Journal of Computer Science 4 (3): 205-210, 2008
ISSN 1549-3636
© 2008 Science Publications

Analysis of String Matching Compression Algorithms

Krishnaveny Nadarajan and Zuriati Ahmad Zukarnain
Faculty of Computer Science and Information Technology, Universiti Putra Malaysia,
Selangor, Malaysia 603 89466565

Abstract: The improvement rate in microprocessor speed by far exceeds the improvement in DRAM memory. This increasing processor-memory performance gap is the primary obstacle to improved computer system performance. As a result, the size of main memory has increased gradually. To utilize main memory's resources effectively, data compression can be applied. Through compression, the data can be compressed by eliminating redundant elements. Thus, this study incorporates compression to main memory in order to fully utilize its resources and improve performance of data access. This project evaluates the performance of the compression algorithms in order to increase the performance of memory access. The compression algorithms are string matching based which are LZW and LZSS. Through simulation, the effectiveness and compressibility of the algorithms were compared. The performances of the algorithms were evaluated in term of compression time, decompression time and compressed size. The simulation result shows that LZSS is an efficient compression algorithm compared to LZW.

Key words: LZSS, LZW, string matching, lossless compression

INTRODUCTION

Data compression is a technique of encoding information using fewer bits than an unencoded representation would use through specific encoding or compression algorithm. All forms of data, which includes text, numerical and image, contain redundant elements. Through compression, the data can be compressed by eliminating the redundant elements[21].

Data compression techniques use a model to function. The input stream, generated from a data source, is fed into a compressor. The compressor then codes and compresses data. To regenerate original data from the compressed data, decoder is used. The decoder applies the reverse algorithm of that used by the compressor. Moreover, the decoder has some prior knowledge as to how the data is being compressed[4].

Data compression is divided into two major categories, which is lossless compression technique and lossy compression technique. In lossless compression, no information is lost and the decompressed data are identical to the original uncompressed data. While, in lossy compression, the decompressed data may be an acceptable approximation to the original uncompressed data[9].

Lossless compression technique is grouped into two, statistical analysis based compression and string matching compression algorithm. This project focuses on analyzing string matching compression algorithm.

The increase of processor clock speed caused the gap between processor, main memory and disk space to widen. As a result, the size of cache and main memory increased. This performance gap affects the reliability and the performance of the memory resource access overall. Thus, this study incorporates compression to main memory in order to fully utilize its resources.

RELATED WORKS

All forms of data contain redundant elements. Through compression, these redundant elements can be eliminated. First algorithm for compressing data was introduced by Claude Shannon[3]. At present there are lots of techniques available for compressing data, each tailored to match the needs of a particular application.

Compression techniques can be classified into two categories, lossless compression techniques and lossy compression techniques. The classification is based on the relationship between inputs and outputs after a compression or expansion cycle is complete. In lossless compression, the output exactly matches with the input after a compression or expansion cycle. Lossless techniques are mainly applicable to data files where a single bit loss can render the file useless. In contrast, a

Corresponding Author: Zuriati Ahmad Zukarnain, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, Selangor, Malaysia 603 89466565

lossy compression technique does not yield an exact copy of input after a compression.

Navarro *et al.,* 1999 presented compressed pattern matching algorithms for the Lempel-Ziv-Welch (LZW) compression which run faster than a decompression followed by a search[1]. However, the algorithms are slow in comparison with pattern matching in uncompressed text if compared the CPU time. This mean the LZW compression did not speed up the pattern matching.

Matias *et al*., 1999, resolved the issue of online optimal parsing by showing that for all dictionary construction schemes with the prefix property greedy parsing with a single step look ahead is optimal on all input strings this scheme is called flexible parsing or (FP)[2]. Kniesser *et al*., 2003, proposed a method for compressing the scan test patterns using LZW that does not require the scan chain to have a particular architecture or layout[3]. This method leverages the large number of Don't-Cares in test vectors in order to improve the compression ratio significantly. Efficient hardware decompression architecture is also presented using existing in-chip embedded memories.

