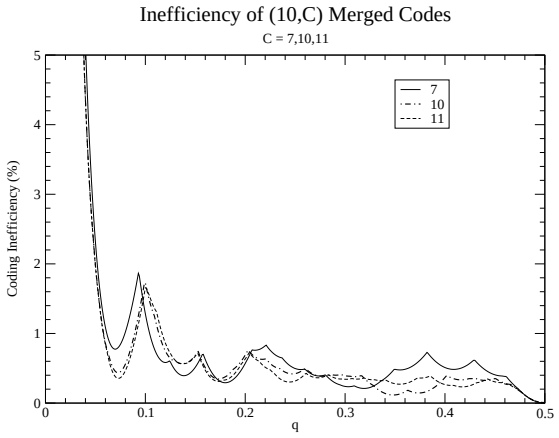

Figure 4: I of (10,C) merged codes.

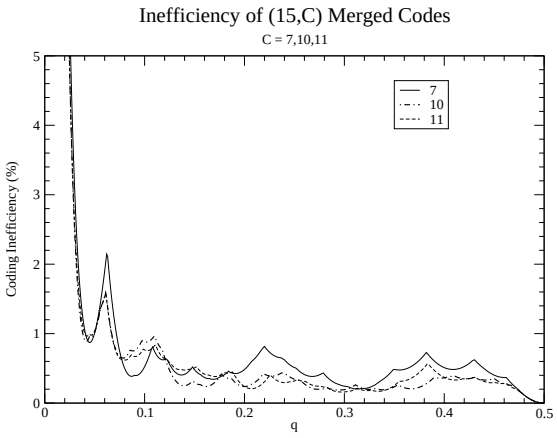

Figure 5: I of (15,C) merged codes.

Our observation is that as W increases the inefficiency of the resulting codes tends to be less, and the point where the code becomes inefficient moves to the left. We define the *point of inefficiency*, as q tends to zero, as the value of q where the inefficiency jumps above the 1 percent mark and does not return below it. For a given W the resulting codes for each C are as expected from previous research: the inefficiency of the codes varies almost unpredictably and there are regions where one C is more efficient than another. For the merged codes not pictured here the observations were similar. We conclude from this that in general if a coder is to use one and only one merged code, and even a small search for a most optimal code is infeasible, then the code should be created with the largest W possible. Given W and q we select the value of C that yields the least inefficient code and satisfies any requirements to limit the size of the entries in the coding tables.

Figure 6 shows the correlation between the theoretical and empirical inefficiency of $(10, 10)$ merged codes (results were similar for other codes studied). The overall theoretical curve was obtained by selecting for each value of q_i at a granularity of 0.001 the $(10, 10)$ merged code with the least inefficiency at that q_i. Empirical values were obtained by creating a $(10, 10)$ merged code for each value of q_i at a granularity of 0.01, and encoding a memoryless bitstream with $q = q_i$. There are some places on the plot where the empirical value does not match the expected theoretical value. This is because the empirical values were recorded with the assumption that that the $(10, 10)$ merged code created for a certain q_1 would be most optimal at that q_1. This assumption is flawed, as there are circumstances when a $(10, 10)$ merged code created for a nearby q_2 is more efficient at q_1 than the table created for q_1.

Figure 7 shows the inefficiency curve that results if for each value of q the most optimal code is selected from among those created in our study. The reduction in inefficiency is dramatic. Excluding the values where the curve rises to the point of inefficiency a coder using these codes would have a worst-case inefficiency of 0.4%.

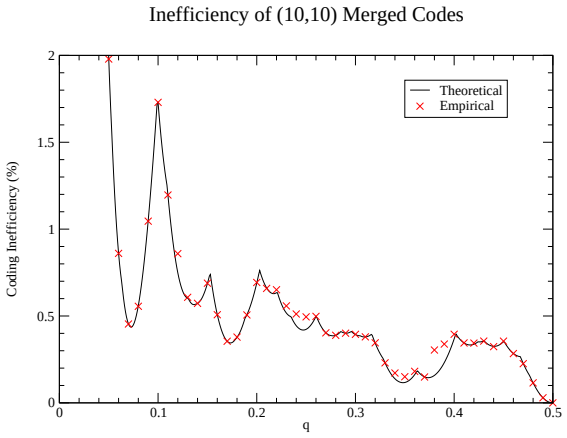

Figure 6: I of (10,10) merged codes.

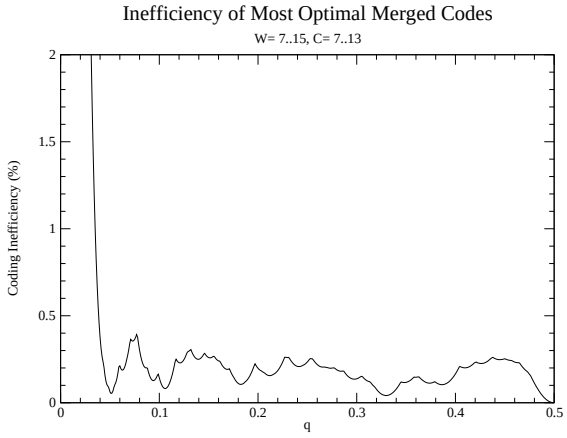

Figure 7: I of most optimal codes.

