A New Locally Adaptive Data Compression Scheme using Multilist Structure

HENRY KER-CHANG CHANG AND SHING-HONG CHEN

Graduate School of Resources Management, National Defense Management College, PO Box 90046-17, Chung-Ho, Taipei, Taiwan, ROC

A new locally adaptive data compression scheme has been proposed in this paper. The proposed scheme may be viewed as an extension of the previous work by Bentley *et al.* in 1986. It is developed by the application of multilist structure and separate treatments for different types of data. Improving the overall performance of the method provided by Bentley *et al.* is its objective. The multilist structure aids in shortening the length of a transmitted code word and the treatment of numeric data aids in compression effect improvement. The feasibility of the proposed scheme has been validated by several experimental results and some theoretic analyses. The availability of various prefix coding methods developed by Järnvall have finally been tested here. The incorporation of G1 prefix code in the proposed compression scheme works well in performance promotion.

Received January 25, 1992, revised January 15, 1993

1. INTRODUCTION

Concerning the communication problem in a computer network, data compression is important because the cost of both transmission time and storage space can be reduced if a compression technique is adopted. These advantages are particularly obvious to digital communication while the application of information interchange on miscellaneous networks has recently become popular. A locally adaptive data compression scheme based on the principle of reference of locality has been previously proposed by Bentley *et al.* [1]. A word-based compression scheme is implemented using a self-organizing list as an auxiliary data structure. Both the sender and receiver are to maintain lists of identical content. When the compression process begins to work, the locality of reference implies that a sequential search over the list of words is processed so that the frequently appearing words can always be promptly accessed in the front of the list and are coded by short integer codes. A procedure named 'move-to-front' (MTF) reorganizes the sequence for the words in the list as each word is processed. The reorganization routine is to delete the position for the currently processing word and move it to the front of the list. When the current encoding word cannot be found in the list, the encoding word is to be assigned a number of one more than the total number of words in the list and is inserted at the front of the list. The sender is then to transmit the number followed by that word to the receiver. The receiver is to know that a new word is coming as both the number and word are detected. However, only the number representing the position of a word in the list is transmitted if the list has already contained that word. To maintain the word list efficiently, two interlinked data structures are used. Bentley *et al.* suggest to prefix the binary representation of an integer i, $i \geqslant 1$ with leading $\lfloor \log i \rfloor$ zeros, $\lfloor\,\rfloor$ representing the floor function. The prefix of $\lfloor \log i \rfloor$ zeros indicates the length of a bit string for an integer i and separates the boundary between coded words. Thus, this treatment encodes an integer i with $1+2\lfloor \log i \rfloor$ bits. A binary trie is applied to convert words into positional integers and a binary tree records the order of words in the word list. The following example cited from Bentley *et al.* [1] may illustrate the detailed operations for the sake of compressing a message of small letters separated by single spaces. The following message is assumed here to be transmitted

the car on the left hit the car number I left.

The corresponding encoded words will be

1 the 2 car 3 on 3 4 left 5 hit 3 5 6 number 7 I 5.

Bentley *et al.* emphasize that the locally adaptive data compression scheme has many implementation advantages: it is simple, allows fast encoding and decoding, and requires only one pass over the data to be compressed. The theoretical analysis of performance has been proved to never have performed significantly worse than Huffman coding. In addition, if the message to be transmitted includes locality of reference, the scheme can substantially outperform Huffman coding because a word will have a short encoded word if it is frequently used and a long encoded word if it is used rarely. Some experimental results are demonstrated by comparing three compression algorithms of byte-level Huffman coding, word-level Huffman coding and the MTF scheme. The size of the word list for the MTF scheme is also tested using 8, 16, 32, 64, 128 and 256 words. The experimental results validate the feasibility of Bentley *et al.*'s algorithm. It also reveals a fact that the larger the size of the word list, the better performance the algorithm can have. However, the authors believe that this phenomena occurs since the word list structure used in their scheme will create a problem. Just as the description in their paper, maintenance of an efficient word list is somewhat complicated and is hard to implement. The reorganization of the word list using a trie structure and the searching over word list are known to be very complex when being put into comparison with other routines. In addition, the length of the code for the integer, representing the position of a word in the list, is too long since the height of a trie is closely relevant to the length of the list. Another problem appears that their method does not have an appropriate treatment for numeric data. They have the numeric data encoded in a way similar to text words, each numeric data has a transmission cost consisting of an extra positional indicator of $2\lfloor\log i\rfloor+1$ bits and the numeric data in ASCII format if the numeric data is a new one. While a general text file probably contains less redundant numeric data, most of the numeric data would be encoded as new words and would cause a large increase in transmission cost using Bentley *et al.*'s method. Those factors may dramatically influence the performance. An appropriate selection of the data structure and consideration of easy implementation may improve the overall performance. A new locally adaptive data compression scheme is therefore proposed here.

