New Techniques in Context Modeling and Arithmetic Encoding

by Charles Bloom (cbloom@mail.utexas.edu)

Introduction

Finite context markov modeling, and in particular the PPM method, is known to be one of the most effective methods for achieving high compression levels [3]. The PPM method is also very useful in hybrid combinations with other compressors [14,15]. The PPM method, however, requires abundant memory and processing power. This paper presents four new techniques which address the various problems of context coding.

1. **PPMCB :** A recent development is the discovery that only the most recent occurrence of each context need be remembered to achieve a high level of compression. This results in an extremely simple and memory-use conservative data structure, with O(1) computational complexity. The PPMCB heuristic takes advantage of these properties to produce one of the fastest context coders in existance.
2. **Order1-ListLRU :** Unfortunately, the PPMCB method only applies well to order-2 and higher models : order-1-0 must still be modeled with the PPM method. Order-1-0 context coding is also important because it is an excellent solution for compressing literals generated in LZSS parsing [16]. The result of this combination is LZAri, one of the most effective compressors in terms of compression per unit time taken [15]. Lelewer and Hirschberg have previously reported on ways to reduce the memory use of order-1-0 PPM to 11k bytes, while simultaneously speeding up execution [8]. A new method is revealed herein which uses an LRU cycle of only a few contexts, uses less memory (7K), and compresses better than previous methods. This method is found to even out-compress exact order-1 PPM.
3. **Order1-StaticHuff :** Another problem with the PPM method is its use of an arithmetic coder. Arithmetic coders, while appealing for their optimality, are quite slow. It is the complexity of arithmetic coding which makes PPM unsuitable to high-speed applications. A method is presented which permits the use of a Static (two pass) Huffman coder in order-1-0 context coding [12]. This coder gets only 5% less compression than order-1 arithmetic coding and operates at speeds which are five times faster.
4. **Deferred Summation :** Finally, a new method of performing extremely fast approximate order-0 adaptive arithmetic encoding is presented, using the "Deferred-Summation" model,

byte-aligned output, approximate renormalization, and no division (without approximation).

Section 1 : A high order context modeling method : PPMCB

The PPMCB algorithm can be thought of as a PPM implementation which keeps only the most recent character in each context. Thus, each context need have only the following information:

predicted character identity (`PredChar`)
`PredChar` count (`MatchCount`)
escape count (`NoMatchCount`)

requiring a total of 3 bytes of memory per context.

Context lookups are done using a hash table. A hash function h = HF(c) , where c is the context, and h is the hash index, is used to convert the value of the order-n context c_n into the hash index h_n, which has bounds $0 <= h_n <= H_n$, where H_n is the size of the order-n hash table. The hash function this author favors is:

$$h = ((c >> k) \wedge c) \& (H - 1)$$

where k is a constant which ensures that all (or most) of the bits in c have some effect on the value of h; for example, for n = 4 (an order-4 context, which makes c a 32 bit value) and H = 65536, the best value of k has been found to be k = 11. This hash function creates a high degree of randomness, and is also very easy to compute; it works better for this application than the popular modulo-division hash functions [10].

Two different collision avoidance methods were tested : no collision avoidance, and "context confirmation". The context confirmation method keeps the actual value of c_n in a table of H_n members. When the hash index is used to look up the `PredChar` and the counts, the actual value of c_n for which that information was generated is also looked up. If the stored value is not equal to the current value, the model escapes to the next lower order and the new value of c_n is inserted into the table - without writing an escape, because the decoder will also know to escape. Otherwise, the normal PPMCB procedure occurs.

The normal PPMCB procedure is to test if the current character matches `PredChar`. If so, the `PredChar` is encoded, and `MatchCount` is incremented. Otherwise, the escape is encoded, the model drops to the next lower order context, and `NoMatchCount` is incremented.

There are three very significant properties of PPMCB :

1. Nothing but binary arithmetic coding is done, which means that a very fast coder could be used, such as the QM coder used in JPEG arithmetic encoding [17].

