Final Report for Period: 10/1999 - 09/2003 **Submitted on:** 09/29/2003
Principal Investigator: Mukherjee, Amar . **Award ID:** 9977336
Organization: U of Central Florida

Title:
Algorithms to Improve the Efficiency of Data Compression and Caching on Wide-Area Networks

Project Participants

Senior Personnel

Name: Mukherjee, Amar

Worked for more than 160 Hours: Yes

Contribution to Project:

Professor Amar Mukherjee is the Principal Investigator of this project and is in charge of all the reserach and targetted activities and guidance of research assistants working under this project.

Post-doc

Graduate Student

Name: Zhang, Nan

Worked for more than 160 Hours: Yes

Contribution to Project:

Nan Zhang has been working as a Graduate Research Assistant with the project. He is working on developing compression algorithms and a theory of transforms developed under this project. He is also reading literature on compressed domain search problem to come up with a formulation of a problem area for doctoral dissertation. He has been supported by this grant in the past.

Name: Motgi, Nitin

Worked for more than 160 Hours: Yes

Contribution to Project:

Nitin Motgi was involved in the networking and infrastructure development aspects of the project. He worked on setting up an online compression utility webpage as a test bench for various comnpression algorithms and also worked on compressed data transmission infrasturucture tools. Nitin also worked on the development of new lossless compression algorithms for text. He has been supported
in this research grant during Fall of 2000. He has now accepted a job and is not working on this project.

Name: Awan, Fauzia

Worked for more than 160 Hours: Yes

Contribution to Project:

Ms. Fauzia Awan was a student in the gaduate level Multimedia Data Compression course that I taught Spring of 2000 and did a term project related to this project. Since then she got interested doing a MS thesis under this project and worked as a Reserach Assistant in the prtoject for one year. She defended her thesis summer of 2001.
She has now accepted a job outside.

Name: Satya, Ravi

Worked for more than 160 Hours: No

Contribution to Project:

Mr. Ravi Vijya Satya is a graduate student who participates in this reserach project. He has participated in many reserach seminars of the M5 group and contributed in the lively discussions on several aspects of the project. His major interest is now Bioinformatics and DNA sequence compression and analysis.

Name: Tao, Tao

Worked for more than 160 Hours: No

Contribution to Project:

Mr Tao Tao is interested in compressed domain pattern matching for both text and images. Since his work also involves sorted context and BWT and other transforms, he interacts with other members of the M5 Reserach group and has contributed in many discussion seminars directly related to the current project.

Name: Sun, Weifeng

Worked for more than 160 Hours: Yes

Contribution to Project:

Mr. Weifeng Sun is a new Ph. D. student supported by the Department of Computer Science here at UCF. He has joined our group and is conducting research on lossles text compression algorithms. His main contribution is the work on the StarNT transform.

Name: Iqbal, Raja

Worked for more than 160 Hours: No

Contribution to Project:

Mr. Iqbal participated in implementing some parts of the Star transform used for our data compression algorithm and collaborated with Fauzia Awan in the preliminary development of the LIPT transform.

Name: Kruse, Holger

Worked for more than 160 Hours: No

Contribution to Project:

Holger Kruse was associated in this reserach in the early phase of writing the grant proposal and a joint paper with Mr. Kruse was published. Mr. Kruse left UCF after one semester since the beginning of the project.

Undergraduate Student

Technician, Programmer

Other Participant

Name: Franceschini, Robert

Worked for more than 160 Hours: Yes

Contribution to Project:

Dr. Franceschini worked for a period of approximately three months during summer of 2000 as
a Research Associate (post-doctoral) with support from matching funds under this project from the University of Central Florida. He was assisting the Graduate Reserach Assistants and was involved in producing the draft of a paper along with multiple authors for the project. In the second year, he was not involved in the project work at all, except that he serves as a member of the faculty committee for thesis done under this project. He has left UCF and accepted a job with a company.

Research Experience for Undergraduates

Organizational Partners

Other Collaborators or Contacts

I have been in touch with two well-known researchers in the data compression field: Tim Bell of Computer Science Department, University of Canterbury, New Zealand and Don Adjeroh of the Department of Computer Science and Electrical Engineering, West Virginia University. We have been working on several joint papers on compressed domain pattern matching and two research proposals have been awarded by NSF, the latest being a Small ITR proposal that was awarded effective September 15,2003. We have developed an online compression utility website (vlsi.cs.ucf.edu)under the current project. We plan to link this up with the Canterbury website under the new ITR initiative.