STRING MATCHING COMPRESSION ALGORITHM

The string matching compression algorithms that were analyzed are LZSS (Ziv-Lempel-Storer-Szymanski) and LZW (Ziv-Lempel-Welch) algorithm.

LZSS algorithm: The LZSS compression algorithm makes use of two buffers, which are dictionary buffer and look ahead buffer. The dictionary buffer contains the last *N* symbols of source that have been processed, while look ahead buffer contains the next symbols to be processed. The algorithm attempts to match two or more symbols from the beginning of the look ahead buffer to a string in the dictionary buffer, if no match is found, the first symbol in the look ahead buffer is output as a 9 bit symbol and is also shifted into the dictionary[12].

If a match is found, the algorithm continues to scan for the longest match. It is intended that the dictionary reference should be shorter than the string it replaces. The LZSS algorithm compress series of strings by converting the strings into a dictionary offset and string length. For example, if the string mnop appeared in the dictionary at position 1234, it may be encoded as {offset = 1234, length = 4}.

The LZSS dictionary is not an external dictionary that lists all known symbol strings. The larger *N*, the longer it takes to search the whole dictionary for a match and the more bits will be required to store the offset into the dictionary. Typically dictionaries contain an amount of symbols that can be represented by a whole power of 2. A 432 symbol dictionary would require 9 bits to represent all possible offsets. If need to use 9 bits, the 512 symbol dictionary might needed to have more entries.

Since dictionaries are sliding windows, once the (N + 1)th symbol is processed and added to the dictionary, the first symbol is removed. Additional new symbols cause an equal number of the oldest symbols to slide out. In the example above, after encoding mnop as {offset = 1234, length = 4}, the sliding window would shift over 4 characters and the first 4 symbols (offsets 0 ... 3) would slide off the front of the sliding window. m, n, o and p would then be entered the dictionary into positions (N - 4), (N - 3), (N - 2) and (N - 1)[21].

LZW algorithm: The LZW Algorithm maintains a dictionary of strings with their codes for both compression and decompression process. When any of the strings in the dictionary appears in the input to the compressor, the code for that string is substituted, the decompressor, when it reads such a code, replaces it with the corresponding string from dictionary. As compression occurs, new strings are added to the dictionary. The dictionary is represented as a set of trees, with each tree having a root corresponding to a character in the alphabet. In default case, there are 256 trees with all possible 8 bit characters[21].

At any time, the dictionary contains all one character strings plus some multiple character strings. By the mechanism by which strings are added to the dictionary for any multiple character string in the dictionary, all of its leading substrings are also in the dictionary. For example, if the string PRADA is in the dictionary, with a unique code word, then the strings PRA and DA are also in the dictionary, each with its own unique code word.

The algorithm will always match the input to the longest matching string in the dictionary. The transmitter partitions the input into strings that are in the dictionary and converts each string into its corresponding code word. Since all one character strings are always in the dictionary, all of the input can be partitioned into strings in the dictionary. The receiver accepts a stream of code words and converts each code word to its corresponding character strings[4].

MATERIALS AND METHODS

An important criterion in performance evaluation of the compression algorithm is the selection of

Table 4: Details of Files in Calgary Corpus

File name	Description	Size (bytes)
bib	Bibliography	111261
book1	Fiction book	768771
book2	Non-fiction book	610856
geo	Geophysical data	102400
news	USENET batch file	377109
obj1	Object code for VAX	21504
obj2	Object code for Apple Mac	246814
paper1	Technical paper	53161
paper2	Technical paper	82199
pic	Black and white fax picture	513216
progc	Source code in "C"	39611
progl	Source code in LISP	71646
progp	Source code in PASCAL	49379
trans	Transcript of terminal session	93695

evaluation technique. There are three techniques available, which are analytical modeling, simulation and measurement. For this project, simulation technique is selected. This is due to the fact that simulation technique allows incorporations of more details while undertaking fewer assumptions.