Järnvall [3] emphasized how to improve the performance of the coding scheme previously developed by Bentley *et al.* Both the prefix coding method and the indication of a new symbol have been discussed. Two new prefix coding techniques have been provided by him for encoding the list position. Positional integers are encoded by two approaches which are extended from the γ code and δ, named 'Delta' in this paper for convenience, code developed by Elias [2]. The γ code is exactly the prefix code used by Bentley *et al.* in which there are $\lfloor\log i\rfloor$ leading zeros followed by $\lfloor\log i\rfloor+1$ bits consisting of the binary representation of integer i. The δ code exploits the γ code in determining the length of its codeword. The first part of the codeword $\delta(i)$ is $\gamma(\lfloor\log i\rfloor+1)$ and the rest is the binary representation of integer i without the leading 1-bit. Let $B(i,l)$ denote using l-bit for the binary representation of integer i, $0 \leqslant i \leqslant 2^l-1$ (e.g. $B(6, 5)=00110$). He designs a general format of prefix code G_n, $n \geqslant 1$, for integer i as

$$G_n(i)=\left\{\begin{array}{cc} 10^n, & \text{if } i=1 \\ 1B(i, n+1), & \text{if } 1<i<(n+1)^2 \\ 0^{\lfloor\log i\rfloor-n}B(i,\lfloor\log i\rfloor+1), & \text{if } i \geqslant (n+1)^2 \end{array}\right\},$$

where the standard binary representation of integer i is $B(i,l)$ and the 0^n denotes n consecutive zeros. Another prefix code D_n derived from G_n is

$$D_n(i)=G_n(\lfloor\log i\rfloor+1)B_R(i,\lfloor\log i\rfloor),$$

where $B_R(i,m)$ denotes the m rightmost bits of the binary representation of a positive integer i. He further compares his prefix coding methods with the Fibonacci code for the sake of finding the advantage of his coding methods. Each prefix coding has been concluded by him to be able to be the best in a certain limited range of integers; a specific prefix coding method does not exist which outperforms all other methods for any case. He also selects the usage of 1 byte to indicate the first appearance of a symbol, instead of using only 1 bit as a flag previously suggested by Moffat [5]. Since the multilist data structure of the proposed scheme has already induced a shorter length than that for a single list, finding out what is the best prefix coding method for positional integers in the proposed scheme therefore becomes the motivation here. Besides, the 1-bit indicator for any new symbol in a text is applied in the proposed data compression scheme.

A new locally adaptive data compression scheme is proposed in this paper which is developed on the basis of a multilist data structure and different compression approaches for both numeric data and punctuation marks. The motivation for the proposed scheme originates from the previous work of Bentley *et al.* [1] The proposed compression scheme has advantages including not only the original characteristics of the result of Bentley *et al.*'s work, but a higher ratio of data compression can also be gained. Several experimental tests of text files, Pascal programs and numeric data validate the characteristics of the proposed scheme.

The remainder of this paper is organized as follows. The implementation of the proposed compression scheme is explained in Section 2. Both the encoding and decoding algorithms are described. Empirical tests and theoretic analysis of the proposed scheme are analyzed in Section 3. The performance is further compared with those of Bentley *et al.*'s compression techniques. The observed idea is finally summarized in Section 4 and proposals for future researches are discussed.

2. THE PROPOSED DATA COMPRESSION SCHEME

According to the consideration of reducing the cost of transmission and storage, the proposed compression scheme emphasizes how to shorten the word list and how to access the word list efficiently. In addition, we introduce a treatment for both numeric data and punctuation marks which works well for the promotion of compression. The proposed scheme shortens the word list by using a multilist (n-list) structure so that the position of a specific word in the list can be represented by an integer which has a shorter bit length in representation than the one generated by a single word list. The multilist is searched over through usage of a random access approach for the sake of considering efficient access of the word list. Both numeric data and punctuation marks are encoded in a specific list where we apply the prefix coding method with special indicators for numeric data to reduce the codes in the original format.

Since the proposed data compression scheme is designed in a one-pass operation mode, both the sender and receiver have to synchronously maintain 27 words lists of an identical content. The sender and the receiver initially create these empty lists. Twenty-six lists among them represent word lists of alphanumeric data leading with characters from 'a' to 'z'. The other list, indexed by a special code '@', contains punctuation marks, integers, real numbers and an indicator, integer one, which marks a new word. The data structure of the 27th list has been elaborately designed here for the sake of shortening the bit length of encoded words for non-alphanumeric data as much as possible and to provide a special treatment for the new word. The first array of each list is kept empty for compression process requirements. We apply the idea suggested by Jarnvall that integer one is used as an indicator for new words, all positional integers must be larger than one. In this paper, all data to be transmitted are assumed to be generated within a predefined domain. In addition, the alphanumeric data and all non-alphanumeric data, both numeric data and punctuation marks, are treated with different approaches in the proposed compression scheme. The structure of the lists may notably vary depending on the scope of practical applications. The domain of text files is limited here to use only 26 alphabets, integers, real numbers and some punctuation marks for the sake of clear explanation. The proposed compression scheme can be easily tailored to any application. In the following, we describe the compression approaches for texts, numeric data and punctuation marks separately.

