SOME MEASURED PERFORMANCE BOUNDS AND IMPLEMENTATION CONSIDERATIONS FOR THE LEMPEL-ZIV-WELCH DATA COMPACTION ALGORITHM

H. D. Jacobsen

ABSTRACT

Lempel-Ziv-Welch (LZW) algorithm is a popular data compaction technique that has been adopted by CCITT in its V.42bis recommendation and is often implemented in association with the V.32 standard for 9600 bps modems. It has also been implemented as Microcom Networking Protocol (MNP) Level 7, where it goes by the name of Enhanced Data Compression. LZW compacts data by encoding frequently occurring input strings with a single output symbol. The algorithm automatically generates a string dictionary for each symbol at each end of the transmission path. The amount of compaction that can be derived with the LZW algorithm varies with the type of data being transmitted and the efficiency by which table entries can be indexed. Table indexing is usually implemented by use of a hashing table. Although some manufacturers advertise a 4-to-1 gain in throughput, this seems to be an extreme case. This paper documents a implementation of the exact ZLW algorithm. The results presented in this paper are significantly less, typically on the order of 1-to-2 for ASCII text, with substantially less compaction for pre-compacted files or files containing random bit patterns.

The efficiency of the LZW algorith on ASCII text is shown to be a function of dictionary size and block size. Although fewer transmitted symbols are required for larger dictionary tables, the additional bits required for the symbol index is marginally greater than the efficiency that is gained. The net effect is that dictionary sizes beyond 2K in size are increasingly less efficient for input data block sizes of 10K or more. The author concludes that the algorithm could be implemented as a direct table look-up rather than through a hashing algorithm. This would allow the LZW to be implemented with very simple firmware and with a maximum of hardware efficiency.

ARTICLETITLE: "Some Measured Performance Bounds and Implementation Considerations for the Lempel-Ziv-Welch Data Compaction Algorithm"

KEYWORDS: (1) Data Compression (Principal Topic)
(2) Lempel-Ziv Algorithm (Primary Category)

THE LEMPEL-ZIV-WELCH (LZW) ALGORITHM

The Lempel-Ziv algorithm for data compaction* was first published in 1977 (1). An efficient hardware implementation of the algorithm was published in 1984 by Terry A. Welch of the Sperry Research Center, Sudbury, Massachusetts (2). The Welch implementation is referred to as the Lempel-Ziv-Welch (LZW) algorithm, and is currently used in V.42bis and MNP 7 data compaction protocols

The LZW algorithm is described in terms of the following:

Input File:	Data to be transferred. Data is usually in the form of binary information divided into 8-bit bytes.
Source Alphabet:	All legitimate symbols that can be encountered in the input data. Bit transparency transmission is assumed, so all 8-bit patterns are considered legitimate data. The source alphabet is, therefore, composed of 2^8 or 256 distinct symbols.
Dictionary Table:	Also referred to as the string table or translation table. The dictionary maps variable-length strings of input characters into a fixed-length output symbols, where the code corresponds to the dictionary index for the encoded string. An identical dictionary is generated on each end of the transmission channel. The dictionary is initialized for each of the possible single-character input characters.
Output Symbols:	Address of the table entry corresponding to the encoded string. The number of bits used for each symbol in the output alphabet is determined by the maximum number of output symbols in the dictionary. For a 12-bit output word, a maximum of 2^12 or 4096 output symbols (input strings) could be identified.
LZW Algorithm:	If a string w has been encountered previously and is in the table with the index n, and a new string wK is encountered where w is extended by the character K, a subsequent table entry, say index m, is created representing the string wK. The index for w is transmitted and the next index transmitted identifies K as the

---

*Data compaction, rather than data compression, is used in this paper to represent the process by which the number of transmitted symbols is reduced with no loss of information. Data compression, in the information theoretic sense, allows for the potential loss of information.

extension character for the string w, which allows the dictionary to be extended to include the string wK (with index m) at the receiving end. Figure 1 illustrates input data, dictionary entry, and the output data for a simple example of this algorithm. Figure 1 indicates that the index m for each new string wK is given a successive index. However, this cannot be done without some form of associative memory or table look-up. Lacking these, the index is typically generated with the use of a hashing table, which leaves some addresses unuseable.