Activities and Findings

Research and Education Activities: (See PDF version submitted by PI at the end of the report)

Project Summary

The goal of this research project is to develop new lossless text compression algorithms and software tools to incorporate compression for archival storage and transmission over the Internet. The approach consists of pre-processing the text to exploit the natural redundancy of English language to obtain an intermediate transformed form via the use of a dictionary and then compressing it using existing compression algorithms. Several classical compression algorithms such as Huffman, arithmetic, LZ-family (gzip and compress) as well as some of the recent algorithms such as Bzip2, PPM family, DMC, YBS, DC, RK, PPMonstr and recent versions of Bzip2 are used as the backend compression algorithms. The performance of our transforms in combination with these algorithms are compared with the original set of algorithms, taking into account both compression, computation and storage overhead. Information theoretic explanation of experimental results are given. The impact of the research on the future of information technology is to develop data delivery systems with efficient utilization of communication bandwidth and conservation of archival storage. We also develop infrastructure software for rapid delivery of compressed data over the Internet and an online compression utility website as a test bench for comparing various kinds of compression algorithms. The site (vlsi.cs.ucf.edu) will be linked to a very well known compression website which contains the Canterbury and Calgary text corpus. The experimental research is linked to educational goals by rapid dissemination of results via reports, conference and journal papers and doctoral dissertation and master's thesis, and transferring the research knowledge into the graduate curriculum via teaching of a graduate level course on data compression.

Goals and Objectives

The goal of this research project is to develop new lossless text compression algorithms and software tools to incorporate compression for archival storage and transmission over the Internet. Specific objectives for this period were:

1. Development of new lossless text compression algorithms.

2. Development of software tools to incorporate compression in text transmission over the Internet and on-line compression utility for a compression test bench.

3. Measurement of performance of the algorithms taking into account both compression and communication metrics.

4. Development of a theory to explain the experimental results based on information theoretic approach.

Executive Summary

The basic philosophy of our compression algorithm is to transform the text into some intermediate form, which can be compressed with better efficiency. The transformation is designed to exploit the natural redundancy of the language. We have developed a class of such transformations each giving better compression performance over the previous ones and all of them giving better compression over most of the current and classical compression algorithms (viz. Huffman, Arithmetic and Gzip (based on LZ77), Bzip2 (based on Burrows ûWheeler Transform), the class of PPM (Partial Predicate Match) algorithms (such as PPMD), RK, DC, YBS and PPMonstr). We also measured the execution times needed to produce the pre-processing and its impact on the total execution time. We developed several transforms (Star(*) and LPT) and two variations of LPT called RLPT and SCLPT. We developed four new transforms called LIPT, ILPT, LIT and NIT, which produce better results in terms of both compression ratio and execution times. The algorithms use a fixed amount of storage overhead in the form of a word dictionary for the particular corpus of interest and must be shared by the sender and receiver of the compressed files. Typical size of dictionary for the English language is about 1 MB and can be downloaded once along with application programs. If the compression algorithms are going to be used over and over again, which is true in all practical applications, the amortized storage overhead is negligibly small. We also develop efficient data structures to expedite access to the dictionaries and propose memory management techniques using caching for use in the context of the Internet technologies. Realizing that certain on-line algorithms might prefer not to use a pre-assigned dictionary, we have been developing new algorithms to obtain the transforms dynamically with no dictionary, with small dictionaries (7947 words and 10000 words) and studying the effect of the size of the dictionaries on compression performance. We call this family of algorithms M5zip.

During the final year, we developed a new transform engine StarNT for a multi-corpora text compression system. StarNT achieves a superior compression ratio than almost all the other recent efforts based on BWT and PPM. StarNT is a dictionary-based fast lossless text transform. The main idea is to recode each English word with a representation of no more than three symbols. This transform not only maintains most of the original context information at the word level, but also provides some kind of 'artificial' but strong context. The transform also exploits the distribution of lengths of the words and frequency to reduce the size of the transformed text which is provided to a backend compressor. Ternary search tree is used to store the transform dictionary in the transform encoder. This data structure provides a very fast transform encoding with a low storage overhead. Another novel idea of StarNT is to treat the transformed codewords as the offset of words in the transform dictionary. Thus the time complexity of O(1) for searching a word in the dictionary is achieved in the transform decoder.