A one-pass coding scheme is developed in this paper on the basis of predefined Huffman codes for these leading 27 characters. The Huffman codes for these 27 characters (Table 1) are generated by the conventional static Huffman coding technique. A random access approach is then defined to access each list efficiently according to the leading character of any word. The integer representing the position of a word in a specific list is encoded using the prefix code previously developed by Järnvall [3]. When a token of a text file is to be sent, the sender will access a specific list indexed by the leading character for determination of the position of the word in the list. So the lists are organized in a sequence according the ASCII codes of alphabets from '@' to 'z'. These lists are chosen randomly using the leading character in the ASCII format. The sender will sequentially search the selected word list for the purpose of finding whether or not the currently processing word is in that list. The sender will transmit only a Huffman code for the leading character of the word and an integer which marks the position of the word in the list. Otherwise, a Huffman code for a special code '@' followed by the prefix code of integer one and the original word is sent. The combination of code '@' and integer one indicates that the following is a new word and the word has to be inserted into the list. Once the word is sent or received, it is moved to the front of the list based upon the principle of referential locality.

The numerical data is processed by different approaches. A special code '#' will indicate integers. When the first integer in the text file appears, the message consisting of Huffman code '@', the prefix code of integer one, the Huffman code '#' and the prefix code of that integer is sent to the receiver. The combination of Huffman code '#' and the prefix code of integer one is the indication of the appearance for the first integer. Afterward, any integer is encoded by the combination of Huffman code '@', the prefix code of the position for code '#' in the 27th list and the prefix code of the integer. The real number will use another code '&'. It will be encoded in the same way as for integers except for the integer part and the fraction part of a real number are separately transformed into corresponding prefix codes. The decimal point of a real number is omitted in the proposed coding technique. The Huffman code for '&' followed by these two prefix codes are sent.

Punctuation marks are treated by the same way as it is for text words except that all punctuation marks are stored in the 27th list. The combination of Huffman code '@', the prefix code of integer one and the original punctuation mark is sent for the new punctuation mark. It will be encoded by the Huffman code '@' plus the prefix code of the position in the 27th list.

In summary, the proposed data compression scheme supports different encoding processes for different types of data; the whole treatment can be clearly illustrated by the diagram shown in Figure 1. All procedures and the coding algorithm for the proposed compression scheme are shown in the Appendix.

TABLE 1. Huffman codes for leading characters of word lists

Character	*Weight*	*Code*
a	4	0010
b	4	0011
c	4	0100
d	2	10000
e	2	10001
f	2	10010
g	1	110010
h	2	10011
i	4	0101
j	1	110011
k	1	110100
l	2	10100
m	1	110101
n	2	10101
o	4	0110
p	2	10110
q	1	110110
r	2	10111
s	4	1111
t	4	0111
u	1	110111
v	1	111000
w	2	11000
x	1	111001
y	1	111010
z	1	111011
@	6	000

