COMMUNICATIONS
RESEARCH CENTRE
CENTRE DE RECHERCHE
SUR LES COMMUNICATIONS

Trisignal Data Compression Optimization Study

Final Report

Dr. Michael Sablatash
Communications Research Centre
3701 Carling Avenue
Ottawa, Canada K2H 8S2
Telephone: (613) 998-2061
Fax: (613) 990-0316

November 30, 1994

FINAL REPORT FOR TRISIGNAL COMMUNICATIONS

1.0 Introduction

The three questions on data compaction provided by Trisignal Communications in their research projection description of Aug. 31, 1994, and the discussion of the emphasis to be given to the topics in the letter by J. Belleau dated Nov.7, 1994, are addressed here. Recommendations for short-term and long-term activities are given.

For ease of reference the questions are repeated below:

Question 1: What is the relative performance of Lempel-Ziv algorithm compared to other adaptive compression techniques available today?

Question 2: What are the performances of the techniques studied in 1 when working with Lempel-Ziv compressed files?

Question 3: What are the possible modifications that can be done on Lempel-Ziv algorithm to improve its performances?

The letter of Nov.7 also emphasizes giving more attention, in the short term, to methods which will provide improvements to Lempel-Ziv techniques, even slight ones, than to those requiring large efforts from Trisignal's engineering team.

It was also emphasized that the compaction algorithms had to run on processors of limited CPU and memory capacity. Consequently, processing speed and memory requirements are criteria that should be considered when comparing the performance of candidate algorithms.

Refererences from the interim report, as well as some new ones, were studied and carefully considered to help provide answers to the questions and address other concerns and constraints, and to arrive at recommendations. The numbers of those references which appeared in the interim report (i.e., references [1] to [50]) are the same as those used here, and this reference list is **not** repeated in the current report. New references (i.e., those references beginning at [51]) are listed in the reference section at the end of this report.

Since the LZ algorithm referred to in question 1 consists of two families emanating from the LZ77 and LZ78 algorithms, performances of all of these, as well as statistically-based adaptive algorithms, were studied. A few of the best in each category were selected and comparisons among all those selected are summarized in this report. References to performance comparisons among others are given.

The book "Text Compression", by T.C.Bell, J.G.Cleary and I.H.Witten, published in 1990, and cited as reference 25 in the interim report, contains excellent descriptions and quite thorough comparisons of the performances of most of the compaction techniques up to 1990. For this short project it is not reasonable to try to improve on these descriptions and comparisons. Consequently, we provide a synopsis of the results from that work as a starting point. A number of other studies of performance comparisons to 1990 are also briefly discussed. Comparative performance studies of compaction algorithms reported since 1990 as thorough and systematic as those reported in [25] have not been found, so information from papers has been used to provide these as well as possible. In this way question 1 has been been addressed.

There is not as much information available relevant to question 2, so information before and after 1990 is not distinguished in responding to it.

Question 3 seems to be second (if not first) in priority. Because of the rather comprehensive information available in [25] on the subject of this question as well, this report first provides a discussion of this question up to 1990 (based on [25]), and then the period after 1990 is covered based on a selection of more recent papers.

2.0 Response to Question 1

Answer and Recommendation: In terms of compression performance, the original Lempel-Ziv algorithms are significantly inferior to many of the techniques available today. However, increases in performance have to be balanced against increases in implementational complexity.

In decreasing order of priority, the best choices for the short term (0-2 years) evaluation and development are (1) LZFG[25]; (2) Type II algorithms in [51]; and (3) modified LZ scheme in [38].

Dictionary schemes, such as those recommended above in the LZ family, are still useful for their speed and economy of memory, but it has been predicted [25] that the improvement of computing technology will make high-performance statistical schemes feasible for general use, and that dictionary coding will eventually be of historical interest only. Quite a number of references cited earlier describe schemes which show that these predictions are coming true. Therefore, it is recommended that the following techniques, in roughly decreasing order of perceived priority, be studied and considered for future implemetation and testing to meet future customer demands competitively: (1) Fast PPM, QA/Rice [64, 65]; (2) PPMC [25]; (3) ZLWASPC[14]; (4) the order 3-1-and-0 and order-2 and-0 algorithms of [20]; (5) rule-based methods [37, 62]; (6) context modeling; (7) combined LZ, arithmetic and Lynch-Davisson coding [44]; (8) ANTF [40]; (9) WDLZW and other word-based schemes [33, 40, 63]; and (10) finite-state modeling [20, 25].

General Background and Discussion Pertaining to Question 1

To 1990 the most comprehensive performance comparisons among data compaction techniques belonging to the LZ77 and LZ78 families, as well as statistically-based adaptive techniques, have been provided in the book by Bell, Cleary and Witten[25]. Readily understandable explanations of the principles underlying each of the techniques, accompanied by examples to illustrate how they work, are given. A great deal of effort would be necessary to improve on these. Therefore, the discussion of question 1 to 1990 mostly summarizes descriptions of techniques and comparisons among them given in [25], and recommends those which seem most useful and promising for Trisignal Communications.

A similar presentation of descriptions, performance comparisons and recommendations is given for techniques to 1990 which do not appear in [25], and for techniques from that time to the present, also classified under the adaptive dictionary coding LZ77 and LZ78 families, and statistically-based adaptive techniques.

The statistically-based techniques include adaptive versions of Huffman coding and arithmetic coding, context modeling and state-based modeling.

Almost all practical adaptive dictionary encoders are variations within the LZ77 and LZ78 families. Only the essence of the principles underlying these are given here. Excellent explanations and examples are provided in [25]. Mathematical treatments and concise algorithms may be found in [12], [13], [29], [34] and [39]. Algorithms in C are given in [26]. Algorithms useful for implementations are also given in [9] and [42]. Rigorous theoretical trearments are developed in [4], [6], [7], [8], [11], [12], [13], [15], [19], [21], [29], [31], [34], [35],[36], [39], [44], [47], [50] and [51].

The essence of all LZ-based coding is that phrases are replaced with pointers to where they have occurred earlier in the text. The original LZ papers were highly theoretical and difficult to understand. Other authors have given more accessible descriptions. These descriptions often include innovative variants, so LZ coding is not a single ,well-defined algorithm, but an evolving family of algorithms. The LZ77 family is characterized by pointers which reach back to indicate a substring in the previous N characters, while the pointers in the LZ78 family can reach back to indicate a previously parsed phrase, and there is no restriction on how far back a pointer may reach. A summary of principal LZ variations is given in Table 8-2 of [25], and is reproduced as Table 1 below for ease of reference. Descriptions of these LZ variations, with illustrative examples and figures, and advantages and disadvantages, are provided in [25], Chap. 8. The compression performances of the six techniques in the LZ77 family, and the six techniques in the LZ78 family were compared using 14 different files, and displayed using bar graphs[25]. From best to worst the LZ77 family members ranked as follows: LZB, LZH, LZSS, LZ77, LZJ' and LZR. The LZ78 family members ranked as follows: LZFG, LZMW, LZT, LZ78, LZC and LZW. (LZJ' is a slight variant of LZJ described in [25]). The compression performances of the six members of the statistical family were also compared using the same 14 files. These six adaptive statistical techniques are described in Chapters 4-7 of [25]. They are designated by mnemonic acronyms and ranked in decreasing order of compression performance as follows: PPMC, DMC, WORD, MTF, DAFC and HUFF. PPMC, DAFC and WORD are finite-context models. The HUFF scheme uses a simple order-0 model, requiring only a few hundreds of by\tes of memory. MTF stands for Move to Front coding, and works on the principle that the appearance of a word in the input makes that word more likely to occur in the near future. DMC stands for "dynamic Markov coding". The basic idea of DMC is to maintain frequency counts for each transition in the current finite-state model, and to "clone" a state when a related transition becomes sufficiently popular. Finally, the three techniques from the three families which yielded the best compression performance for each family were compared in the last round of the "tournament". PPMC, for almost every one of the fourteen files, achieved the best compression, but LZFG and, to a lesser extent, LZB, are competitive, and consume considerably less time and memory than PPMC.