Several performance metric have been selected to evaluate the performance of LZSS and LZW compression algorithm. Following are the selected performance metrics:

- Compression time : Amount of time the algorithm takes to compress a file or data
- Decompression time : Amount of time the algorithm takes to decompress a file or data
- Compressed size (%) : Amount of compression achieved

$$\frac{\text{(Size of the input data - Size of the compressed data)}}{\text{Size of input data}}$$

To test the performance of the algorithm, standard test file, Calgary Corpus is used. The Calgary Corpus is the most referenced corpus in the data compression field and is the de facto standard for lossless compression evaluation. Calgary Corpus consists of text files, images and other achieve files. Details of files in the Calgary Corpus are in Table 1.

RESULT AND DISCUSSION

This section analyzes and discuses the results obtained for the performance metrics from the simulation using the standard test files of Calgary Corpus. The efficiency and compressibility performance of LZSS and LZW algorithm is evaluated in term of compression time, compressed size (%) and decompression time. The performances of the algorithms were analyzed under different size of dictionary. For this project purpose, the evaluated dictionary size is 2K, 4K and 6K.

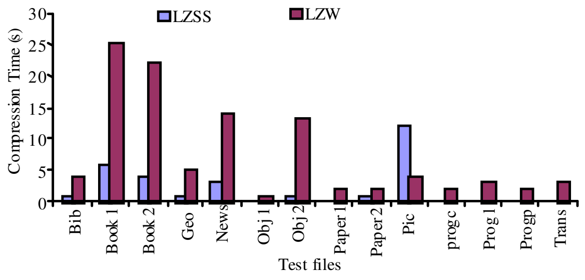

Fig. 1: Compression time for dictionary size 2K

Figure 1 shows the result of compression time of the test files when the dictionary size is set to 2K. LZSS has less compression time compared to LZW, except for the case of pic file.

When compressing the pic file, LZW gives better results. The reason for this exception is the uncommon content of this file, which contains a lot of nulls. When the length component of LZSS is just 5 bits, LZSS can put a pointer to no more than 32 bits. LZW, however, can have a pointer to an infinite string and will look for the longest match in the dictionary. When using LZW, each entry has an old string and a new letter, hence it can save much longer strings.

LZW with a pointer of 9 bits build a file that contains the 256 possible characters of ASCII. This will put the pairs (0, 1) (1, 2)…(254, 255) into the dictionary of LZW, according to the algorithm of LZW. If a 0 is added after the last 255, the pair (255,0) will be added and these numbers will use up the entire dictionary. The dictionary has 512 entries. The first 256 entries are of the single characters, while the other 256 entries are of the pairs.

LZW can handle these pairs better because LZSS saves one bit for character or pointer flag, several bits for the pointer and several bits for length component. Usually LZSS does not replace a pair of characters by a pointer because of the high price, but even if LZSS replaces the pair, the gain will be small, while LZW can save more bits when pointing just to a pair of characters. Thus, LZW algorithm compress image file faster than LZSS algorithm[22].

Figure 2 and 3 show the result when the dictionary size in increased to 4K and 6K. LZSS still compressed faster than LZW except for the *pic* file.

Figure 4 shows the compression time under three different size of dictionary for a file from the Calgary