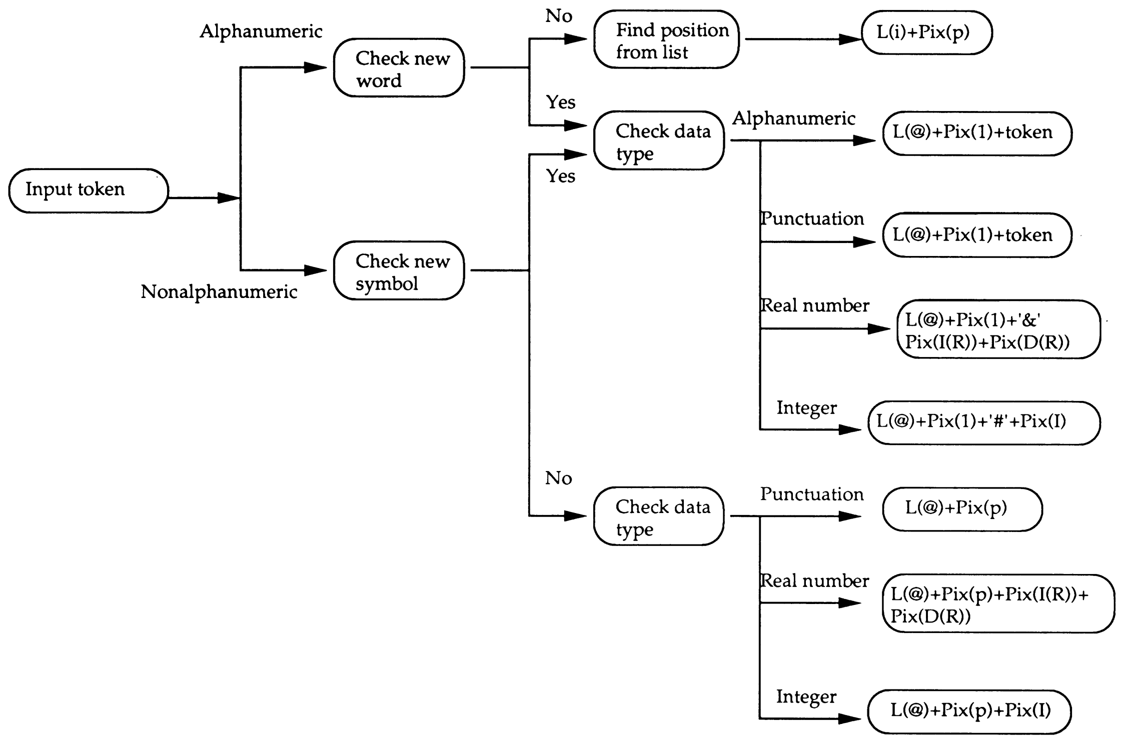

FIGURE 1. Encoding flow chart of the proposed data compression scheme. $L(i)$: list code; $Pix(x)$: Prefix code for an integer x; $I(R)$: integer part of a real number R; $D(R)$: decimal part of a real number R; p: positional integer greater than one; I: integer.

3. PERFORMANCE EVALUATION AND COMPARISON

Two data compression techniques have been implemented here for performance comparison in order to evaluate the performance of the proposed new locally adaptive data compression scheme and to gain further insight into its performance. We complete two kinds of tests. The first one reveals the general function of the proposed compression scheme for text files consisting of both alphabetic and numeric data. The second one concentrates on the superiority of the proposed scheme for handling numeric data only. Finally, different prefix coding techniques based upon Järnvall's research for positional integers are also tested in order to find the availability of one prefix format which is the most appropriate prefix code for all positional integers. Seven prefix coding techniques have been tested here in order to find the best prefix coding technique for the proposed data compression scheme.

3.1. Empirical Tests

Two different types of test data have been considered here for the sake of demonstrating the generality of the proposed compression scheme. The first test data consists of six text files of various sizes ranging from 17 to 33 Kbytes. Another five Pascal programs of sizes over 11 Kbytes were tested in a second experiment. The arrangement of test data has two reasons. The first reason is that a difference essentially exists between these two types of test data. The variation for the content of a program is generally less than that for a text file because the vocabulary of the program is always subject to a limited domain. The second reason is that an enlarged size of test data can keep the performance analysis and comparison more objective. The results of two compression techniques have been compared here, including both Bentley *et al.*'s locally adaptive coding which uses a list of 256 words and the proposed scheme. The performance comparison is analyzed and measured in bits according to the memory space needed to encode the test data. The results of the experiments are listed in Table 2. The numbers listed in Table 2 reflect all the encoded bit lengths for all given test data. The compar-

TABLE 2. Comparison of compression results of two different techniques

Data type	*Size*		*Bentley* et al.*'s method*	*The proposed scheme*
	bytes	*bits*		
Text files	17027	136216	38494	25275
	18620	148960	46155	31780
	19426	155408	49028	34902
	21321	170568	48946	32705
	30426	243408	72243	50535
	32345	258760	77386	54604
Pascal programs	11896	95168	38400	37283
	12860	102880	42166	41085
	14348	114784	46172	45036
	15701	125608	50747	48715
	16285	130280	52647	51497

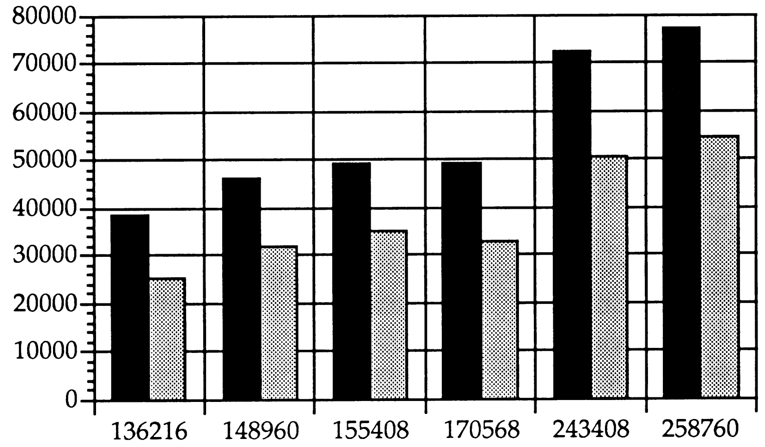

FIGURE 2. Performance comparison for text files. ■, Bentley's method. □, Proposed method.