TABLE 1 SUMMARY OF PRINCIPAL LZ VARIATIONS ACCORDING TO [25]

LZ77	Ziv and Lempel (1977)	Pointers and characters alternate Pointers indicate a substring in the previous N characters
LZR	Rodeh et al. (1981)	Pointers and characters alternate Pointers indicate a substring anywhere in the previous characters
LZSS	Bell (1986)	Pointers and characters are distinguished by a flag bit Pointers indicate a substring in the previous N characters
LZB	Bell(1987)	Same as LZSS, except a different coding is used for pointers
LZH	Brent(1987)	Same as LZSS, except Huffman coding is used for pointers on a second pass
LZ78	Ziv and Lempel (1978)	Pointers and characters alternate Pointers indicate a previously parsed substring
LZW	Welch(1984)	The output contains pointers only Pointers indicate a previously parsed substring Pointers are of fixed size
LZC	Thomas et al. (1985)	The output contains pointers only Pointers indicate a previously parsed substring
LZT	Tischer(1987)	Same as LZC but with phrases in a LRU list
LZMW	Miller and Wegman(1984)	Same as LZT but phrases are built by concatenating the previous two phrases
LZJ	Jakobsson(1985)	The output contains pointers only Pointers indicate a substring anywhere in the previous characters
LZFG	Fiala and Greene (1989)	Pointers select a node in a trie Strings in the trie are from a sliding window

Other performance criteria besides compression (more accurately, compaction) are speed, memory and convergence. To 1990 the latter three factors are also compared in [25] for selected examples from the three families. There are relationships and tradeoffs among compression, speed and memory. Table 2 shows the requirements of techniques from each of the families for memory and speed. Speeds are given for coding the 769-Kbyte file "book1" on a 1-MIP VAX 11/780. Among the statistical schemes PPMC and WORD give excellent compaction for a moderate cost. In comparison, DMC is particularly uneconomical. The speeds of PPMC and WORD are not high enough for ISDN applications. Note that encoding and decoding require similar resources for the statistical methods. On the other hand, LZFG is much faster, and requires less memory, especially for decoding. For LZB decoding is particularly cheap, but encoding is slow. Among all the methods LZFG appears to be the only one which could be useful for ISDN applications, because its speed of 6,000 characters per second for encoding equals 48,000 bits per second, and its speed of decoding of 11,000 characters per second is equivalent to 88,000 bits per second. These figures are close enough to the ISDN speeds of 64,000 bits per second and 128,000 bits per second that faster processing can be obtained to meet these requirements.[1] A final performance criterion should be considered. This is convergence[25]. It was found that LZ78 converged very slowly, while DMC and PPMA (an earlier version of PPMC) converged much faster. Memory was not limited. It should be noted that the fact that LZ78 and LZR are known to achieve compression converging to the entropy of an ergodic source as the size of the input increases.is of little benefit in practice because convergence is far too slow to achieve the limiting performance for practical applications.

TABLE 2 MEMORY AND SPEED REQUIREMENTS OF SOME COMPRESSION SCHEMES FOR "book1"

Scheme	Memory (Kbytes)		Speed (char. per sec.)	
	Encoding	Decoding	Encoding	Decoding
PPMC	500	500	2,000	2,000
WORD	500	500	3,000	3,000
DMC	5,000	5,000	1,000	1,000
LZFG	186	130	6,000	11,000
LZB	64	8	600	16,000

Many further details about performance are discussed in [25], and the reader is encouraged to study these for a more complete understanding.

Some performance studies to 1990 for schemes not appearing in [25] are examined next. It is somewhat frustrating and disappointing that performance studies of data compaction methods documented in sources other than [25] are not as thorough. Nevertheless it may be that some of these will yield significant improvements over those examined so far.

The performances of ten schemes are presented in [14]. They are:

1. UNIX pack, a standard implementation of Huffman coding (PACK).

[1]Of course the speeds quoted here are based on a 1-MIP VAX 11/780. Many of today's high performance microprocessors can significantly exceed this performance, particularly for bit-level operations such as the manipulation of bit strings.

2. UNIX compact, a standard implementation of adaptive Huffman coding (CPACT).
3. 256-character move-to-front list using the full 8-bit ASCII alphabet with the output byte stream compressed using Huffman coding (MTF) [52].
4. Bentley-Sleator-Tarjan-Wei coding with list length 239 and a maximum word length of 16 characters (BSTW) [52], [53].
5. Miller and Wegman's A2 algorithm [54].
6. UNIX compress, a standard implementation of the LZ compression algorithm. Two versions of compress are listed: one ("compress-b12") using a dictionary size of 4K, and the other ("compress-b16") using a dictionary size of 64K.
7. LZ coding where dictionary entries are arithmetically coded (ZLWA).
8. LZ with arithmetic coding and string estension (ZLWAS).
9. LZ with arithmetic coding, string extension and the predictive strategy described in [14].
10. LZ with arithmetic, string extension, predictive strategy and conditional dictionary entry (ZLWASPC).

Test text files were chosen representing the most common kinds of text that are stored and communicated on computer networks. The kinds of text files chosen are:

1. A file of student essays (file essays) to show compression performance on normal English text.
2. A Pascal source code file (file Pascal) to show compression performance on computer language source code.
3. The TEX source to the complete TEX manual (file TEX) to show compression performance on a document containing text and formatting language commands.
4. A file of USENET news articles (file News) to show compression on items that are commonly transmitted across computer networks.
5. A file of three repeated characters (file abc) to highlight the effect of code doubling on repetitive files.
6. An executable program (file csh) to show compression performance on a binary file.

Each of these test files was compressed using the ten methods. Table 3 shows the percentage compression for the test files using each of the ten methods.

TABLE 3: PERCENTAGE COMPRESSION FOR TEST FILES

Compression Algorithm	File Essays	File Pascal	File TEX	File News	File abc	File csh
PACK	47	41	38	33	75	21
CPACT	47	41	38	33	75	21
MTF	41	38	35	31	N.A.	19
BSTW	61	75	64	48	N.A.	32
MWA2	60	N.A.	N.A.	47	N.A.	N.A.
ZL12	54	63	52	33	99.2	26
ZL16	60	70	61	46	99.2	36
ZLWA	61	71	62	48	99.2	38
ZLWAS	63	83	66	51	99.9	41
ZLWASP	63	84	67	54	99.9	45
ZLWASPC	66	84	69	55	99.9	45

The percentage compressions, in only very gradually decreasing order,for ZLWASPC, ZLWASP,and ZLWAS, vary by a maximum of 4% - and this in only one instance. The performances of ZLWA and ZL16 are very close, showing that ZL coding with arithmetic coding performance gains may be almost completely compensated by increasing the dictionary size in ZL coding - with the additional benefit of increased processing speed. Except for the csh file the BSTW method is superior to the ZL16 and the ZLWA methods. As discussed in [14] in more detail, string extension greatly increases the number of code words. In all but one of the test files, however, it reduces the number of coder emissions by enough so that compression is improved. For most of the test files, a conditional dictionary (as used in ZLWASPC) both greatly reduces the number of dictionary entries and reduces the number of emissions. In [14] it was found that the ZLWASPC encoder runs reasonably quickly, compressing about 2000 bytes pere second on a Sun-3. This speed could be greatly improved by using an optimized arithmetic encoder, removing debugging code and removing the statistics collection code. In comparison the well-optimized ZL16 encoder runs at about 30,000 bytes per second on a Sun-3. Table 4 shows the run times obtained in [14] for several of the tested algorithms on file "essays".