2. Because only the most recent occurrence of each context is kept, the context-modeling routines have O(1) complexity.

3. Finally, another consequence of only keeping the most recent occurrence of each context is that unbounded-length context modeling is very fast. In fact, the same data structures used for LZSS string matching may be used for context-matching : only a file position need be retrieved (all forward structures are replaced by analogous backward structures) [18]. The `PredChar` is then the character immediately following the context's file position. The `PredChar` and escape counts can be kept in a table indexed by the length of the context.

This method performs exceptionally well for high orders. The order-3 hash table may even be reduced to a mere 4096 members without significant compression performance reduction (only a 2% improvement is gained by using a 65536 member hash table) resulting in an order-3 context model which requires only 12k bytes of memory!! Compression is significantly improved by using context confirmation.

For higher compression, another variable may be kept in each context : `NumConsecutiveMatches`. Each time a match is made, this is incremented. Each time a no-match is made, this is set to zero. When a no-match is made, the currently stored `PredChar` is not replaced if this count is greater than two.

Section 1.1 : A high performance implementation

This uses exact order-5 modeling, and an order-2-1-0 PPM model with full exclusion. For contexts with low counts (`MatchCount` and `NoMatchCount` less than 16), the actual counts are not used for coding the event; instead, the actual number of matches and nomatches which have occurred in other contexts with the same `MatchCount` and `NoMatchCount` is used (in the style of LZP4 [14]). This implementation is still much faster, and uses much less memory than standard PPM methods. The performance of this order-5 compressor on the Calgary Corpus is:

	PPMCB	PPMC
bib	1.946	2.11
book1	2.423	2.48
book2	2.150	2.26
geo	4.466	4.78
news	2.487	2.65
obj1	3.722	3.76
obj2	2.563	2.69
paper1	2.416	2.48
paper2	2.391	2.45
pic	0.792	1.09
progc	2.449	2.49
progl	1.753	1.90
progp	1.752	1.84
trans	1.520	1.77
average	**2.345**	**2.48**

Figure 1 : high compression PPMCB performance

Section 2 : Memory-Efficient Order-1 Coding

PPMCB solves the problems of high memory use and low speed of high-order context modeling, but it does not address lower (1 and 0) order modeling. All of the order-1 models presented here may also be used for order-2 modeling. This is facilitated by hashing the order-2 context down to 12 bits, and using context confirmation (as described for PPMCB). Note that PPMCB is faster and uses less memory than these methods, but these methods achieve better compression.

Several order-1-0 PPM implementations were explored in detail. All implementations use a full order-0 model, which was implemented using a Fenwick tree [17], and PPMC escape probability estimation for the order-1 to order-0 transition [9]. The order-1 models explored were:

1. **Order1-Raw :** a full order-1 context model, using 256 Fenwick trees [17].
2. **Order1-FixedList :** the model described by Lelewer and Hirschberg [8]. 256 lists are kept, one for each order-1 context. Each list is a list of N characters, N counts, and an escape count. N = 20 was found to be optimal by Lelewer and Hirschberg.
3. **Order1-ListLRU :** a new model, similar to Order1-FixedList, except that only 100 lists are kept, each of length 28. Additionally, 256 pointers to lists are kept. Each time a context is seen, the list pointer is examined. If the list pointer is not null, the list pointed to is used to encode the next character, it is updated, and it is moved to the front of an LRU of lists. If the list pointer is null, the last list on the end of the LRU is removed from the LRU, cleared, and assigned to the new context. The list pointer for the context which was previously using this list is set to null. The retrieved list is then put on the front of

the LRU.

4. **Order1-ListLRU2 :** this is the same as Order1-ListLRU except that it keeps two LRUs : one of 80 lists, each of length 24, and another of 20 lists, each of length 48. The short lists cycle just as they do in Order1-ListLRU. Whenever a short list becomes full, a long list is assigned to that context, using the same procedure as for assigning short lists to empty contexts. The short list which was at that context is assigned to the context which the long list came from.