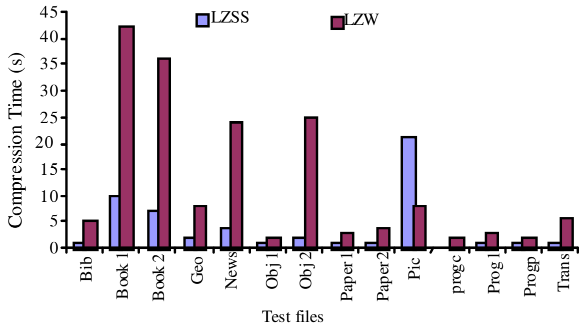

Fig. 2: Compression time for dictionary size 4K

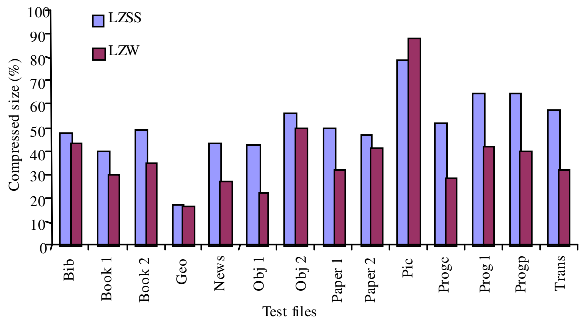

Fig. 5: Compressed size for dictionary size 2K

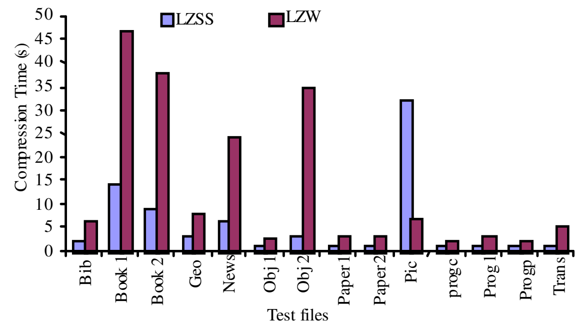

Fig. 3: Compression time for dictionary size 6K

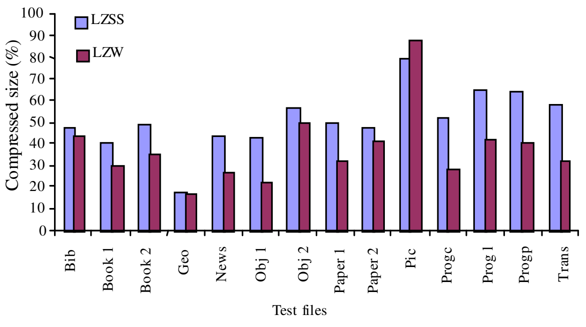

Fig. 6: Compressed size for dictionary size 4K

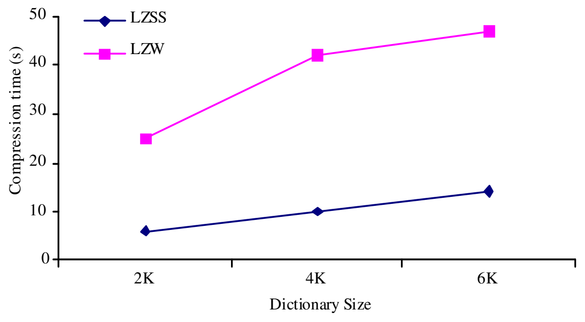

Fig. 4: Compression time for file Book 1

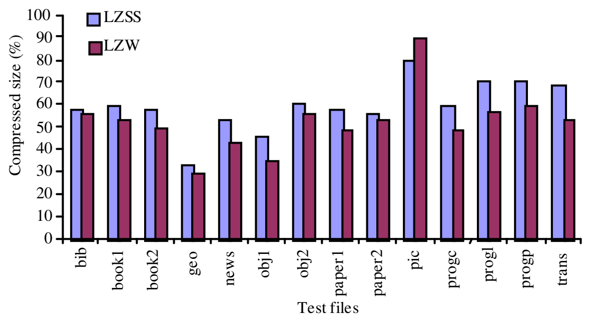

Fig. 7: Compressed size for dictionary size 6K

Corpus, which is Book 1. For compression time, it can be seen that as the dictionary size increases from 2K to 6K, the amount of time both algorithm takes to compress a file also increases.

The compression time increases as the dictionary size increases. During compression, searching for a match in a large dictionary requires large amount of time, thus increases the compression time. The graph also shows that LZSS compression algorithm require less amount of time to compress a file than LZW algorithm.

Figure 5 shows the result of compressed size achieved with the dictionary size of 2K. LZSS algorithm produce high percentage of compressibility compared to LZW, except in the case of pic file.

Figure 6 and 7 show the result when the dictionary size in increased to 4K and 6K. LZSS algorithm has high percentage of compressed size compared to LZW algorithm except for the pic file.