isons are also illustrated in Figures 2 and 3. The proposed compression scheme can be affirmatively confirmed here from the results of the experiment to have an improvement over the previous work of Bentley *et al.* for text file compression. The original objective for improving the efficiency of Bentley *et al.*'s compression method has definitely been achieved. The reasons for performance improvement can be found from the collected statistical data shown in Table 3. The relationship between the required length of the word list for Bentley *et al.*'s method and the proposed scheme in the running time is compared in Table 3. As mentioned before, a shorter word list needs to be used here which can then derive a shorter bit string to represent a positional integer for encoding a text word. The length of the word list is actually dependent on the size of a test file. While the number of distinct text words increases, the bit string of a positional integer for encoding a text word proportionally increases. The numbers of distinct words for each test file are shown in the fourth column of Table 3. There are 880 different text words for the first test file as an example. The fifth column of Table 3 lists the number of text words transmitted for a given test file. All statistic data in fifth column verify that Bentley *et al.*'s method needs longer bit strings in encoding text words. Another problem appears that since a fixed-length list of 256 text words is always less than the total number of different words, an abnormal case has therefore been found here that a commonly appearing text word may be repetitively transmitted as a new word. A general problem to be found here is that the currently processing word is exactly the previously deleted word, but it has to be inserted into the list again. The sixth column of Table 3 lists all the numbers of words retransmitted for each test file. The influence of those retransmitted words significantly expands the lengths of encoded words. Taking the number for the first test file as an example, 290 words have to be retransmitted here although these 290 words had already been encoded before. A fact may thus be encountered here that a primitive word is probably transmitted many times once the list is full. The extra load of re-encoding these 290 words will induce two overloads in that both the transmission time and storage space are increased.

The last two columns of Table 3 list both the maximum length and the average length among the 27 word lists for all test files using the compression scheme proposed here. The maximum length is the length for the longest word list among the 27 lists to encode a test file and the average is also computed by dividing the sum of the length by 27. All the maximum lengths of the 27 lists for each test file are approximately only half of Bentley *et al.*'s technique. The average length is even shorter of approximately only one-fourth the list size of Bentley *et al.*'s technique. The advantages of the proposed compression scheme are thus two-fold. One is a smaller positional integer to encode a word; the other is that it will never retransmit any word. These are the reasons which explain why the proposed scheme can obtain a better performance than Bentley *et al.*'s technique. Once the sizes of test files are expanded, the proposed compression scheme is thought to give even better compression results.

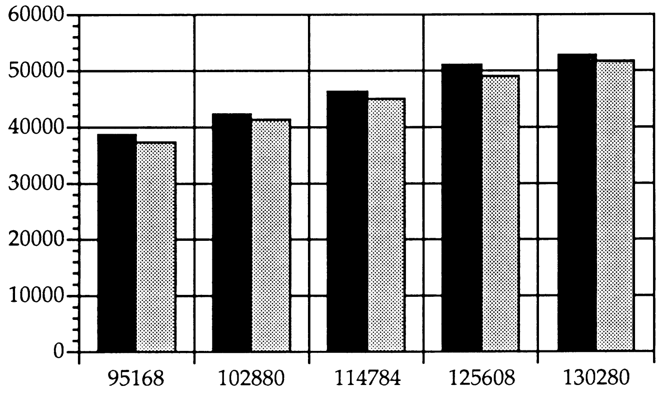

FIGURE 3. Performance comparison for Pascal programs. ■, Bentley's method. □, Proposed method.

To test the superiority of the proposed scheme for handling numeric data, we take the indices of wholesale prices from 1983 to 1990 in Taiwan as data to be compressed. There are five numeric data files of various

TABLE 3. Comparison of length of word list in running time

Data type	Size		Number of words	*Bentley* et al.*'s method*		*The proposed scheme*	
	bytes	*bits*		*new words transmitted*	*repetitive send*	*average length*	*maximum length*
Text files	17027	136216	880	1170	290	32	98
	18620	148960	868	1181	313	32	103
	19426	155408	809	1168	359	29	80
	21321	170568	1044	1487	443	38	117
	30426	243408	1264	2032	768	46	136
	32345	258760	1292	2144	852	47	139
Pascal programs	11896	95168	266	268	2	9	21
	12860	102880	282	284	2	10	25
	14348	114784	294	296	2	10	25
	15701	125608	310	312	2	11	26
	16285	130280	321	324	3	11	28

sizes. The compressed results are shown in Table 4. It is obvious that the proposed scheme has about 30% cost reduction, i.e. it is much better than that for Bentley *et al.*'s method. It is especially worthy to note that Bentley *et al.*'s method has a negative compression effect, as it can be seen in Table 4. The compression results are even more expensive than the original file sizes. The reason is that Bentley *et al.*'s method treats the numeric data in a way similar to alphabetic data. A numeric data therefore needs a transmission cost including the original ASCII code format plus the indicator of the new word.