The transform encoder and transform decoder share the same dictionary, which is prepared off-line according to the following rules: 1) Most frequently used words (i.e. 312 words) are listed in the beginning of the dictionary according to their frequencies. 2) Other words are sorted

according to their lengths and frequencies. Words with longer lengths are stored after words with shorter lengths. If two words have the same length, word with higher frequency of occurrence is listed after word with lower frequency. 3) To achieve better compression performance for backend data compressor, only letters [a..zA..Z] are used to represent the codeword. One other angle of study is to adapt dynamically to domain-specific corpus (viz. biological, physics, computer science, XML documents, html documents). We experimentally measure the performance of our proposed algorithms and compare with all other algorithms using three corpuses: Calgary, Canterbury and Gutenberg corpus. Finally, we develop an information theory based explanation of the performance of our algorithms.

We have also started work on a new lossless text compression algorithm called M5Zip which obtains the transformed version of the text dynamically with no dictionary or with small dictionaries (7947 words and 10000 words).The transformed text is passed through a pipe of BWT transform, inversion frequency vector, run length encoding and arithmetic coding. Our results indicate that the algorithm achieves 11.65% improvement over Bzip2 and 5.95% improvement over LIPT plus Bzip2. The investigation on this class of algorithms will continue for future proposals and papers.

In the 'Activities Attached File' we attach two papers: 1) a paper describing the family of Star compression algorithms and 2) a paper describing the details of StarNT family of compression algorithms.

Findings:

Major Findings

The major findings for this reporting period are as follows.

The basic philosophy of our compression algorithm is to transform the text into some intermediate form, which can be compressed with better efficiency. The transformation is designed to exploit the natural redundancy of the language. We have developed a class of such transformations each giving better compression performance over the previous ones and all of them giving better compression over most of the current and classical compression algorithms (viz. Huffman, Arithmetic and Gzip (based on LZ77), Bzip2 (based on Burrows ûWheeler Transform), the class of PPM (Partial Predicate Match) algorithms (such as PPMD), RK, DC, YBS and PPMonstr). We also measured the execution times needed to produce the pre-processing and its impact on the total execution time. We call this family of compression algorithm 'Star compression' algorithms which includes *-transform, LPT, LIPT, RLPT, SCLPT, LIT, NIT . We then improve upon our original ideas and develop a new transform StarNT.

StarNT achieves a superior compression ratio than almost all the other recent efforts based on BWT and PPM. StarNT is a dictionary-based fast lossless text transform. The main idea is to recode each English word with a representation of no more than three symbols. This transform not only maintains most of the original context information at the word level, but also provides some kind of 'artificial' but strong context. The transform also exploits the distribution of lengths of the words and frequency to reduce the size of the transformed text which is provided to a backend compressor. Ternary search tree is used to store the transform dictionary in the transform encoder. This data structure provides a very fast transform encoding with a low storage overhead. Another novel idea of StarNT is to treat the transformed codewords as the offset of words in the transform dictionary. Thus the time complexity of O(1) for searching a word in the dictionary is achieved in the transform decoder.

The transform encoder and transform decoder share the same dictionary, which is prepared off-line according to the following rules: 1) Most frequently used words (i.e. 312 words) are listed in the beginning of the dictionary according to their frequencies. 2) Other words are sorted according to their lengths and frequencies. Words with longer lengths are stored after words with shorter lengths. If two words have the same length, word with higher frequency of occurrence is listed after word with lower frequency. 3) To achieve better compression performance for backend data compressor, only letters [a..zA..Z] are used to represent the codeword.

Experimental results show that the average compression time has improved by orders of magnitude compared to our previous dictionary based transform LIPT and for large files, viz. 400Kbytes or more, the compression time is no worse than those obtained by bzip2 and gzip, and is much faster than PPMD. Meanwhile, the overhead in the decompression phase is negligible. We draw a significant conclusion that bzip2 in conjunction with this transform is better than both gzip and PPMD both in time complexity and compression performance.

One of the key features of StarNT compression system is to develop domain specific dictionaries and provide tools to develop such dictionaries. Results from five corpora show that StarZip achieves an average improvement in compression performance (in terms of BPC) of 13% over bzip2 -9, 19% over gzip -9, and 10% over PPMD. Further details about the StarNT transform and experimental results can be found in the attached document.