The Lempel-Ziv-Welch Algorithm

Input Data: ABADCABCA

Dictionary	String	Index	Contents
(ASCII)	NUL	0	0
	SOH	1	1
	...	...	...
	A	65	65
	B	66	66
	C	67	67
	D	68	68
	...	...	...
	AB	256	65B
	BA	257	66A
	AD	258	65D
	DC	259	68C
	CA	259	67A
	ABC	260	256C

Output Data:	65 .	66 .	65 .	68 .	67 .	256 .	259
Decoded Data:	A	B	A	D	C	AB	CA

Figure 1

BOUNDING THE DICTIONARY AND BLOCK SIZES

The LZW algorithm depends on recurring strings of characters for data compaction. It should be expected, therefore, that its efficiency will be a function of the number of recurring strings identified within the input data, the length of those strings, and how frequently they are encountered.

The number of strings that are encoded for a given block size is determined by the constraints placed on the dictionary and the size of the input data block. Thc string dictionary can be expanded only at the expense of a longer output symbol. Longer blocks of data will have a larger number of encoded strings but will have asymptotic behavior due to the limited number of strings defined in the dictionary.

The length and frequency of strings identified in the dictionary are also a function of the input data itself. As shown in Figure 2, random bit patterns (e.g., image files having high image content) and pre-compacted files experience very little compaction with the LZW algorithm. This can be expected, as the first has little in the way of repetitive pattern and the second has had recurring patterns removed though other compaction techniques. The compaction of these data types with the LZW algorithm is not significantly influenced by dictionary or block sizes, and therefore are not considered in our analysis. We will restrict our analysis to an input file containing uniform ASCII text. An algorithm that optimizes the compaction of ASCII text is expected to be globally optimal.