The motivation for Order1-ListLRU is that only a few contexts will actually be seen, so it is unnecessary to keep information for all contexts. Order1-ListLRU2 is motivated by the observation that some contexts tend to occur very often (such as the ’ ’ (space) context in text files) and these contexts need longer histories than others.

The performance of these order-1 models is reported below, in terms of output bits per input byte, on the files of the Calgary Corpus [19]. Memory use is reported, in bytes, not including the memory used by the order-0 model.

Order-1 Models

	Raw	FixedList	ListLRU	ListLRU2	
mem use	131583	11776	6968	7128	(bytes)
bib	3.449	3.524	3.506	3.491	
book1	3.597	3.671	3.645	3.642	
book2	3.769	3.782	3.747	3.738	
geo	4.729	4.725	5.002	5.000	
news	4.151	4.213	4.163	4.172	
obj1	4.560	4.502	4.320	4.325	
obj2	4.106	4.014	3.772	3.768	
paper1	3.817	3.847	3.794	3.781	
paper2	3.619	3.658	3.628	3.621	
pic	0.835	0.864	0.858	0.857	
progc	3.845	3.878	3.831	3.815	
progl	3.317	3.245	3.223	3.213	
progp	3.357	3.368	3.336	3.330	
trans	3.484	3.479	3.427	3.411	
average	**3.617**	**3.626**	**3.589**	**3.583**	

Figure 2 : compression of order-1 models

The surprising and delightful result is that the Order1-ListLRU structures are both extremely memory-use efficient, and (unlike the FixedList structure of Lelewer and Hirschberg) they compress even better than the Order1-Raw model. The author presumes that this is because the LRU heuristic is effective at modeling the local-skew of the data. The added complexity of ListLRU2 over ListLRU does not justify its use, except when maximum compression is desired.

Section 3 : Order-1 Static Huffman

This method uses Order-0 Static (two pass) Huffman encoding on several arrays.

Order-1 Static Huffman may be used in the same situations as Order-0 Static Huffman : when all of the data to be encoded is available at transmission time. This means it is not suited to adaptive ("one-pass" or "on-line") compression.

First, a description of an order-0 static Huffman coder is needed. When the ’client’ wishes to encode a character, it is simply added to an array (Array0). After all characters have been added, the client tells the encoder to transmit the data. At this point, the character frequencies are counted, and the Huffman code lengths are computed. Then, these code lengths are transmitted (in a compressed form). Lastly, an alphabetically-ordered code is generated using the Huffman code lengths, and Array0 is transmitted using this code [13].

Order-1 static Huffman coding is essentially identical, except that 257 arrays are used, and characters are added to ArrayN, where N is the value of the order-1 context. At transmission time, the length of each array is measured. If an array is shorter than M (the minimum array length), it is added to Array256 , the ’merge array’. Finally, all non-merged arrays and the merge-array itself are transmitted using the method described for order-0 static Huffman.

A value of M as low as 16 results in maximum compression, while larger values of M result in faster operation (because fewer Huffman trees need be built). Typically a value of M = 128 is suitable; in this case, only about 20 Huffman trees are built. Also note that 256 binary flags must be sent to indicate whether or not an array was merged (these flags may be run-length compressed into about 3 bytes). For non-merged arrays and the merge-array itself, the length of the array must also be sent. The length of the arrays and the Huffman code lengths are sent using variable-length integers; the Huffman code lengths are further compressed by transmitting the maximum valued byte encountered and by run-length encoding the zero-length codes (symbols which were never seen).

The order-1 Static Huffman method may be easily extended to higher orders by using an order-n context, and hashing it down to an index, h, of 12 bits or less. This h is then used as the index into the character arrays, instead of using the value of the order-1 context. This produces a fast, reasonably good stand-alone compressor.