We developed compressed domain pattern matching algorithms based on sorted context and laid the foundation of a future research proposals in this field. The papers are concerned with searching for pattern in the compressed text without or only partial decompression. The sorted context of the BWT transform gives rise to very efficient binary and q-gram based search strategy for performing exact and approximate pattern matching operations. A Small ITR proposal to NSF based on this research foundation has been awarded.

Training and Development:

Six Ph.D. students and three Masters students have participated and contributed in this research project, but not all of them received direct support from the grant. Dr. Robert Franceschini and Mr. Holger Kruse acquired valuable reseach experience working on this project and making some early contributions. A Masters student Ms. Fauzia Awan has defended her thesis and has graduated summer of 2001. One Masters student Mr. Raja Iqbal briefly collaborated with Ms. Awan in her research. Mr. Nitin Motgi worked for about a year on the M5Zip project and then accepted a job outside. Four Ph. D. students (Mr. Nan Zhang and Ravi Vijaya Satya, Mr. Tao Tao and Mr. Weifeng Sun) have worked on the project at different phases of the project. We have formed a research group called M5 Research Group which has been meeting weekly or bi-weekly to discuss reserach problems and make presentations on their work. This gives the students experience of teaching graduate level courses and seminars. The overall effect of these activities is to train graduate students with the current research on the forefront of technology. Each one of them acquired valuable experience in undertaking significant programming tasks.

Outreach Activities:

Journal Publications

Tim Bell, Don Adjeroh and Amar Mukherjee, "Pattern Matching in Compressed Text and Images", ACM Computing Survey, p. , vol. , ( ). Submitted

F. Awan and Amar Mukherjee, "LIPT: A Lossless Text Transform to Improve Compression", Proceedinds of the International Conference on Information Technology:Coding and Communication (ITCC2000), p. 452, vol. , (2001). Published

N. Motgi and Amar Mukherjee, "Network Conscious Text Compression System (NCTCSys)", Proceedings of the International Conference on Information Technology:Coding and Computing (ITCC2001), p. 440, vol. , (2001). Published

Fauzia Awan, Ron Zhang, Nitin Motgi,Raja Iqbal and Amar Mukherjee
, "LIPT: A Reversible Lossless Text Transform to Improve Compression Performance", Proc. Data Compression Conferemce, p. 311, vol. , (2001). Published

M. Powell, Tim Bell, Amar Mukherjee and Don Adjeroh, "Searching BWT Compressed Text with the Boyer-Moore Algorithm and Binary Search", Proc. Data Compression Conference (Eds. J.A.Storer and M. Cohn), p. 112, vol. , (2002). Published

D. Adjeroh, A. Mukherjee, T. Bell, M. Powell and N.Zhang, "Pattern Matching in BWT-transformed Text", Proc. Data Compression Conference, p. 445, vol. , (2002). Published

Weifeng Sun, Nan Zhang and Amar Mukherjee, "A Dictionary-Based Multi-corpora Text Compression System", Proceedings of the Data Compression Conference, p. 448, vol. , (2003). Published

Weifeng Sun, Nan Zhang and Amar Mukherjee, "Dictionary-based Fast Text Transform for Text Compresion", Proceedings International Conference on Information Technology: Coding and Computing, p. 176, vol. , (2003). Published

Books or Other One-time Publications

Amar Mukherjee and Fauzia Awan, "Text Compression", (2003). Book, Published
Editor(s): Khalid Sayood
Collection: Lossless Compression Handbook
Bibliography: 0-12-620861-1 Elsvier Science/Academic Press

Web/Internet Site

URL(s):

http://vlsi.cs.ucf.edu/

Description:

This site is for the M5 Reserach Group and the VLSI System Reserach Laboratory under the direction of Professor Amar Mukherjee. A pointer from this site leads to a site relevant to this reserach grant. There is also a pointer to our new "online compression utility".

Other Specific Products

Contributions

Contributions within Discipline:

We expect that our research will impact the future status of information technology by developing data delivery systems with efficient utilization of communication bandwidths and archival storage. We have developed new lossless text compression algorithms that have improved compression ratio over the best known existing compression algorithms which might translate into a reduction of 75% text traffic on the Internet. We have developed an online compression utility software that will allow an user to submit any text file and obtain compression statistics of all the classical and new compression algorithms. The URL for this is: vlsi.cs.ucf.edu.