TABLE 4: RUN TIMES TO COMPRESS FILE "ESSAYS"

Algorithm	Run Time (Seconds)
PACK	4
CPACT	40
ZL16	6
ZLWASPC	85

Fast parallel VLSI implementations of ZL encoders have been developed. Faster arithmetic coders in both hardware and software are available than used in the implementations of [14], thus increasing the speed of the enhanced encoders. Without such improvements in speed the ZL16 seems like the best choice among those compared in [14], for Trisignal's shorter term needs. If a very fast arithmetic encoder were developed ZLWASPC would be the best choice, and would offer considerable improvement over ZL16.

Next, the performances of compaction schemes in [38] were examined. The design criteria for real time source encoders for communications used in [38] are worth noting. These are:

(1) memory;
(2) processing speed;
(3) compression performance;
(4) delay;
(5) compression of non-stationary sources;
(6) avoidance of data expansion;
(7) interaction with the ARQ system;
(8) error propagation; and
(9) implementation complexity.

Three hybrid schemes were described. These are:

(1) concatenation, examples of which are the use of run length encoding prior to Huffman coding, and the concatenation of LZ and arithmetic coding;

(2) initial selection, in which part of the data is initially compressed using several coding schemes in parallel, and the code achieving better compression selected (the popular "zip" software is an example of this approach); and

(3) dynamic selection, in which several coding schemes are operated in parallel, and a decision is made periodically on which code to select.

The last approach is more appropriate in the case of a communications system than the second one as the source characteristics are likely to be non-stationary. Four commercially-developed adaptive source coding schemes were tested off-line, and a comparison made on the basis of performance and complexity. These were:

(1) a variable length encoding scheme, in which a dynamic selection is made from several codes with predetermined length distributions on the basis of frequency distribution, with memory requirement of about 1 kbyte;

(2) a hybrid run length, first-order Markov scheme, employing run length encoding, followed by adaptive variable length encoding based on first order statistics, with memory requirement of about 36 kbytes, and a limit on the total number of conditional probability estimates;

(3) a hybrid first order Markov process with sliding window, empoying both variable length encoding based on first order statistics, and sliding window string encoding, with total memory requirement of about 24 kbytes; and

(4) LZ, with a simple cyclic scheduling algorithm (CSA) for storage recovery, employing the dynamic dictionary LZ algorithm, with the character extension improvement and CSA storage recovery heuristic, and a memory requirement which is dependent on the dictionary size, and may be as low as 3 kbytes for a 512 entry dictionary or 28 kbytes for a 2048 entry dictionary.

Of the design criteria for real time adaptive source encoders memory requirements and processing time are probably the most critical. The LZ algorithm with the character extension improvement and a simple storage recovery heuristic provides performance equivalent to first-order variable length encoding algorithms, at a fraction of the processing time.

The three tables 5-7, below provide comparisons among the schemes based on compression ratios and on the memory size for scheme 4.

TABLE 5. COMPARISON OF COMPRESSION RATIO ACHIEVED BY THE MODIFIED ZIV-LEMPEL ALGORITHM WITH A HYBRID RLC/VL SOURCE CODE AND AN IDEAL VARIABLE LENGTH CODE

Data	0th Order	Scheme (1): Hybrid 1	Scheme (4): ZL (512)
Program Source	1.7	1.7	2.0
Assembly Lang.	1.8	1.7	2.2
Text	1.9	1.8	2.1
Program obj/exe	1.8	1.8	2.1
Mac/GEM	1.5	1.7	2.3
Spread/Database	1.9	1.6	2.0
Compuserve	1.4	1.3	1.3
Processing time: (normalized)	--	2.7	1.0

The performance studies in [38] show that scheme 4, the modified LZ algorithm, is the best design choice among the four schemes compared, as can be seen from an examination of Tables 5-7. Further discussion supporting this choice is given in [38].

Three different word based data compaction schemes are described in [40]. Comparisons of performances with those of the LZW and a 4th order arithmetic encoding are made there. The three word based schemes are Move to Front (MTF), Frequency to Front (FTF) and Alpha-Numeric to Front (ANTF). The ANTF word based scheme outperforms the LZW algorithm in compression ratio when the word dictionary contains 256 or more indices. It also outperforms the MTF and FTF algorithms for 7 out of 9 files for FTF, and 8 out 9 files for MTF. Implementation details are not given in [40]. Further study to determine speed and memory requirements of the ANTF algorithm is needed to determine its appropriateness for Trisignal's applications.

A model called "Telcor" [55] combines concepts from both Huffman and LZ. Like LZ it uses the fact that certain character strings appear more frequently in a data stream than do other strings. Like Huffman, Telcor adapts so that the more frequently a character is used, the greater the degree of data compression that is achieved. Telcor uses up to 2,048 tables, each table containing up to 16 different characters. It is claimed in [55] that

TABLE 6. COMPARISON OF COMPRESSION RATIO ACHIEVED BY MODIFIED ZIV-LEMPEL ALGORITHM WITH TWO HYBRID SOURCE CODING SCHEMES

Data	Scheme (2): Hybrid 2	Scheme (3): Hybrid 3	Scheme (4): ZL (2048)
Program source	2.6	2.9	2.6
Assembly lang.	2.7	2.8	2.7
Text	2.5	2.8	2.8
Program obj/exe	1.9	1.7	2.0
Mac/GEM	2.0	1.9	2.2
Spread/Database	2.0	2.2	2.2
Compuserve	1.2	1.2	1.3
Processing time: (normalized)	2.8	10.	1.0

TABLE 7. COMPRESSION RATIOS FOR MODIFIED ZIV LEMPEL ALGORITHM SHOWING THE EFFECTS OF VARYING DICTIONARY.SIZE

	Dictionary size.			
Data type	512	1024	2046	4096
Program source	2.0	2.4	2.6	2.7
Assembly lang.	2.2	2.5	2.7	2.8
Text	2.1	2.4	2.5	2.5
Program obj/exe	2.0	2.0	2.0	2.0
Mac/GEM	2.3	2.3	2.2	2.2
Spread/Database	2.0	2.2	2.2	2.2
Compuserve	1.3	1.3	1.3	1.3

Telcor achieved the fastest effective data transmission rate of any data compression technique. Implementation details and performance comparison tests are not given, although a description of how the system works is provided. The system description is interesting enough that further details should be obtained and the algorithm tried. As noted in [9], however, taking the LZW codes and running them through an adaptive

Huffman coder generally yields a few more percentage points of compression, but at the cost of considerably more complexity in the code as well as quite a bit more run time. The available technology must be carefully examined to determine whether such enhancements are now cost effective.