A report of the performance of the order-1 static Huffman coder, along with the performance of a full order-1-0 PPM model, follow:

	O1-StaticHuff	O1-Raw	
speed	60k	7k	(bytes per second)
bib	3.449	3.449	
book1	3.496	3.597	
book2	3.678	3.769	
geo	5.304	4.729	
news	4.101	4.151	
obj1	5.510	4.560	
obj2	4.315	4.106	
paper1	3.934	3.817	
paper2	3.627	3.619	
pic	1.615	0.835	
progc	4.147	3.845	
progl	3.400	3.317	
progp	3.528	3.357	
trans	3.577	3.484	
average	**3.834**	**3.617**	

Figure 3 : order-1 Static Huffman performance

Section 4 : A New Data Structure for Cumulative Probabilities

A new method of storing cumulative probabilities is presented, which makes use of deferred-innovation : a requested change is not performed on the data structure until some finite time after that change was requested.

The current state of the art in cumulative probability (cumprob) storage is the Fenwick Tree [7]. Maintenance of this data structure, however, still requires approximately 40% of the CPU time used by an order-0 arithmetic coder, making cumprob maintenance the most time-consuming phase.

To solve this, the input is divided into quanta of length N. For each quantum there are two probability tables, A and B. For any one quantum, Table A is already full of information, and Table B is empty. Cumprobs are read from Table A, and used to encode symbols. Probability updates are performed on Table B. At the end of a quantum, Table B is applied to Table A, and Table B is cleared. The next quantum is then compressed.

Because of the separation of the "Read CumProb" (only performed on Table A) and the "Update CumProb" (only performed on Table B) operations, a big improvement may be made : Table A is stored as a linear array of summed cumulative probabilities. Table B is kept as a linear array of un-summed symbol counts. Thus, the "Read CumProb" operation is a simple array reference. "Update CumProb" consists only of incrementing an array value. The "Apply Table B to Table A" operation becomes a process of summing the counts into cumprobs.

If M is the size of the alphabet being encoded (recall: N is the quantum size), then this method requires O(N) + O(M) time for each quantum, or O(1) + O(M/N) time for each byte. Maintaining a Fenwick Tree requires O($\log_2$M) time for each byte [7]. There is an N (approximately 32, for M = 256), for which these two methods take the same amount of time; lower values of N (such as N=1) are faster with a Fenwick Tree; larger values of N are faster with this new method of deferred-probability-summation. In practice, values of N as high as 1024 may be used without significant penalty to compression, which results in extremely fast speeds.

Note that in all cases, if a character is not in the order-0 model and an escape is transmitted, then the character is sent with an order-(-1) model, which is simply a flat code.

Section 4.1 : Improvements to Deferred-Summation

As presented above, the Deferred-Summation model compresses order-0 data at about 0.2 bpb worse than a full-precision model. Several improvements are presented here which reduce this penalty to less than 0.1 bpb.

First, if N = 255 is used, Table B is cleared between quanta, and the escape count is set to 1, then the cumulative probability total (cumprobtot) of Table A is 256 at all times. Thus, the division performed by the arithmetic coder may be replaced by a right-shift of 8 bits. Using an 8 bit right-shift is highly desirable in conjunction with the "Cormack Renormalization" described in the following section, and is also much faster than division.

Better compression can be attained if N = 127 is used, and Table B is halved after each quantum instead of being cleared (note that, due to integer rounding effects, N will be variable and slightly larger than 127 in order to produce a cumprobtot of 256).

Still better compression can be attained by incrementing the escape count every time an escape is taken, instead of just using a fixed escape count of 1. This further decreases N, in practice. Also, because the escape count is never decremented, it helps to limit the escape count to a maximum value of 40.

Because the cumprobtot is so low (256) decoding can be implemented with a direct lookup table : when Table B is copied to Table A, a third table (Table C) is created. Table C has 256 entries, each of which specifies the character (or escape) decoded by a certain "target cumprob". The result is that decoding requires the same amount of time as encoding, in O() notation; the actual amount of time required to decode a byte is approximately double that required to encode a byte.