TABLE 4. Comparison of compressed results for numeric data using two different techniques

File sequence	Size		*Bentley* et al.*'s method*	*The proposed scheme*
	bytes	*bits*		
1	942	7536	7901	5019
2	1135	9080	9891	6133
3	1325	10600	11829	7277
4	1614	12912	14829	8983
5	1822	14576	17041	10074

The amount of extra cost increases rapidly if the data files are of less redundancy.

Finally, seven coding methods have also been tested here in the proposed data compression scheme for the sake of finding the availability of various prefix coding techniques based on the Järnvall research. These seven coding techniques include the original Delta prefix code, G1, G2, G3, D1, D2 and the Fibonacci code. All results are compared with that of Bentley *et al.*'s method. Again, the test data consists of text files and Pascal programs. All statistical data are listed in Table 5. Figure 4 illustrates the compression results for the text files and Figure 5 depicts the comparison for the Pascal programs. Not all coding methods proposed by Järnvall work well for the test data but the coding method G1 has the best compression efficiency among them.

This can be easily found here from the experimental results. The reason is that there are 94.6% of positional integers which are less than nine. This is accordance with the assertion made by Järnvall that the G1 coding method is especially suitable for integers less than nine. The prefix coding method designed by Järnvall is therefore helpful to the improvement even the multilist struc-

TABLE 5. Comparison of compression results based on various prefix coding formats

Data type	Size		*Bentley* et al.*'s method*	*The proposed scheme with various prefix coding formats*						
	bytes	*bits*		*Delta*	*G1*	*G2*	*G3*	*D1*	*D2*	*F*
Text files	17027	136216	38494	25553	24342	25049	28399	24939	28289	26404
	18620	108960	46155	30853	30143	31231	35368	31427	35564	32952
	19426	155408	49028	33785	32822	33948	38420	34468	38940	35985
	21321	170568	48946	31937	31429	31915	36159	32118	36362	34061
	30426	243408	72243	49341	48028	48904	54183	49338	55614	52016
	32345	258760	77386	53336	51787	51364	58076	53207	59913	56145
Pascal programs	11896	95168	38400	34722	33022	35883	50731	36693	41541	37283
	12860	102880	42166	38292	36470	39652	45002	40494	45844	41082
	14348	114784	46172	41970	39901	43400	49252	44385	50237	45033
	15701	125608	50747	46336	44011	47842	54289	48982	55429	49696
	16285	130280	52647	47994	45560	49567	56248	40742	57423	51476

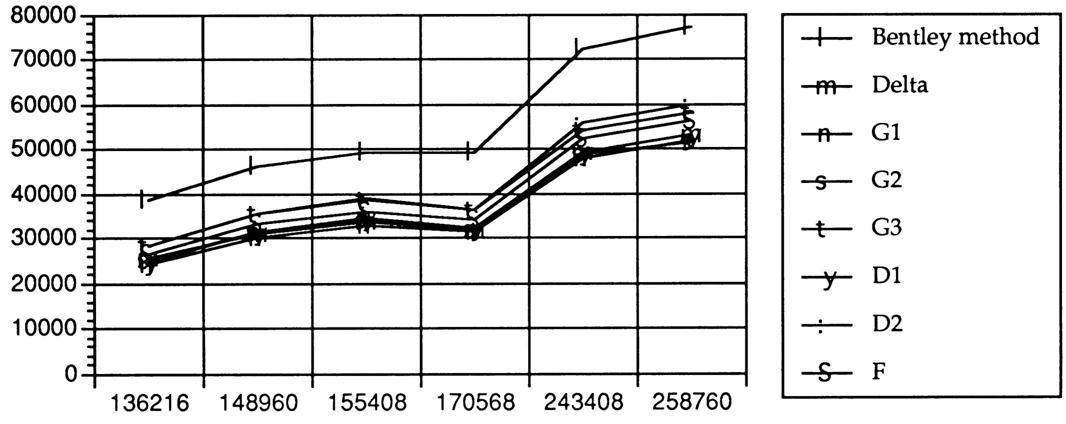

FIGURE 4. Performance comparison using various prefix coding formats for text files.