Summarizing my recommendations from among the choices available to 1990, the best choice seems to be LZFG in the short term (1 to 2 years), and ZLWASPC and PPMC in the longer term, if suitably high speed and cost-effective technology can be implemented for the latter. The modified LZ algorithm described in [38] and word-based schemes[2] such as ANTF also appear to merit scrutiny in the longer term.

It is time to turn to techniques from the beginning of 1990 to the present (Nov. 1994). Here we summarize and compare their performances, and make some recommendations from among these.

The improved sliding window data compression algorithm described in Bender and Wolf's two papers [12,13] offers improvements in compression ratio of about 10% over LZ77 and between about 20% and 100% over LZW. It appears that it may be worthwhile to implement and test this algorithm in the short term.

On the other hand the excellent unified algorithmic and analytic description in Yokoo's paper [15], which should be extended to describe, unify, and provide insights into other methods as well, does not appear to yield an algorithm with a sufficiently improved compression ratio to be sufficiently profitable for implementation. Pursuing Yokoo's insightful analysis, however, may yield the understanding to develop new algorithms with better performance. This seems like a good long-term project.

The coding method in [21] based on LZW adopting the least recently used deletion heuristic yields no more than about 5% improvement over LZW, so it appears not to be worthwhile to pursue.

The word-based dynamic algorithm WDLZW in [33] offers extremely good compression for text while still being able to compress other data forms. Except for graphs and binary data it outperforms LZFG by between about 5% and 62%. It yielded better compression

[2]A note of caution on word based schemes is that they may be less generally applicable than those schemes that operate at the bit level.

ratios than LZW for all 8 files tested, often by large factors. The detailed comparative results for compression ratios based on experimental results are presented in [33]. The implementation described in the paper used 239.4K of memory, but the method for using memory was rather wasteful and it was noted that a wide range of strategies were available to use it more efficiently. A sequential search strategy, which is rather slow, was used in the implementation. Improved search strategies are known but were not implemented in the paper. Thus, this method is quite promising for Trisignal's needs, but would take some to optimize, so is not suitable for short-term development.

Two enhancements are described in [56] which improve LZW and LZC (the UNIX compress method) by up to 8% - i.e., a file that is compressed to 49% of its original size by the standard LZW algorithm is compressed to about 41% of its size by the improved algorithm. The two ways of improving LZC are loading the dictionary at a faster rate, and phasing in increased lengths of binary numbers gradually. It was noted that it had not been easy to find easily implementable methods for improving the performance of LZC, especially when one imposes the additional constraint that the execution time requirements should not be severely affected. This indicates, in the author's opinion, diminishing returns to improvements in LZW and LZC. The methods of this paper may be worthwhile implementing and testing in the short term, although the gains are relatively modest.

In the abstract [57] it was found in a test of five 100-Kbyte text and executable files that the compression ratios of V.42bis were markedly inferior to those of several other algorithms, notably the Cleary-Witten adaptive Markov code and the Fiala-Greene Ziv-Lempel variant. However, according to [57], since all the better compression methods have much larger data structures, none is superior to V.42bis for use in a contemporary modem. A few sentences supporting this are given in the abstract, and the availability of a technical report from the author elaborating upon it is asseverated. This reference is noted here because it raises the possibility that significant improvements in compaction methods require major redesign of modems.

In [51] it is shown that there is a redundancy in the LZ coding algorithm since there exists a large number of the same substrings in the reference part of the buffer. A method is proposed to improve the coding by eliminating that redundancy. In the proposed scheme, using an algorithm designated as type I, matched string search step is required in the decoding, which has not been used in the conventional scheme, but which eliminates the advantage of fast decoding of the LZ algorithm. To keep the advantage of fast decoding, another method employing an algorithm called type II is proposed, in which the matched string search is not required in the decoding, at the cost of a slight deterioration in the compression ratio. Tabular comparison of two variants of each of the two algorithms with the UNIX compress algorithm, using ten different test files, place in evidence compression ratio improvements of from 5 to 50% over the latter. The many details of the theory, algorithms and performance are given in [51]. The methods are compared with the Bender-Wolf method [12, 13], abbreviated as the LZBW method, and the Morita-Kobayashi method [34], abbreviated as the LZMK method, which is similar to the Fiala-Greene method [58].

The LZBW method does not completely eliminate the redundancy. A problem is then left that an improvement cannot be expected when a string composed of a single kind of symbol (aaaa.....) is to be encoded.

LZMK is an excellent coding method based on the Patricia tree data structure used for the longest matched string search. The type II algorithm is used in this method in the decomposition of the string into words. The substrings obtained from the reference part are represented in the Patricia tree, and a codeword is composed of a node number and a location along the arc between the node and its parent. By this technique the representation for the length of the matched string can be shorter. The representation for its position may become longer (letting the number of words that can be referred to be D, the total number of intermediate nodes and leaves is 2D-1 at the maximum). Because of this the performance of LZMK may be deteriorated compared with the conventional scheme for some data.

Since the techniques in [51] improve the compression ratio by 5 to 50% over UNIX compress, and are presented in considerable detail, it is recommended that Trisignal consider implementing and testing the faster two type II algorithms in the short term. It appears that these methods are more promising than those proposed in [12, 13 and 34].

An extremely fast Ziv-Lempel algorithm is described in [59]. It was new, simple and locally adaptive, and belongs to the LZ77 class. The algorithm, called LZRW1,almost halves the size of text files, uses 16K of memory, and requires about 13 machine instructions to compress and about 4 machine instructions to decompress each byte. This results in speeds of about 77K and 250K bytes per second on a one MIPS machine. The algorithm runs in linear time and has a good worst case running time. It adapts quickly and has a negligible initialization overhead, making it fast and efficient for small blocks of data as well as large ones. LZRW1 has a compression ratio worse than UNIX compress, but runs four times faster, making it possibly the fastest adaptive text compression algorithm yet. Thus, if speed is more important than the loss of 10% compression ratio, LZRW1 would be a very good short term choice.

An awareness of the speeds which are being achieved for compression and decompression within about the last two years is given in [43] and [60]. In [43] the LZ77 algorithm is implemented in a high-speed VLSI design yielding a compression rate of 13.3 million bytes per second at a clock rate of 40 MHz. In [60] the LZ78 algorithm is implemented, using a content-addressable memory (CAM), in a single chip CMOS processor performing compatible compression and decompression at 10 Mbps. Where high speed is an overriding requirement these two examples provide an understanding of what is now practically feasible for the LZ77 and LZ78 algorithms.

In [44] a technique is presented that combines the merits of LZ coding and their variants with those of the arithmetic and Lynch-Davisson coding, with the desirable attributes of a small memory size requirement and high speed. The technique can be easily realized by a parallel-sequential method, which is very important for hardware implementation. It also provides good data compression in the case of comparatively small block lengths. This code may be a worthwhile candidate for design, implementation and testing in the long-term. Quite a lot of effort is required to go from this rather theoretical paper to practical realization.