Note that there is a pathological case for Deferred-Summation which does not exist for a normal order-0 model : if the input data was symbol X repeated N times, then symbol Y repeated N times, then symbol X repeated N times, etc. , Deferred-Summation would write each symbol in about 11 bits, whereas a normal arithmetic encoder would write each symbol in about 0.5 bits.

Applying exclusion to the order-(-1) model was found to give only a negligible improvement to compression, and was abandoned so that an 8-bit right-shift could be used in place of the division operation.

Note that neither Deferred Summation nor the Fenwick Tree structure are well suited to modeling sparse alphabets, such as those encountered in high-order context modeling. Deferred Summation suffers not only a loss of ‘speed per byte of memory used’ efficiency, but also a great loss of compression efficiency. Simple linear lists are more effective in this case [20].

Section 4.2 : A Fast Arithmetic Encoder for use with Deferred Summation

An extremely arithmetic encoder is presented for use with the Deferred-Summation probability model.

First, some notation for the arithmetic encoder must be presented:

`L`,`R` = Low Bound and Range of the current arithmetic-encoder interval (these are `C` and `A` in Langdon-Rissanen terms). Both `L` and `R` are 24-bit values.
Initially, `L = 0` and `R = 0xFFFFFF`

`sL`,`sR`,`sT` = symbol-low,symbol-range, and symbol-total : the cumulative probability interval for the character which is to be encoded, and the sum of all symbol probabilities

Now, a basic design for an order-0 arithmetic encoder is given, which divides the operations:

1. model : provides cumulative probabilities
this is handled by the Deferred Summation tables, or a Fenwick tree.

2. scaling : reduces the interval of the arithmetic encoder so as to specify the desired character

```
L += sL*R/sT
R  = sR*R/sT
```

3. renormalization : possibly enlarge the interval to ensure the necessary precision, and transmit part of the currently specified interval.

Because `sT = 256`, Step 2 may be replaced with:

```
L += (sL*R) >> 8
R  = (sR*R) >> 8
```

The multiplications are left as-is, because they are quite fast on most architectures (division is typically four times slower than multiplication).

Step 3 uses a method introduced by F. Rubin [3], and popularized by G.V. Cormack's DMC implementation [1]. It is:

```
while ( R < 256 )
   {
   output(L>>16);
   L <<= 8; R <<= 8;
   L &= 0xFFFFFF; R &= 0xFFFFFF;
   if ( (L + R) > 0xFFFFFF ) R = 0xFFFFFF - L;
   }
```

This method is approximate, and therefore produces slightly longer output than the canonical CACM-87 style normalization [5]. However, the output is byte-aligned, and the normalization loop is almost always only performed once; these properties make this method extremely fast.

The line:

```
if ( (L + R) > 0xFFFFFF ) R = 0xFFFFFF - L;
```

is where the extra output comes from. It can be seen that R is shrunk to less than it need be, so extra normalizations are needed, and more bytes are output. However, this only happens 1/65536th of the time, so the excess output is negligible.

The above-mentioned line is due to F. Rubin, and is the only difference between this normalization method and the Rissanen-Langdon (R-L) style normalization [2]. An R-L coder handles the case of `( (L+R) > 0xFFFFFF )` by propagating a carry-over into the previously-output data.

The resulting order-0 coder is faster than all known adaptive (one-pass) coders, including Adaptive Huffman.

A report of results (in terms of output bits per input byte on the files of the Calgary Corpus) for the DCC95 coder (with code-bits = 32 and frequency-bits = 16), using an adaptive order-0 character model, and the order-0 DefSum coder (described herein) follows (the DCC95 coder uses a Fenwick Tree for the model, R. Neal's Shift-Add method for the scaling phase, and CACM-87 style renormalization) [6]:

	DefSum	DCC95	
speed	50k	6k	(bytes per second)
bib	5.520	5.221	
book1	4.700	4.535	
book2	4.886	4.784	
geo	5.898	5.659	
news	5.422	5.190	
obj1	5.710	5.997	
obj2	6.078	6.212	
paper1	5.104	5.019	
paper2	4.758	4.624	
pic	1.198	1.195	
progc	5.411	5.244	
progl	4.888	4.798	
progp	5.016	4.907	
trans	5.689	5.542	
average	**5.020**	**4.923**	

Figure 4 : order-0 Deferred Summation performance

Conclusion

Several new developments in context-coding have been presented.