5 Coding Tables and a Simple Coder

An encoding table entry is composed of three fields: the codeword, the codeword length, and the source word length. For speed the sizes of the table entries used are the same as the standard word sizes: 16 or 32 bits. For simplicity the field widths are fixed. For the 16–bit entry the length fields are 3 bits wide, and the codeword field is 10 bits. For the 32–bit entry the length fields are 5 bits wide, and the codeword field is 22 bits wide. The fixed field widths limit the number of different codes that can be created. For example, an $(8, 8)$ merged code could use 16–bit table entries, whereas an $(11, 8)$ merged code could not. However, a *bias* may be applied to one of the field width values to make a table entry size useable when it otherwise would not be. For example, for a given q if an $(11, 8)$ merged code is computed and the resulting source word lengths have a range of 8 or less, say the longest word is 10 and the shortest is 3, then a bias of 3 may be subtracted from all source word lengths before they are written to the encoding table. When the actual encoding is performed a bias of 3 is added to each source word length that is obtained from the coding table.

The coder has an input and output buffer. It also has two registers, the *source* register and the *coded* register. Each register is 64 bits wide. Bits to be encoded are taken from the source register, and encoded bits are placed on the coded register. When the number of available source bits in the source register falls to 32 or less an additional 32 bits are transferred from the input buffer. When the coded register contains 32 or more coded bits 32 bits are transferred to the output buffer.

Given a (W, C) merged code the coding algorithm is:

$wordlen = 0$;
$codelen = 0$;
$numSrcBits = 64$;
$numCdeBits = 0$;
Load 64 source bits into the source register;

while (source data remains)

Use the next W bits in the source register to get a value v;
Consult CodeTable[v]
    Get *wordlen*, *codelen*, and *codeword*;

Shift coded register *codelen* bits;
Write *codeword* to coded register;
$numCdeBits = numCdeBits + codelen$;
Shift source register to remove *wordlen* bits;
$numSrcbits = numSrcbits - wordlen$;

if ($numSrcBits \leq 32$)
    transfer 32 source bits from the input buffer;
    $numSrcBits = numSrcBits + 32$;
if ($numCdeBits \geq 32$)
    transfer 32 coded bits to the output buffer;
    $numCdeBits = numCdeBits - 32$;
end

To test the speed of the coder we encoded bitstreams that were random but weighted so that a desired (p, q) was obtained. We then encoded the bitstream with a merged code for that (p, q).

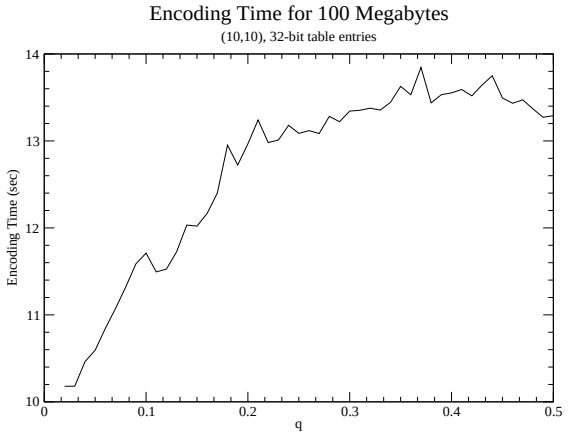

Figure 8: Encoding time vs. q for 100 megabyte data.

Figure 8 shows a diagram of the encoding time for 100–megabyte bitstreams, as q goes from 0.01 to 0.50. The test system was an SGI O2 with a 200 MHz MIPS R5000 processor with a 32 kbyte data cache. The encoding table entries were 32 bits wide. As q approaches 0.50 more and more table entries are touched and must be loaded into the cache. The portions of the bitstream being worked on are also loaded into cache, and so the performance of the coder degrades as table entries and bitstream data compete for cache space. As q tends to 0 fewer table entries are actually used, and so there is more data in cache and less loading of table entries. At some point the majority of the table entries used are those that are powers of 2. For a 1024–entry table this means that only 10 entries are likely to be used. This has good implications:

for smaller values of q a larger W can be used without causing a major detriment to performance, although the memory allocated for tables grows.

6 A Bin Coder for Wavelet Bitplanes

As a demonstration of the speed of merged codes, we present here a bin coder used for losslessly encoding the bitplanes of coefficients resulting from the wavelet transform of an 8–bit grayscale image. A *bin* is a simple coder that uses a code that is most ideal for a certain (p, q). Examples of bin coders are presented in [6] and [8]. Our encoder is a practical example of how merged codes may be applied to a specific purpose. As this bin coder is not more general–purpose, the amount of overhead required to encode the wavelet coefficients is minimal.