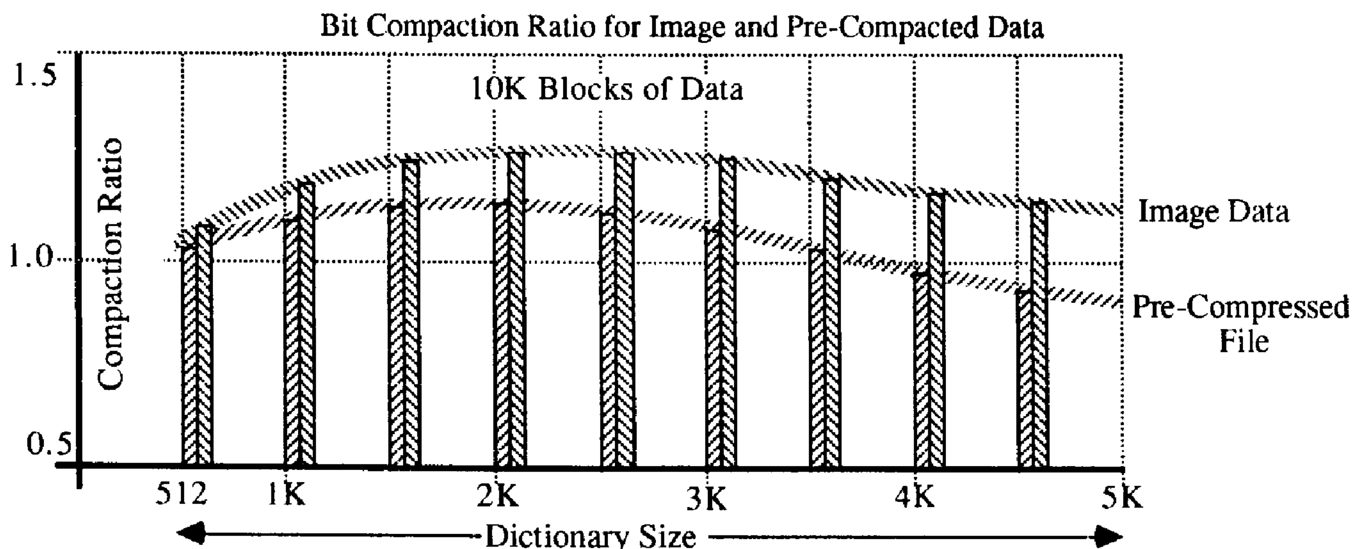

Figure 2

As noted in reference [2], the principal concern in implementation of the LZW algorithm is the storing and accessing of dictionary entries. In an ideal situation, the output symbol of length k bits would provide 2^k dictionary entries. Most algorithms use a form of hashing for developing the dictionary index, with some entries unaccessible by the hashing technique. We will assume, for the sake of bounding the LZW algorithm, that a form of indexing is implemented which allow us to utilize all possible output symbols as string references.

OBSERVATIONS AND CONCLUSIONS

LZW algorithm provides two measures of compaction efficiency: the ratio of input symbols to output symbols (symbol compaction), and the ratio of input bits to output bits (bit compaction). The latter is the statistic of interest. Compaction ratios were determined for a range of cases: for dictionary size from 512 (9-bit addressing) to 16K (14-bit addressing); and for data block sizes from 5 kilobytes to 50 kilobytes. Although the symbol compaction increases with dictionary size, the rate of increase is marginally less than the rate by which addressing overheads are increased. The resulting bit compaction ratio is shown in Figure 3.