First, the PPMCB method was presented as a fast method for modeling high-order correlation using only O(1) time. One fast PPMCB coder attains an average of 2.35 bits per byte on the Calgary Corpus.

Second, the Order1-ListLRU structure was presented, which uses only 7k bytes of memory to perform order-1 modeling. The compression achieved by the Order1-ListLRU is better than that attained with standard order-1 modeling.

Thirdly, a method was presented which facilitates the use of two-pass Huffman encoding in place of order-1-0 modeling and arithmetic encoding. This method is very fast (about 10 times faster than arithmetic encoding) and achieves compression similar to that of order-1 arithmetic encoding, with exceptional performance on large files.

Finally, a very fast order-0 coder was presented, using the Deferred Summation probability model, and a very fast arithmetic coder with byte-aligned output and approximate renormalization. This coder is approximately 8 times faster than previous arithmetic encoders, and achieves only about 0.1 bpb less compression.

Note that there are still unaddressed problems with order-1 coding : Order-1 Static Huffman is fast, but non-adaptive, and memory-greedy; Order1-ListLRU is memory-use efficient, but no faster than exact order-1 coding.

References

1. G.V. Cormack , DMC coder implementation, available by FTP from : plg.uwaterloo.ca/pub/dmc
2. G.G Langdon, Jr. and J. Rissanen, "A simple general binary source code", IEEE Trans. Information Theory IT-28 (1982), p. 800 - 803
3. F. Rubin, "Arithmetic Stream Coding Using Fixed Precision Registers", IEEE Trans. Information Theory IT-25 (6) (1979), p. 672 - 675
4. C. Jones, "An Efficient Coding System for Long Source Sequences", IEEE Trans. Information Theory IT-27 (3) (1981), p. 280 - 291
5. I.H. Witten, R.M. Neal, J.G. Cleary, "Arithmetic Coding for Data Compression", Communications of the ACM 30 (1987), p. 520 - 540
6. A. Moffat, R. Neal, I.H. Witten, "Arithmetic Coding Revisited", DCC95, p. 202 - 211
7. P.M. Fenwick, "A New Data Structure for Cumulative Frequency Tables", Software-Practice and Experience 24 (3) (1994) 327-336.
8. D.S. Hirschberg and D.A. Lelewer, "Context Modeling for Text Compression", in J.A. Storer, Image and Text Compression, (1992) p. 113 - 144
9. A. Moffat, "Implementing the PPM data compression scheme", IEEE Trans. Communications COM-38 (11) (1990), p. 1917-1921
10. D.E. Knuth, The Art of Computer Programming, (1973) p. 508-509
11. T.C. Bell, J.G. Cleary and I.H. Witten, Text Compression (1990)
12. D.A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proc. I.R.E. 40 (9) (1952) p. 1098-1101
13. J.B. Connell, "A Huffman-Shannon-Fano Code", Proceedings of the IEEE (1973), p. 1046 - 1047
14. C. Bloom, "LZP : A New Data Compression Algorithm", submitted to DCC96
15. P.C. Gutmann, "Practical Dictionary/Arithmetic Data Compression Synthesis", MsCS Thesis, Dept. of Computer Science, The University of Auckland
16. J.A. Storer and T.G. Szymanski, "Data Compression via Textual Substitution", Journal of the ACM 29 (4) (1982), p. 928 - 951
17. W.B. Pennebaker and J.L. Mitchell, JPEG : Still Image Data Compression Standard (1993) p. 149 - 167
18. P.M. Fenwick and P.C. Gutmann, "Fast LZ77 String Matching", Dept. of Computer Science, University of Auckland, Tech. Report 102, Sep. 1994
19. The Calgary Compression Corpus, available by FTP from: ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/
20. P.M. Fenwick, private communication