An algorithm designed by Rice and recently implemented in VLSI [61] may be particularly appropriate for applications in which small data packets are required. The algorithm was designed for image transmission in space communications applications. However, its performance was compared, using three image suites as a basis of comparison, in [61] with LZW, adaptive Huffman coding, arithmetic coding, and two-pass Huffman based lossless JPEG coding, the first three of which have often been used for compression of text data. The resulting compression performances are exhibited in comparative graphical illustrations and tables. The Rice algorithm, with added enhancements in a VLSI implementation called the USES chip, achieved the highest compression in 24 out of the 26 images in the USC miscellaneous monochrome images. Using the same pre-processor as is used in the USES chip, the enhanced Rice algorithm had a performance comparable to adaptive Huffman coding. USES, however, outperformed all the algorithms on small images - indicative of the Rice algorithm's rapid adaptation to scene statistics. The Rice algorithm does not use a history of strings or statistics, but operates on a block by block basis. This characteristic of not operating on statistics of past data results in the implementations not requiring external memory and allowing the Rice algorithm to be used on small packets of data. If a data packet should be lost, the Rice-based encoder and decoder could continue to operate without needing to reset the system. The first Rice algorithm based chip set was described as well as a second generation encoder which adds new low entropy coding options and on chip packetizing. The first chip set has become a commercial product and the new encoder has been baselined for the LandSat 7 satellite project. Indeed, it appears that no other algorithm has been tailored for packetized data, and yet this appears to have important commercial potential. However, in its current form, the techniques is probably too speciallized towards its given application to be appropriate for Trisignal.

In [37] and [62] a rule-based encoder (RBE) and decoder (RBD), respectively, are described. Considerable improvement, up to 50%, in the compression ratio (CR) is realized when the analytical method of LZ77 is replaced in the decoder by the nonanalytical rule based pattern recognition method developed in [62]. The proposed decoding algorithm (RBD) is easy to implement. In examples with different string lengths the percentage improvement in the CR obtained by replacing the LZ encoder by the RBE was found to be a minimum of 230% over the LZ78 algorithm [37]. Comparisons with LZ algorithms are given in [37] and [62]. The algorithms appear to be easy to implement, are low in use of resources, and achieve rather large compression. More comprehensive testing is necessary for a completely satisfactory evaluation.

In [63] word-based text compaction algorithms are described. The source alphabet symbols are chosen to be the words of English or, in general, alternate maximal strings of alphanumeric characters and non-alphanumeric characters. The compaction algorithm is able to take advantage of longer-range correlations between words and thus achieve better compression. The large size of the dictionary leads to some imlementation problems, but these are overcome to construct word-based LZW, word-based adaptive Huffman, and word-based context modelling compaction algorithms. The overall result is that performance is consistently better than the UNIX compress program, no matter which word-based method is used. Much further experimentation is needed. For now it is recommended that a simple and fast word-based algorithm be used for text file compression. Of possibilities the word-based LZW algorithm would seem to be the closest fit. Table 8 shows the relative compression performances. The pair of numbers shown for the state-based context compression reflects the fact that the algorithm normally requires initialization with a vocabulary giving the parts of speech for words. In the experiments the initial vocabulary contained all the words used in the first test document (the csh manual description). The upper number in each pair shows compression performance when that vocabulary is used; the lower number shows performance when no vocabulary at all is used - i.e., the algorithm simply assigns words

TABLE 8: COMPARATIVE COMPRESSION PERFORMANCES

		Relative Size After Compression				
	Original Size in bytes	UNIX compress	WB-Adpt Huffman	WB-LZW	State-Based	WB-First Order
On-line manual page for csh	66772	40.4%	29.8%	34.3%	21.6% 29.3%	29.0%
On-line manual page for make	63761	42.7%	34.7%	35.8%	29.5% 33.6%	31.7%
LaTeX file	83106	44.7%	36.1%	32.0%	36.0% 35.2%	33.7%
	24706	39.2%	28.5%	26.4%	32.3% 27.0%	25.3%

to one of five classes as it seems to find appropriate. These word-based algorithms look very promising for text compression.

Context modeling for text compression is discussed in [20]. A summary is given in the Oct. 20, 1994 interim report, and it is thorough enough that the reader is referred to it to grasp the essential ideas. Adaptive schemes have emerged as very promising approaches to compressing text. They are the most promising alternative to LZ methods. Corroborating [25], Algorithm PPMC represents the state of the art in context modeling. PPMC provides very good compression (approximately 30% on average), but has a large memory requirement (500 Kbytes) and executes slowly. Comparisons with other schemes are made in [20] and [25]. An alternative to PPMC called the order 3-1-and-0 method achieves much of the compression performance of PPMC without the large memory requirement (in fact it requires only one-fifth as much run-time memory). This method has the additional advantage of executing much faster. Compression factors of less than 2.4 bits per character are achieved for source code files and less than 2.8 bits per character for a large variety of file types using less than 100Kbytes of internal memory. The order-2-and-0 algorithm of [20] achieves respectable compression using only 48 Kbytes of internal memory. The compression performance of this method is superior to that of "compress" and uses less than 10% as much memory.

In [64] it is observed that the best compression results are obtained from the use of high-order models in conjunction with statistical coding techniques. The best compression reported in the literature comes from the PPM (prediction by partial matching) method of Cleary and Witten; the most widely used implementation is Moffat's PPMC. The PPM methods use adaptive context models with a fixed maximum order, and arithmetic coding for the coder. In [64] it is shown that one can obtain significantly faster compression with only a small loss of compression efficiency by modifying both the modeling and coding aspects of PPM. The important idea is to concentrate computer resources where they are needed for good compression while using simplifying approximations where they cause only slight degradation of compression performance.
On the modeling side, the explicit use of escape symbols is eliminated, approximate probability estimation is used, and the repeated-symbol-exclusion mechanism. In the coder the time-consuming arithmetic coding step is replaced with various combinations of quasi-arithmetic coding and simple prefix codes from the Rice family [65]. Quasi-arithmetic coding is a variation of arithmetic coding that uses lookup tables after performing all the arithmetic ahead of time. The computations are done to low precision to keep the table sizes manageable. A Fast PPM method described in [64] outcompresses the compress program on all text files. Fast PPM with quasi-arithmetic coding gives compression performance comparable to that of PPMC, especially for larger files. Fast PPM, even using quasi- arithmetic coding alone, is always faster than PPMC; the version that uses some Rice coding is nearly twice as fast as PPMC. Table 9 shows the results of experimental comparisons of the Fast PPM method with PPMC and with the UNIX compress program.

Except for a summary of recommendations this completes the brief discussions of relative performance of LZ algorithms and other adaptive compression techniques, in response to question one. The next task is to distill the above performance comparisons to provide recommendations for short term (i.e., immediate, with about a one-to two-year maximum development time) and longer-term .

TABLE 9: COMPRESSION AND ENCODING THROUGHPUT ON THE TEN TEXT FILES IN THE CALGARY CORPUS. COMPRESSED SIZES ARE EXPRESSED IN BITS PER INPUT SYMBOL. ENCODING THROUGHPUT IS EXPRESSED IN THOUSANDS OF UNCOMPRESSED BYTES PER SECOND.