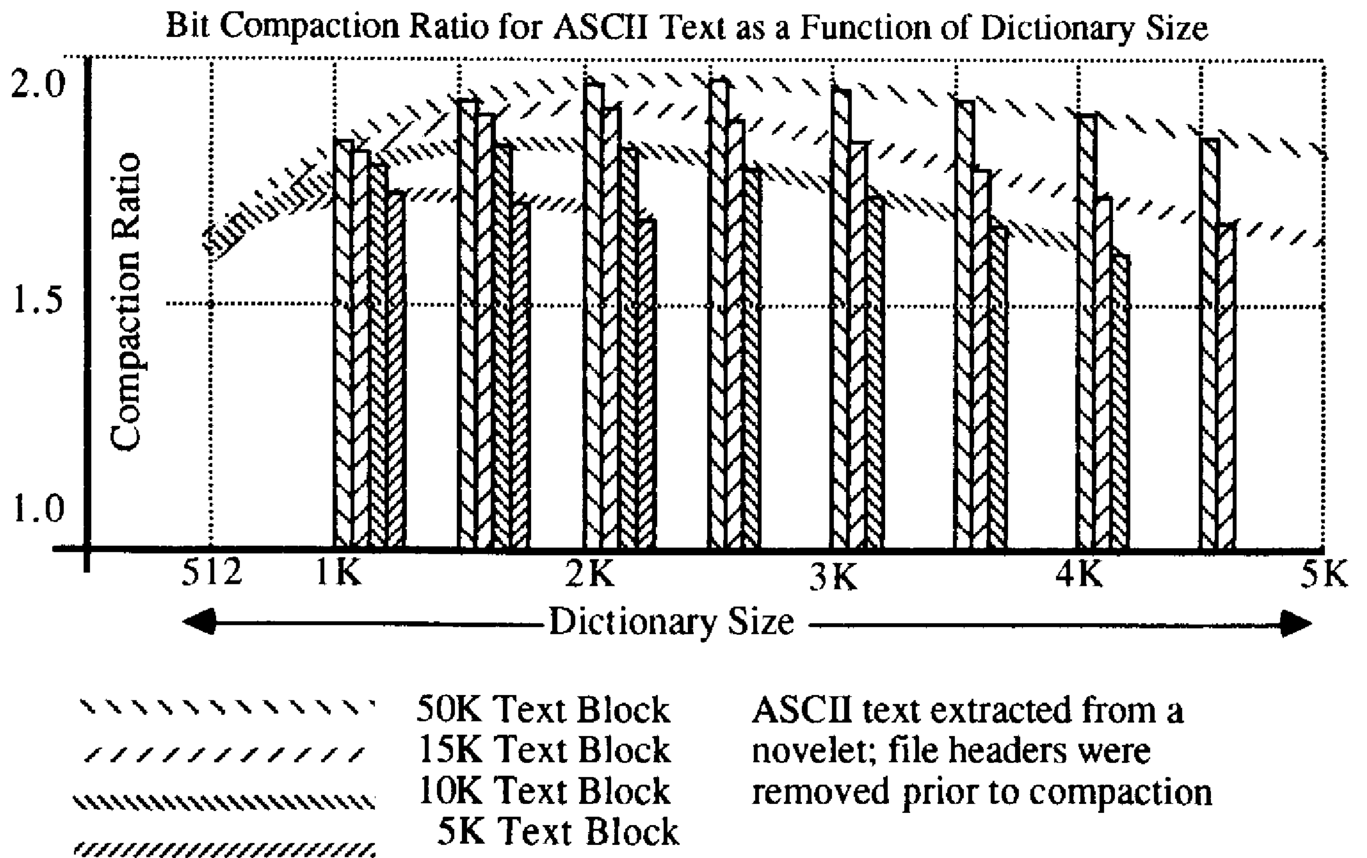

Figure 3

A careful review of Figure 3 indicates that the bit compaction rate for the LZW algorithm on ASCII text is optimized on or near a dictionary size of 2 kilobytes, i.e., a dictionary utilizing 11-bit addressing. This provides a compaction ratio of from 1.7 (5K data blocks) to 1.95 (50K data blocks). This being the case, it is reasonable to implement the LZW algorithm using direct table indexing rather than a hashing function as is commonly used. A table look-up technique would require an indexing array of size (2K x 256), with each array entry containing an 11-bit index. The indexing array would indicate, for each of the dictionary entries (input strings), the index for all extensions of that string that have been

encountered previously. 11-bit indexing is somewhat awkward in digital design where hardware components are configured in 4- and 8-bit increments. The use of 12-bit indexing could be used with approximately 10% degradation in performance.

These results have significant import to a practical implementation of the LZW algorithm. A (2K x 256 x 11 bit) addressing array would require considerably less than one megabyte of dynamic random access memory (DRAM). One megabyte DRAM is currently selling for approximately $30.00 at OEM prices, which would make this implementation cost effective. The resulting algorithm for implementing the LZW algorithm is very simple, as shown in the logical flow of Figure 4 for the transmitting end. (A QuickBASIC program listing is given in Appendix A.) An equally simple program code is required at the receiving end for recovering the compacted code. These implementations would also run considerably faster than the hashing techniques currently used with LZW implementations.