Figure 8 shows the compressed size under three different size of dictionary for a file from the Calgary Corpus, which is Book 1. For compressed size, as the dictionary size is increased from 2K-6K, the percentage of compressibility achieved also increases. If the dictionary size is large, it can store more characters or strings, thus, during compression, the compressor will

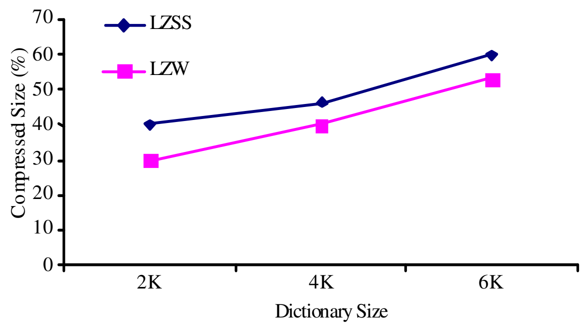

Fig. 8: Compressed size for file book1

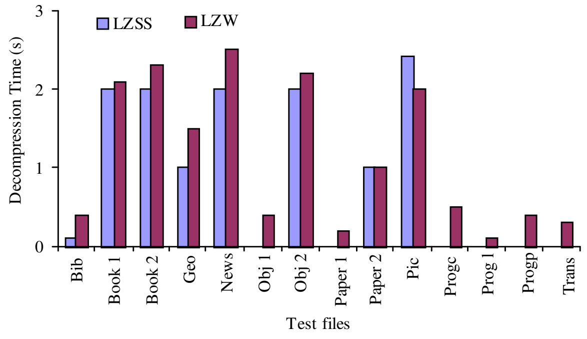

Fig. 9: Decompression time

only output the index or reference to the character in the dictionary. This contributes to high percentage of compressibility.

The compressibility achieved through LZSS is higher than LZW algorithm. It is because, in LZSS, a file can be built containing the single characters or (offset, length) pairs as described previously. Such a file will be compressed by LZSS, always in a better manner than by LZW. The compressibility is dependent on the pointer size of LZSS.

LZW always puts a pointer, while LZSS uses pointers only in the appropriate cases. If LZSS creates fewer pointers, it will indicate that LZSS has chosen not to put a pointer because it is less adequate. In contrast, LZW puts a pointer because this is its usual behavior and that pointer is more adequate[22].

Figure 9 shows the result of decompression time for both LZSS and LZW algorithm. The graph show that LZSS algorithm requires slightly lower amount of time to decompress a file compared to LZW algorithm except for pic file, where LZW decompress faster. The amount of time needed to decompress a file is same to all size of dictionary.

This is due to the fact that decompression process is reverse of compression process using string marching algorithm. During decompression, the decompressor does not require to search through the dictionary. As each code is encountered, it is translated into corresponding character string to produce output.

CONCLUSION

The objectives of this project had been achieved through simulation study. The simulation result shows that overall LZSS compression algorithm is an efficient algorithm compared to LZW compression algorithm. LZSS achieved high percentage of compressibility, compression speed and decompression speed. LZSS algorithm able to compress or decompress a file faster compared to LZW algorithm. However, when compressing pic file, LZW algorithm gives better result. The reason for this is the uncommon content of this file which contains a lot of nulls. LZW algorithm can have a pointer to an infinite string, while LZSS only have pointer to a finite string.

If there is a long sequence of the same character, LZW can compress it in a constant few bytes assuming the length component is long enough to grip the number of the characters. LZSS, however, has to construct the pointers step by step and it will have pointers to two bytes or three bytes.

The compression time increases as the dictionary size increases. During compression, searching for a match in a large dictionary requires large amount of time, thus increases the compression time. The compressed size (%) increases as the dictionary size increases. The percentage of compression achieved is high with large size dictionary.

Data compression provide increased network throughput without an increase in transmission channel bandwidth. Compressing data allows a user to keep more information in the system memory. The importance of compression gets much more prominent when downloading files as the available network bandwidth has not kept pace with the size of applications or physically transporting files as the storage capacity of magnetic storage devices, like floppy disks, have not kept pace with the size of applications.

One future direction of this project is to enhance the LZSS compression algorithm in order to achieve high compressibility for image compression. The result obtained from this study shows LZSS algorithm compress better except for images. So, the LZSS algorithm could be studied in term of the length size and pointer variation for better image compression.