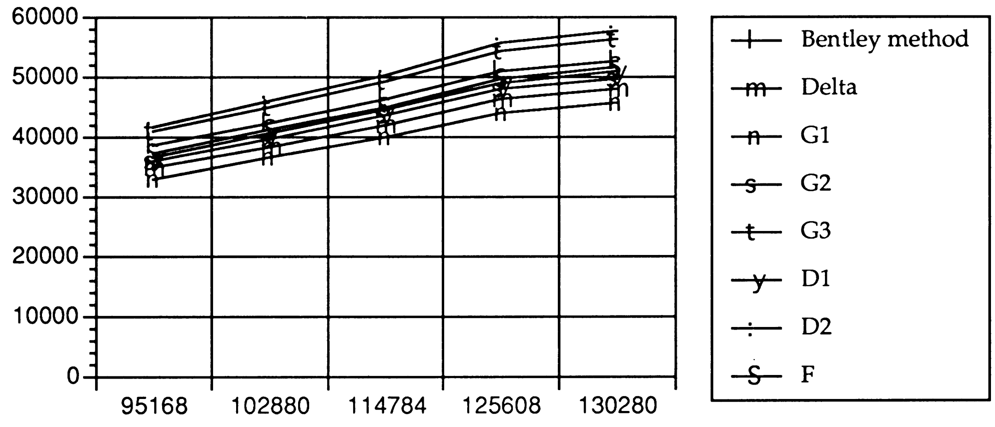

FIGURE 5. Performance comparison using various prefix coding formats for Pascal programs.

ture is used. The incorporation of Järnvall's prefix coding method into the proposed multilist data structure can therefore be naturally confirmed here to induce a significant improvement over the data compression scheme previously developed by Bentley *et al.*

3.2 Theoretic Analysis and Comparison

We can have two assertions here to demonstrate the superiority of the proposed scheme. The first one refers to the advantage of the multilist data structure; the second refers to the advantage of the proposed treatment for numeric data. Both our method and Bentley *et al.*'s method apply the prefix coding for positional integers, the positional integers are strongly dependent on the length of the word list. Since we have 27 lists for the compression process instead of only one list used in Bentley *et al.*'s method, let L be the length for the single list used in Bentley *et al.*'s method, any positional integer in a single list would be encoded in $2|\log L|+1$ bits, but our method may have an average length of approximately less than $L/27$ for all lists, so any positional integer would use $2|\log L/27|+1$ bits to encode. The inequality

$$2|\log L/27|+1 \leqslant 2|\log L|+1$$

should always be true when the prefix codes of both two methods are compared. We should point out that the proposed scheme takes a constant time in finding a specific list from these 27 lists and it is irrelevant to the overall performance. So the inequality definitely reveals the advantage of the proposed multilist data structure.

Bentley *et al.* proved that their method has a transmission cost less than that of static Huffman coding as

$$\rho_{\mathrm{MF}}(X) \leqslant 2\rho_{\mathrm{H}}(X)+1,$$

where $\rho_{\mathrm{MF}}(X)$ is the average number of bits per word used to compress a text file with MTF scheme and $\rho_{\mathrm{H}}(X)$ is the same quantity for the static Huffman code. They assume that the cost of sending the raw words is ignored and the size of the MTF list equals the number of different words to be sent. However, the transmission of a raw word is an important factor for the compression performance, particularly for the numeric data. Using the proposed treatment for numeric data, the cost of sending a new numeric data is $12+2|\log X|+1$ bits where we need 3 bits for Huffman code of '@', 1 bit for integer one, 8 bits for ASCII format of '&' or '#' and $2|\log X|+1$ bits for prefix code of the numeric data X; but $2|\log N|+1+8S(X)$ bits are required for the treatment of Bentley *et al.*'s method where N is the positional integer in a list and $S(X)$ is the ASCII format for numeric data X; 40 bits for a numeric data of five digits as an example. It would be clear that

$$12+2|\log X|+1 \leqslant 2|\log N|+1+8S(X)$$

where $8S(X)$ bits for raw words is the dominate factor if numeric data X is approximately equal to the positional integer N. This inequality clearly shows that the proposed treatment of numeric data outperforms the results of Bentley *et al.*'s approach.

4. CONCLUSIONS

A new locally adaptive data compression scheme has been proposed. The usage of a multilist structure has been applied here with a random access approach to improve the compression performance from that of Bentley *et al.*'s work. We also suggest a treatment for numeric data to further improve the overall performance. Several experimental results have verified our original motivation and have validated the feasibility of our proposed scheme. The reasons for the improvement have also been explained through means of analyzing the statistical data collected in the running time. The overall performance could be improved if the process on the MTF list can be appropriately handled. The multilist structure is applied here instead of using only a single list. The advantage of a shorter encoded word could therefore be obtained here by reducing the length of a word list. An advantage of a fast encoding process could also be obtained since the hashing function of access time $O(1)$ is used and only a search over a short list is necessary. Besides, using the prefix coding for numeric data instead of transmitting it by its primitive ASCII format is helpful in compression effect promotion. The proposed data compression scheme has been shown to be more reliable than other adaptive coding schemes since the asynchronous problem in one word list will influence the code words from that list only, the overall performance will be affected only slightly. The feasibility of the prefix coding method proposed by Jarnvall has also been demonstrated here in the proposed data compression scheme. The G1 prefix coding method was found from the experimental results to provide a better compression performance when most of the positional integers fell in the domain less than nine. The combination of the multilist structure, separate treatments for various types of data and the prefix coding technique has definitely promoted the compression ratio more than that of Bentley *et al.*'s method. Future research may focus on the following:

1. How to develop an efficient compression technique which is error free from the channel noise?
2. Is it helpful to utilize information from the appearances of the words beginning with the same characters?
3. Can a parallel algorithm be profitable to data compression problem?