	Compressed Size				Encoding Throughput			
	Fast QA	PPM QA/Rice	PPMC	Compress	Fast QA	PPM QA/Rice	PPMC	Compress
bib	2.19	2.32	2.12	3.35	23.2	29.0	16.4	111.3
book 1	2.51	2.58	2.52	3.46	23.2	30.1	18.5	108.3
book 2	2.29	2.41	2.28	3.28	23.5	30.6	18.1	111.1
news	2.78	2.94	2.77	3.86	16.9	23.5	12.6	99.2
paper 1	2.62	2.83	2.48	3.77	17.8	24.7	13.6	106.3
paper 2	2.51	2.67	2.46	3.52	21.1	26.5	15.2	102.7
progc	2.68	2.92	2.49	3.87	16.9	23.6	12.4	99.0
progl	1.99	2.16	1.87	3.03	24.8	31.7	18.4	119.4
progp	1.96	2.17	1.82	3.11	22.0	31.1	16.5	98.8
trans	1.88	2.09	1.75	3.27	23.7	32.1	18.0	117.1

3.0 Response to Question 2

Answer and Recommendation: This question has been perceived by Trisignal as being difficult and long term. We concur with this perception. The issue is not one of modifying LZ type algorithms internally, as addressed in question 3, or replacing them by other adaptive algorithms, as addressed in question 1, but one of cascading a compaction scheme following the LZ scheme to further compact LZ-compacted files. It is concluded that cascading the LZW algorithm with arithmetic coding yields a significant increase in compression ratio. Since speed and memory are not

discussed, the practicality for Trisignal's applications cannot be assessed objectively, but it is our opinion that memory will be larger than acceptable, and speed not high enough, unless dedicated hardware is designed and implemented. Therefore, it is recommended that Trisignal remain aware and informed about the developments in relevant technology, and watch for cost-effective opportunities for implementation and deployment.

General Background and Discussion Pertaining to Question 2

The concatenation of additional compaction with an LZ algorithm is addressed in only two papers among all the approximately two hundred references which we have examined. These are references [16] and [71], and of these two only [71] provides a good technical discussion and results of substance. Reference [16] provides only the ideas and concepts for cascading. In [16] the LZ, Huffman-Gallager and adaptive Guazzo algorithms are cascaded and their complementary properties exploited to obtain improved compaction. In [71] LZW is cascaded with arithmetic coding. It is concluded there that cascading the LZW algorithm with arithmetic coding yields a significant increase in compression ratio, as verified by experimental results. Since speed and memory are not discussed, the practicality for Trisignal's applications cannot be assessed objectively, but it is our opinion that memory will be larger than acceptable, and speed not high enough, unless dedicated hardware is designed and implemented.

4.0 Response to Question 3

Answer and Recommendation:There are many possible modifications that can be made to the LZ families of algorithms to improve their performance. Those that correspond to the short-term (0-2 years) recommendations given in response to question 1, seem to be most appropriate to focus company efforts. In decreasing order of priority, these are: (1) the techniques to implement LZFG[25,58]; (2) the techniques to implement the Type II algorithms in [51], and discussed succinctly in item 1 below; and(3) implement LZ algorithm with the CSA storage heuristic described in [38] and discussed succinctly under item 3 below. In the longer term, it is recommended that the LZW coding be enhanced through techniques of using arithmetic coding to represent dictionary entries, improving the speed of the arithmetic coder, using string extension, using a predictive strategy and enhancing it by adding a first order Markov model, using a conditional dictionary entry strategy and enhancing it, to realize ZLWA, ZLWAS, ZLWASP and ZLWASPC schemes, as

discussed in [14], and in items 10-13 below. The combined LZ, arithmetic and Lynch-Davisson coding offers promise, but requires starting from a fairly theoretical description. The WDLZW scheme described in [33] and in item 15 below appears to be quite promising in concept and performance.

General Background and Discussion Pertaining to Question 1

The following are problems with the LZ algrithms and modifications for their improvement that have been tried or identified as plausible.

1. The compression efficiency of the LZ code converges very slowly, often resulting in significantly less than optimal performance for short and moderate lengths of data records. In [51] it is shown that there exists considerable redundancy in the codeword assignment because the same substring appears in different places in the buffer of the LZ encoder. The data structure which eliminates the redundancy in the codeword assignments is described, and then a fast algorithm is shown which makes the decoding easy for the proposed encoding scheme. It is also shown in this paper why the Bender-Wolf method [12, 13] does not completely eliminate the redundancy, and the Morita- Kobayashi method [34, 66] results in a deterioration in compression performance due to its use of the Patricia tree.

2. The problem of transmitting an index of a matched string and an uncompressed next symbol in LZ78 [38] was addressed by Miller and Wegman [54,3,11] and by Welch [48], resulting in LZMW and LZW, respectively.

3. The problem of recovering storage when the dictionary becomes full [38] has received some attention, and a number of solutions proposed. The simplest approaches are to reset the dictionary to its starting condition or to stop adding new strings, both of which have performance penalties; better performance can be obtained by selectively deleting entries. Candidates for deletion include dictionary entries with no dependants; however, some further heuristic is needed. Techniques discussed in [67] are; (i) to delete the least frequently used (LFU) entry, as proposed in [68], or (ii) to delete the least recently used (LRU) entry; both of these techniques require additional processing time and 30-40% more memory. An alternative technique [38] is to delete an entry according to a simple cyclic scheduling algorithm. This approach (CSA) is only fractionally less efficient than the

LFU and LRU heuristics; however, it requires no additional memory and is simpler to implement. The LZ algorithm with LRU or CSA storage recovery heuristics does not suffer from peaks in processing load, as there are no scaling or sorting operations required.

4. Each time a new character is read in, the string table has to be searched for a match. If a match is not found, a new string has to be added to the table. This causes two problems. First, the string table can quickly get very large very. Even if string lengths average as low as 3 or 4 characters each, the overhead of storing a variable length string and its code can easily reach 7 or 8 bytes per code. In addition, the amount of storage needed is indeterminate as it depends on the total length of all the strings. The second problem is the computational overhead resulting from searching for strings. These problems are described in more detail in [9], and solutions to them are also described there. The solution given in [9] to the first problem is the storage of strings as code/character combinations. The details are presented on page 34 of that paper. The problem of consumption of large amounts of storage memory for coding and decoding is addressed in [25] and [46]. The simplest strategy when storage is exhausted is to stop adapting the model. Compression continues with the now static model, which has been constructed from the initial part of the input. Turning off (or limiting) adaptation can lead to deteriorating compression if the initial part of a text is not representative of the whole. An approach that attacks this problem is to clear the memory and begin a new model every time it becomes full. Poor compression will occur immediately after a new model is started, but this is generally offset by the better model obtained later. The effect of starting the model from scratch can be reduced by maintaining a buffer of recent input and using it to construct a new initial model. Also, a new model need not be started as soon as space is exhausted. Instead, when no more storage is available and adapration is necessarily switched off, the amount of compression being achieved could be monitored. A declining compression ratio is used as an indication that the current model is inappropriate, so memory is cleared and a new model begun. All of these approaches are very general, but suffer from regular "hiccups" and the fact that storage is not fully utilized while a model is being built. A more continuous approach is to use a "window" on the text, as for the LZ77 family. This involves maintaining a buffer of the last few thousand characters encoded. The advantage to starting pointers in LZ77 only at the boundary of a previously parsed phrase in a window is discussed in [34]. When a character enters the window (after being encoded) it is used to update the model, and as it leaves, its effect is removed from the model. A slow, but general, way to achieve the effect is to use the

entire window to rebuild the model from scratch every time the window changes (which happens for each character encoded). Clearly each model will be very similar to the previous, and the same result might be obtained with considerably less effort by making small changes to the model. Altternatively, the window effect might be approximated by pruning infrequently used parts of the structure, in the style of the LZJ scheme.