REFERENCES

1. Navarro, G. and M. Ranot, 1999. A general practical approach to pattern matching over Ziv-Lempel compressed text. Proceeding of 10th Annual Symposium on Combinatorial Pattern Matching, pp: 14-36.
2. Matias, Y. and T. Sahinalp, 1999. On optimality of parsing in dynamic dictionary based data compression. ACM SIAM Symposium on Discrete Algorithms.
3. Knieser, M., G. Wolff, A. Papachristou, J. Weyer and R. McIntyr, 2003. A technique for high ratio LZW compression. IEEE Transactions on Communications.
4. Kasera, S. and N. Jain, 2005. A survey of lossless compression techniques.
5. Abali, B., H. Franke, X. Shen, D. Poff and B. Smith, 2001. Performance of hardware compressed main memory. Proceedings of 7th International Symposium on High-Performance Computer Architecture (HPCA), pp: 73-81, January 2001.
6. Ziv, J. and A. Lempel, 1977. A universal algorithm for sequential data compression. IEEE Trans. Inform. Theory, 337-349.
7. Ziv, J. and A. Lempel, 1978. Compression of individual sequences via variable-rate coding. IEEE Trans. Inform. Theory, pp: 530-536.
8. Willard, L., A. Lempel, J. Ziv and M. Cohn, 1984. Apparatus and method for compressing data signals and restoring the compressed data signals. US patent - US4464650, 1984.
9. Storer, J.A. and T.G. Szymanski, 1982. Data compression via textual substitution. J. ACM, 928-951.
10. Welch, T.A., 1984. A technique for high performance data compression. IEEE Comput., 8-19.
11. Bell, T.C., I.H. Witten and J.G. Cleary, 1988. Modeling for text compression. Technical Report, The University of Calgary, Calgary, Alberta Canada, pp: 327-39.
12. Horspool, R.N., 1991. Improving LZW. Proceeding Data Compression Conference (DCC 91), Snowbird, Utah, IEEE Computer Society Press, Los Alamitos, CA, pp: 332-341.
13. Tao, T. and A. Mukherjee, 2004. LZW based compressed pattern matching. Proceeding Data Compression Conference (DCC '04), pp: 568-601.
14. Klein, S.T. and Y. Wiseman, 2000. Parallel Huffman decoding. Proceeding Data Compression Conference DCC-2000, Snowbird, Utah, pp: 383-392.
15. Hirschberg, D.S. and L.M. Stauffer, 1994. Parsing algorithms for dictionary compression on the PRAM. IEEE Comput. Society Press, pp: 136-145.
16. De Agostino, S. and J.A. Storer, 1995. Near optimal compression with respect to a static dictionary on a practical massively parallel architecture. IEEE Comput. Society Press, pp: 172-181.
17. De Agostino, S. and J.A. Storer, 1992. Parallel algorithms for optimal compression using dictionaries with the prefix property. IEEE Comput. Society Press, pp: 52-61.
18. Ekman, M. and P. Stenstrom, 2005. A Robust Main Memory Compression Scheme. International Symposium on Computer Architecture.
19. Kenneth, B. and C. Krste, 2006. Energy aware lossless data compression. ACM Trans. Comput. Syst., 24: 250-291.
20. Motgi, N. and A. Mukherjee, 2001. Network conscious text compression systems (NCTCSys). Proceedings of the International Conference on Information and Theory: Coding and Computing.
21. William S., 2002. High-Speed Networks and Internets: Performance and Quality of Service. 2nd Edn., Prentice Hall.
22. Yair W., 2004. The relative efficiency of data compression by lzw and lzss. Computer Science Department, Bar-Ilan University.
23. Franti, P., E. Ageenko, P. Kopylov, S. Grohn and F. Berger, 2001. Compression of map images for real-time applications. Research Report A-2001-1, Department of Computer Science, University of Joensuu.
24. Michihiro, S., O. Toshihiro, I. Hideyuki and I. Tomoo, 2005. A huffman-based coding with efficient test application. IEEE Comput.
25. Michael, J., G. Francis, D. Papachristou and D. McIntyre, 2003. A technique for high ratio LZW compression. IEEE Comput.