As each wavelet coefficient is 9 bits wide (8 bit magnitude and 1 sign bit) our coder creates and uses 9 bins. After a wavelet transform the resulting coefficients are passed to the coder. The coder separates the coefficients into bitplanes, and for a given bitplane i determines (p_i, q_i) by means of a simple scan, treating it as a memoryless source. Once (p_i, q_i) has been determined, bin i loads the corresponding merged code encoding table, and uses that table to encode bitplane i. If (p_i, q_i) is such that a merged code is too inefficient then a more efficient Golomb code is used instead. Also if (p_i, q_i) is close to 0.50 then no encoding takes place, and the bitplane is placed into a "dummy" bin and is held until all other bins have completed encoding.

To test the speed of this coder we transformed several images and encoded the resulting wavelet coefficients. Images b500 and b834 came from the data generated in [7], and y4.095 and z7.125 from a simulation of crack propagation [1]. Testing was performed on several machines: an SGI O2 with a 200 MHz MIPS R5000 processor, one 250 MHz MIPS R10000 processor on a 45–processor SGI Origin 2000, and a PC with a 2.53 GHz Pentium 4 with 1Gbyte 1066MHz RDRAM memory, running RedHat 6.1. Throughput measurements include only in-memory operations; there was no disk I/O performed. Results from the tests are given in table 2. Table 3 shows how many of each type of bin was created for each image.

	Size		Throughput (Mbyte/sec)		
Image	Before	After	SGI O2	SGI Origin 2000	Linux PC
b500	2.345	0.837	5.852	12.197	37.475
b834	2.379	1.432	6.291	13.517	38.319
y4.095	4.219	0.768	7.533	15.593	48.720
z7.125	4.500	2.102	6.180	13.132	38.628

Table 2: Information about coding speed. Sizes are in Megabytes.

As an example of individual bin speeds, when when using a (15,10) merged code with a source $q = 0.274$ a throughput of 29 megabytes/sec was observed. When using a (10,10) code with a source $q = 374$ a throughput of 35 megabytes/sec was observed.

[1] http://www.llnl.gov/largevis/atoms/brittle-fracture/

Image	Golomb	Merged VV	Dummy
b500	4	5	0
b834	2	7	0
y4.095	6	3	0
z7.125	3	6	0

Table 3: Bins created for each image.

7 Future Work

For the future we plan to focus on developing more general coders that use a modest number of merged codes and have a theoretical inefficiency of less than 1%. We would like to use merged codes to develop a more general implementation of a bin coder, akin to the coders presented in [6] and [8]. In this type of bin coder a finite number of bins are placed so that together they cover the entire range $0 < q < 0.50$, with no bin overlapping another. Bits from a binary source are read and placed into the appropriate bins based on the bits' respective (p, q). Bin coders have some nice advantages. Overall coding rate can be adjusted by changing the number of bins used, and/or by altering the probability modeling scheme that determines the probability of each source bit. The only adaptivity in the coder is in updating the probability model. There is no overhead for updating or altering code trees.

References

[1] Johann A. Briffa. Investigation of the error performance of tunstall coding. B.Eng.(Hons.) Final Year Dissertation, University of Malta, Faculty of Engineering, 1997.

[2] Francesco Fabris. Variable–length–to–variable–length source coding: A greedy step–by–step algorithm. *IEEE Transactions on Information Theory*, 38(5):1609–1617, September 1992.

[3] G.H. Freeman. Divergence and the construction of variable-to-variable-length lossless codes by source-word extensions. In *DCC '93: Data Compression Conference*, pages 79–88, March 1993.

[4] S. W. Golomb. Run-length encodings. *IEEE Trans. Inform. Theory*, IT-12:399–401, 1966.

[5] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952.

[6] A. Kiely and M. Klimesh. A new entropy coding technique for data compression. Technical Report 42-146, Jet Propulsion Laboratory-CIT, August 2001.

[7] Mirin, Cohen, Curtis, Dannevik, Dimits, Duchaineau, Eliason, Schikore, Anserson, Porter, Woodward, Shieh, and White. Very high resolution simulation of

compressible turbulence on the ibm–sp system. Technical Report UCRL–JC–134237, Lawrence Livermore National Laboratory, 1999.

[8] F. Ono, S. Kino, M. Yoshida, and T. Kimura. Bi-level image coding with melcode - comparison of block type code and arithmetic type code. In *GLOBECOM '89. IEEE Global Telecommunications Conference and Exhibition. Communications Technology for the 1990s and Beyond*, pages 255–260, November 1989.

[9] Claude Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, July 1948.

[10] Peter R. Stubley. *Design and Analysis of Dual–Tree Entropy Codes*. PhD thesis, University of Waterloo, 1992.

[11] Peter R. Stubley. Adaptive variable-to-variable length codes. In *DCC '94: Data Compression Conference*, pages 98–105, March 1994.

[12] Andrew Turpin and Alistair Moffat. Practical length-limited coding for large alphabets. In *Eighteenth Australasian Computer Science Conference*, volume 17, pages 523–532, 1995.

This work was performed for DOE by UC, LLNL under contract no. W-7405-Eng-48.