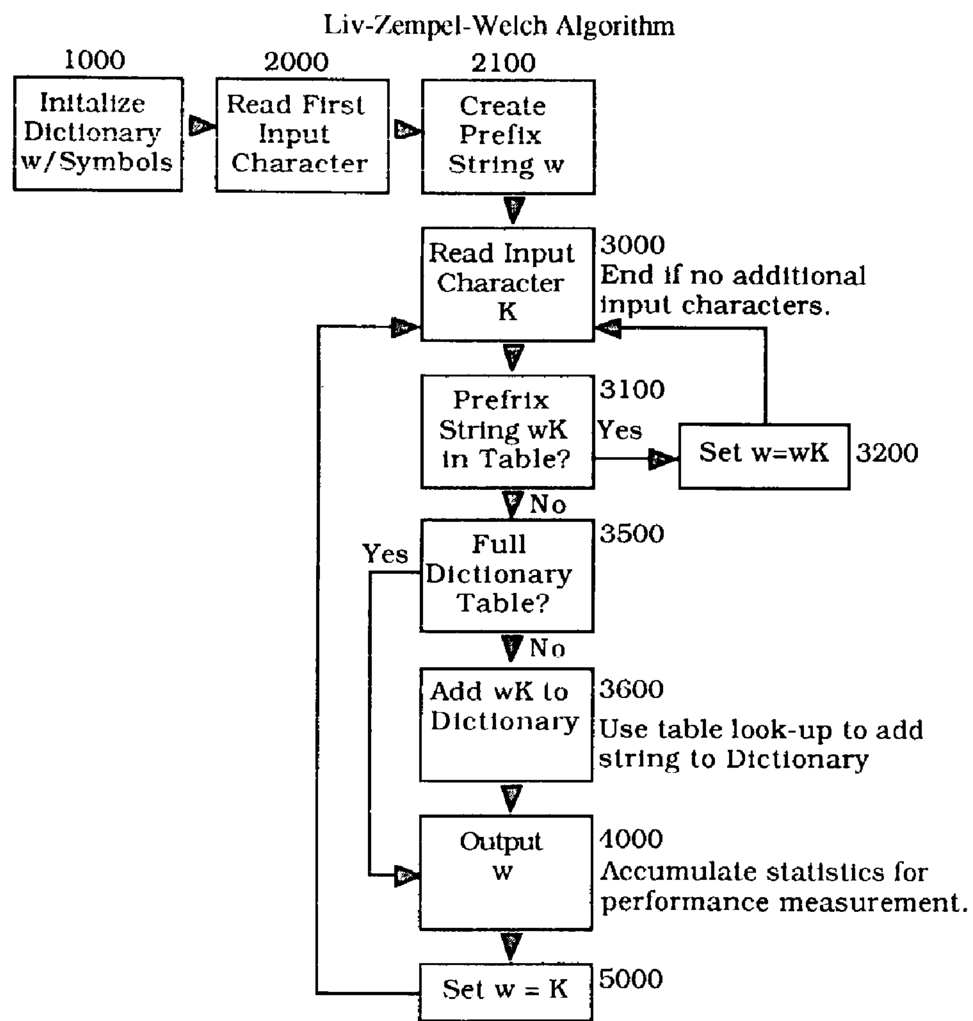

Figure 4

REFERENCES

[1] Ziv, J. and Lempel, A., “A Universal Algorithm for Sequential Data Compression,” IEEETrans.Information Theory, Vol. 1T-23, No. 3, May 1977, pp. 337-343.

[2] Welch.T. A., “A Technique for High Performance Data Compression,” Computer, June 1984, pp. 8-19.

APPFNDIX A:
LISTING OF A SIMPLE TABLE LOOK-UP IMPLEMENTATION FOR THE LZW ALGORITHM (BASIC)

```
1000  REM Limpel-Ziv-Welch Source Coder
1003  REM
1100  REM Define dictionary and initialize to symbol set
      DIM STATIC Dict%(2048,256)
1110  REM Identify input file
      file 1$=FILES$(1)
      OPEN file1$ FOR INPUT AS #1 LEN=1024
1200  REM Set parameters D(ict) and B(lock)
      D=2048
      B=1024*10
2000  REM Begin real data input for compaction
      w$=INPUT$(1,1)
      windex = ASC(w$)
      symbolin=l : symbolout=0
3000  REM Read a character from the input file
      IF EOF(1) GOTO 6000
      k$=INPUT$(1,1)
      symbolin=symbolin+l
      IF symbolin>B GOTO 6000
3100  REM Check to see if string encountered previously
      IF Dict%(windex,ASC(k$))=0 GOTO 3500
      windex=Dict%(windex,ASC(k$))
3500  REM Check to see if table is full
      IF Tindex>=D GOTO 4000
3200  REM Build string further
      w$=w$+k$
      GOTO 3000
3600  REM Add new string to dictionary
      Tindex=Tindex+l : Dict%(windex,ASC(k$))=Tindex
```

```
4000  REM Output a symbol
      REM <Output routine would go here>
      symbolout=symbolout+l
5000  REM Reset output string to first character of new string
      windex=ASC(k$) : w$=k$
      GOTO 3000
6000  REM End of data block
```