The second problem is solved by using a hashing algorithm to store strings. The details are rather lengthy so the reader is referred to [9]. All the practical compression schemes for the LZ77 and LZ78 families use greedy parsing, and in the majority of cases the most time-consuming task is finding the longest match in the dictionary for the next few characters of input. Each decoding step usually requires only a table lookup to find the string identified by a pointer or index. Here we will look at some data structures that can perform the encoding step efficiently This problem is addressed in [46, 25, 18]. Data structures for locating exact matches, such as search trees and hash tables, are well known, and it transpires that these can be adapted to locate the longest matches needed for encoding. The most demanding dictionary compression schemes are those that are adaptive, use a window, and allow pointers to reference anywhere in the window (LZ77, LZSS, LZB and LZH). A new potential starting for a longest match is introduced with each input character entering the window. Potential starting points are removed at the same rate as characters leaving the window. The data structures discussed below apply to this situation, but can be adapted for other forms of dictionary coding. For further details a book on algorithms and data structures must be consulted. A particularly thorough and comprehensive presentation of longest-match string searching for LZ compression is given in [18]. Eight data structures that can be used to accelerate the searching are presented there, and evaluated analytically and empirically, indicating the tradeoffs available among average compression speed, worst-case behaviour and memory requirements. A binary search tree gives good performance for only modest memory requirements, and splaying can be applied to avoid worst case performance. The trie, which would traditionally be the method of choice for this situation, does not appear to offer a particularlly useful compromise between speed and storage, since it is outperformed on both counts by a binary search tree. The list1 method offers particularly good performance considering it requires very little storage. A hash table appears to be a contender, particularly with English text, and there is scope to improve its performance with a larger hash table, and by experimenting with the hash function and search strategy. Worst case performance for most of the data structures is caused by files that contain long repeated substrings, such as an image that contains long runs of zero bits. Poor performance can be avoided by limiting the length of a match, the extent of searching, and/or using a data structure, such as a splay tree, that does not depend on substrings of the text being random.

5. Once a dictionary has been chosen, there is more than one way to choose which phrases in the input text will be relaced by indexes to the dictionary. The task of splitting the text into phrases for coding is called parsing. The most practical approach is greedy parsing [25, 46]. Practical experiments have shown that optimal encoding can be two to three times slower than greedy encoding, but improves compression by only a few percent. The LFF (longest fragment first) and bounded buffer approximations improve the compression ratio a little less, but also require less time for coding. In practice the small improvement in compression is usually outweighed by the extra coding time and effort of implementation, so the greedy approach is by far the most popular. Most dictionary compression schemes concentrate on choosing the set M of phrases in the dictionary and assume that greedy parsing will be applied.. Further discussion is found in [25] and [46], where references dealing with this problem are also cited.

A new implementation of the LZ incremental parsing algorithm is given in [39].

6. The use of multi-bit flags to allow a greater variety of entities in the compressed data stream is described in [69]. This allows interesting encodings for phrases and gives a compressor which is, on average, very similar to LZB (the best of the LZ77 compressors) and better for some files.

7. A problem identified and addressed in [55] is that even if the input to the model is a steady stream of characters ---"a, b, c, d, e" and so on "---the output will soon become bursty. In an example given there, after the first 256 strings, the output of the model is "ab, cd, ef" and so on. After the 768th character the output is "abc, def, efg" and so on. Between the chatacter groupings time is spent by the software in searching the table, assigning string numbers, and filling (or reading) transmission containers. The bursty nature of the Lempel-Ziv algorithm results in the "jerky screen" effect that mirrors the start-stop pattern of the transmission stream. A practical discussion of how this problem was solved is given in [55].

8. Another limitation of the LZ algorithm identified in [55] is the so-called boundary problem. That problem occurs when portions of the same string occur over and over again in the table, even though, logically, it would maske sense to have only one copy of a

string occupy space. Take, for example, the string "there". Embedded within this string are also the strings "here" and "ere," which may very well have their own string number assignments taking up space in the table. The Telcor design described in [55] incorporates a solution to this problem.

9. One problem with code for LZ is that it may not adapt well to compressing files of differing sizes [9]. Usually 14- or 15-bit codes give better compression ratios on large files (because they have a larger string table to work with), but poorer performance on small files. One way to get around this problem is to use variable length codes[9].

10. As discussed in [14] the ZLWAx family of data compressors build their dictionaries using the LZW algorithm and represent dictionary entries using arithmetic coding. Members of the ZLWAx family employ a predictive strategy that takes advantage of the arithmetic coder's ability to assign different probabilities to different dictionary entries, and two enhancements to the basic LZW algorithm.

11. A predictive strategy can be used to exploit the idea of context in the source text. This is done in [14] to obtain ZLWAP coders.

12. String extension makes the dictionary grow faster and contain longer strings, so many sources can be represented by fewer dictionary entries. This idea is used in the ZLWAS coder [14] to allow it to adapt quickly to highly repetitive text when long dictionary entries have a good chance of being used. If the source text makes little use of long dictionary entries, however, string extension can reduce rather than enhance compression. It can increase the number of useless dictionary entries to a point that offsets the gains realized by representing the source text with fewer entries.

13. One problem with LZW coders in general is that they can create many useless dictionary entries. Conditional dictionary entry is a strategy that often reduces the number of useless character extensions for a LZW coder with the string extension enhancement[14]. The idea behind conditional dictionary entry is that when the entries in a dictionary are letters or parts of words, it makes sense to character extend them until they become words. However, once you have a dictionary of words, it makes little sense to build long phrases character by character. It makes more sense to build them up by putting words together. The conditional dictionary entry strategy does not add new dictionary entries that end with a blank space when doing character extension. In simple

terms, once the dictionary contains a complete word, that word cannot be lengthened by character extension. The virtue of this strategy is that it uses a nearly universal word separator, the blank space, as a word boundary. This symbol is uncommon enough in binary files that it does not degrade their compression,and is common enough in program source and text files to enhance compression. This word based strategy, added to prediction, string extension and arithmetic coding produced the ZLWASPC coder, whose performance was best among twelve coding schemes tested on six very different files, for each of these files in [14]. ZLWASP and ZLWAS coding schemes were close second and third bests. Each of these (and the ZLWA strategy, the fourth best) squeezed significant additional compression out of ZLW coding. Future directions for ZLW-based coders are many, but it seems that that improving the speed of the arithmetic coder, enhancing the conditional dictionary entry strategy or adding a first order Markov model to the predictive strategy are the most promising directions. The current conditional dictionary entry strategy has the virtues of simplicity and applicability to many different dinds of text files, but it could be enhanced to recognize attributes of text files and dynamically adjust conditional entry to improve compression. The existing predictive strategy seems incompatible with the first order Markov conditioning scheme from Miller and Wegman's A2 algorithm, but it would be interesting to modify the predictive strategy to benefit from Markov modeling. Another possibility would be to incorporate Storer and Szymanski's linear time OMP/UD scheme [70] into the dictionary heuristics of [14]. Finally, it may be useful to add a dictionary pruning scheme for future experiments.