REFERENCES

[1] J. L. Bentley, D. D. Sleator, R. E. Tarjan and V. K. Wei, A locally adaptive data compression scheme. *Commun. ACM* **29**, pp. 320–330 (1986).

[2] P. Elias, Universal codeword sets and representations of the integers. *IEEE Trans. Information Theory*, **IT-21** pp. 194–203 (1975).

[3] E. Järnvall, *Fine Turing a Locally Adaptive Data Compression Scheme.* Technical report A-1990-4, Depart of Computer Science, University of Tampere, Tampere, Finland (1991).

[4] E. Makinen, On implementing two adaptive data-compression schemes. *The Computer Journal*, **32**, pp. 238–240 (1989).

[5] A. Moffat, Word-based text compression. *Software Pract. Experience*, **19**, pp. 185–198 (1989).

APPENDIX

In this section, we illustrate the encoding and decoding algorithms of the proposed data compression scheme using multilist structure. The following procedures define the major functions to be used in the compression and decompression processes:

hashf(c): compute the index of the leading character '*c*' of a word using hashing function.

position(i, w): compute and return the position for the word *w* in the *i*th word list; the return is zero if the word *w* is not in the list.

word(i, p): compute and return the word for the positional integer *p* in the *i*th list.

mtf(i, p): move the word at position *p* to the front of the *i*th list.

mts(i, p): move the word at position *p* to the second array of the *i*th list.

insertf(i, w): insert word *w* at the front of the *i*th list.

inserts(i, w): insert word *w* at the second array of the *i*th list.

look_up(c): look up the character *c* from the Huffman code table and return the corresponding Huffman code.

en_prefix(p): encode an integer *p* to a prefix binary form.

de_prefix(p): decode a prefix binary form of an integer *p* to the original format.

decode: read bit strings from the input buffer, look up from table *t*, and return the corresponding character.

float(f_1,f_2): merge integer f_1 and fraction f_2 to form a real number.

defloat(w,f_1,f_2): separate a real number into integer f_1 and fraction f_2.

The proposed data compression scheme can be implemented as the following programs using C language:

```
Compress:
{
  input w;
  if (w is not numeric)
  {
    if (w is punctuation)
      i=hashf('@');
    else
```

```
      i=hashf(w[0]);
    p=position(i, w);
    if (p>0)
    {
      cl=look_up(w[0]);
      c2=en_prefix(p);
      if (i=hashf('@'))
            mts(i, p);
      else
            mtf(i, p);
      output cl and c2;
    }
    else
    {
      cl=look_up(hashf('@'));
      c2=en_prefix(1);
      if (i=hashf('@'))
            inserts(i, w);
      else
            insertf(i, w);
      output cl and c2;
      output w in primitive form;
    }
  }
  else
  {
    i=hashf('@');
    cl=look_up(i);
    if (w is an integer)
    {
      c='#';
      p=position(i, c);
      c3=en_prefix(w);
      if (p>0)
      {
        c2=en_prefix(p);
        mts(i, p);
        output cl, c2, c3;
      }
      else
      {
        c2=en_prefix(1);
        output cl, c2;
        output '#' in primitive form;
        output c3;
      }
    }
    else
    {
      c='&';
      p=position(i, c);
      defloat(w, $f_1$, $f_2$);
      c3=en_prefix(f1);
      c4=en_prefix(f2);
      if (p>0)
      {
        c2=en_prefix(p);
        mts(i, p);
        output cl, c2, c3, and c4;
      }
      else
      {
        c2=en_prefix(1);
        output cl and c2;
        output '&' in primitive form;
        output c3 and c4;
      }
    }
  }
}
Decompress:
  {
    c=decode; i=hashf(c);
    if (c='@')
    {
      read a code word p;
      d=de_prefix(p);
      if (d=1)
      {
        read a word from the input buffer in
        primitive form;
        cl=w[0];
        switch cl
        {
          case '#':        read a code word p;
                           dl=de_prefix(p);
                           inserts(hashf('@'), '#');
          case '&':        output dl;
                           read a code word p;
                           dl=de_prefix(p);
                           read a code word p;
                           d2=de_prefix(p);
                           f=float(dl, d2);
                           inserts(hashf('@'), '&');
                           output f;
          case 'a'~'z':    insertf(hashf(cl), w);
          default:         output w;
                           inserts(hashf('@'), w);
                           output w;
        }
      }
      else
      {
        mts(i, d);
        w=word(i, 2);
        output w;
      }
    }
    else
    {
      read a code word p;
      dl=de_prefix(p);
      mtf(i, dl);
      w=word(i, 1);
      output w;
    }
  }
```