14. Take the occurence probabilities of the source messages into consideration in some way and assign variable-length codewords[21]. In addition, to obtain high-speed processing, the MTF method is adopted for the administration of the dictionary (sorting the queueing buffer for each encoding/decoding). The coding rule is that the shallower the entry of a parsed string in the queueing buffer the shorter is the length of the codeword assigned to it. The proposed method has LRU deletion heuristic for the queueing buffer, each of whose entries has a different parsed string.

15. Implementation of word-based dynamic Lempel-Ziv (WDLZW) is discussed in [33]. The novel feature is that the algorithm is optimized for the compression of natural language data, in which all the spaces between words are deleted whenever copy codes or literal codes are sent out. Therefore, better compression rates can be achieved. The algorithm can still compress alternative forms of data. WDLZW improves LZW coding in three ways. (1) Initialising the whole string table with different combinations of symbols. This is a straightforward modification which enables compression to be achieved even

when relatively little data has been processed. (2) Instead of having one string table, it utilizes a hierarchy of string tables. Each successive string table is larger than the preceding one. The underlying idea is that the most frequently used strings should reside in the shorter tables. The shorter tables require fewer bits to identify the corresponding string. (3) Setting up the move-to-front and weighting system. Every time a string is matched an associated frequency counter is updated. In all idealized implementations the tables would be ranked according to frequency counts. In practice, the process of adjusting tables entails too large an overhead for acceptable performance. Instead, a MTF weighting is used. When compressing, the algorithm sends out the related bits to address the longest matching string recognized in the table. After
that the algorithm moves the string into the first position of its block (a block is defined as a group of entries with the same weights), increases its weights byh one, and then moves its position to the front of its new block again. This method aims to ensure that frequently used strings are kept in the larger tables. The overall effect is to minimize the average number of bits required to code a string when compared with a single table implementation. For text-based information WDLZW offers attractive pereformance. For other forms of data, WDLZW provides compression rates similar to those of dynamic Lempel-Ziv systems.

16. Two ways of improving LZC (the UNIX compress command) are selected, implemented and tested in [56]. One of these was a method of loading the dictionary at a faster rate, and had not been used before. The was a method to phase in increased lengths of binary numbers gradually. Together, these two compression methods achieve substantial improvements, especially on shorter files where the dictionary does not normally have a chance to fill to an extent that achieves good compression performance.

17. To obviate the disadvantages of the overhead of traditional LRU implementations, the new dictionary management heuristic "TAG" is introduced in [45]. This provides the compression advantage of LRU schemes, while requiring only a fraction of the additional memory and computational resources of LRU implementations. This paper addresses the problem of realizing LZ78 in hardware and the problem of maintaining a continuously adaptive dictionary for LZ78 in real time. The new dictionary manager resolves the time-space tradeoffs of its predecessors. Its superior compression performance, memory requirements and speed are demonstrated.

References

51. K. Ise and H. Tanaka, "A Note on Performance Improvement of Ziv-Lempel Code", Electronics and Communications in Japan, Part 3, vol. 76, pp.85-93, Aug. 1993.

52. J. L. JBentley, D. D. Sleator, R. E. Tarjan and V. K. Wei, "A Locally Adaptive Data Compression Scheme", Communications of the ACM, vol. 29, pp. 320-330, April 1986.

53. C. D. Thomborson and B. W.-Y. Wei, "Systolic Implementations of a Move-to-Front Text Compressor", In Proc. of the 1989 ACM Symposium on Parallel Algorithms and Architectures, June 1989.

54. V. S. Miller and M. N. Wegman, "Variations on a Theme by Ziv and Lempel", Technical Report RC 10630 (47798), IBM Thomas J. Watson Research Center, July 1984.

55. F. Bacon, "How to quadruple dial-up communications efficiency", Mini-Micro Systems, Feb. 1988, pp.77-81.

56. R. N. Horspool, "Improving LZW", in "DCC '91: Data Compression Conference", edited by J. A. Storer and J. H. Reif. Washington: IEEE Computer Society Press, 1991, pp. 332-341.

57. C Thomborson, "V.42bis and Other Ziv-Lempel Variants" (Abstract), in "DCC '91: Data Compression Conference", edited by J. A. Storer and J. H. Reif. Washington: IEEE Computer Society Press, 1991, p. 460.

58. E. R. Fiala and D. H. Greene, "Data Compression with Finite Windows", Communications of the Association for Computing Machinery, vol. 32, pp.490-505, April, 1989.

59. R. N. Williams, "An Extremely Fast Ziv-Lempel Data Compression Algorithm", in DCC '91: Data Compression Conference, edited by J. A. Storer and J. H. Reif. Washington: IEEE Computer Society Press, 1991, pp.362-371.

60. K.Winters,R.Bode and E.Schneider "A CAM-Basec LZ Data Compression IC", Proc. of the 5th NASA Symposium on VLSI Design 1993, pp. 12.2.1-12.2.8.

61. J. Venbrux, G. Zweigle, J. Gambles, D. Wiseman, W. H. Miller and P.- S. Yeh, "An Adaptive Lossless Data Compression Algorithm and VLSI Implementations", Proc. of the 5th NASA Symposium on VLSI Design 1993, pp. 1.2.1-1.2.16.

62. H. U. Khan, J. Ahmad, A. Mahmood and H. A. Fatmi, "Text Compression as Rule Based Pattern Recognition", Electronics Letters, vol. 29, pp. 1752-1753, 30th September 1993.

63. R. N. Horspool and G. V. Cormack, "Constructing Word-Based Text Compression Algorithms", in DCC '92: Data Compression Conference, edited by J. A. Storer and M. Cohn.Washington: IEEE Computer Society Press, 1992, pp. 62 - 71.

64. P. G. Howard and J. S. Vitter, "Design and Analysis of Fast Text Compression Based on Quasi-Arithmetic Coding", in DCC '93: Data Compression Conference, edited by J.A.Storer and M.Cohn. Washington: IEEE Computer Society Press, 1993, pp. 98-107.

65. R. F. Rice, "Some Practical Universal Noiseless Coding Techniques", Jet Propulsion Laboratory, JPL Publication 79-22, Pasadena, California, March1979.

66. N. Morita and K. Kobayashi, "Data compression of computer files based on restricted reproducible parsing algorithm", Trans. Inf. Proc. Soc. Japan, vol.33, pp. 110-121, 1992.

67. V. S. Miller and M. N. Wegman, "Variations on a theme by Ziv and Lempel", in Combinatorial Algorithms on Words, edited by A. Apostolico and Z. Galil, NATO ASI Series Vol. F12. New York: Springer Verlag, 1992.

68. A. Mayne and E. B. James, "Information Compression by Factorizing Common Strings", The Computer Journal, vol.18, pp.157-160, 1975.

69. P. Fenwick, "Ziv-Lempel Encoding with Multi-Bit Flags", in DCC '93: Data Compression Conference, edited by J. A. Storer and M. Cohn. Washington: IEEE Computer Society Press, 1993, 138-147.

70. J. A. Storer and T. G. Szymanski, "Data Compression via Textual Substitution", Journal of the ACM, vol. 29, pp. 928-951, Oct. 1982.

71. Y. Perl, V. Maram and N. Kadakuntla, "The Cascading of the LZW Compression Algorithm with Arithnetic Coding", in DCC '91: Data Compression Conference, edited by J. A. Storer and J. H. Reif. Washington: IEEE Computer Society Press, 1991, pp. 277-286.