Contributions to Other Disciplines:

Contributions to Human Resource Development:

So far six Ph.D. students and three Masters students have participated and contributed in this research project, but not all of them received direct support from the grant. Dr. Robert Franceschini and Mr. Holger Kruse made contributions in the project before it was officially funded by NSF. A Masters student Ms. Fauzia Awan made significant contributions and successfully defended her thesis. A Masters student Mr. Raja Iqbal worked on this project for a brief period of time and collaborated with Ms. Awan in her reserach. Mr. Nitin Motgi worked on the M5Zip project Currently, four Ph. D. students (Nan Zhang, Ravi Vijaya, Tao Tao and Weifeng Sun) are working on the project. Mr. Tao Tao who finished his Masters thesis three years ago has joined our reserach team. Other members of the M5 Research Group at the School of Electrical Engineering and Computer Science, Dr. Kunal Mukherjee and Mr.Piyush Jamkhandi, made critical comments and observation during the course of this work. The overall effect of these activities is to train graduate students with the current research on the forefront of technology.

Contributions to Resources for Research and Education:

We have taught (in the spring 2000 semester) a new special-topics graduate level course entitled 'Multimedia Compression on the Internet'. The course was taught again spring of 2001 with a new number CAP5015. This has a new URL location: http://www.cs.ucf.edu/courses/cap5015/. This is a graduate level course and 14 students enrolled in the Spring 2000 semester and 11 students enrolled in the Spring 2001. The course was again offered Fall 2002 with 25 students and is offered in Fall 2003 with 22 students. The course has now becoming one of the popular and regular graduate level courses in the department of Computer Science at University of Central Florida. This particular topic has grown directly out of the research that we have been conducting for the last four years on data compression. Lecture topics have included both text and image compression,including topics from the research on the current NSF grant.

A special feature article 'Star Encoding' describing our work on Star compression appeared in the Dobb's Journal,pp.94-96, August, 2004. The article was written by the Editor of the Journal, Dr. Mark Nelson.

The PI also delivered invited talks on research supported by this grant and in general on lossles text compression at universities in U.S. (University of California at Santa Barbara, San Diego, Riverside, Santa Cruz and Oregon State University) and abroad (Indian Institute of Technology, Kharagpur and Indian Statistical Institue , Kolkata during 2001).

The PI also gave a demonstration of his work on data compression and the online compression utility web site at the IDM Workshop, 2001, Ft. Worth, Texas (April 29-30) sponsored by NSF.

Contributions Beyond Science and Engineering:

Categories for which nothing is reported:

Organizational Partners

Activities and Findings: Any Outreach Activities

Any Product

Contributions: To Any Other Disciplines

Contributions: To Any Beyond Science and Engineering

Text Compression

Amar Mukherjee and Fauzia Awan
School of Electrical Engineering and Computer Science, University of Central Florida
amar@cs.ucf.edu

1. Introduction

In recent times, we have seen an unprecedented explosion of textual information through the use of Internet, digital libraries and information retrieval system. The advent of office automation systems, newspapers, journals and magazine repositories have brought the issue of maintaining archival storage for search and retrieval to the forefront of research. As an example, the TREC [TREC00] database holds around 800 million static pages having 6 trillion bytes of plain text equal to the size of a million books. Text compression is concerned with techniques for representing the digital text data in alternate representations that takes less space. Not only it helps conserve the storage space for archival and online data, it also helps system performance by requiring less number of secondary storage (disk or CD Rom) access and improves the network transmission bandwidth utilization by reducing the transmission time. Data compression methods are generally classified as *lossless or lossy*. Lossless compression allows the original data to be recovered exactly. Although used primarily for text data, lossless compression algorithms are useful in special classes of images such as medical imaging, finger print data, astronomical images and data bases containing mostly vital numerical data, tables and text information. In contrast, lossy compression schemes allow some deterioration and are generally used for video, audio and still image applications. The deterioration of the quality of lossy images are usually not detectable by human perceptual system, and the compression systems exploit this by a process called '*quantization*' to achieve compression by a factor of 10 to a couple of hundreds. Many lossy algorithms use lossless methods at the final stage of the encoding stage underscoring the importance of lossless methods for both lossy and lossless compression applications. This chapter will be concerned with lossless algorithms for textual information. We will first review in Section 2 the basic concepts of information theory applicable in the context of lossless text compression. We will then briefly describe the well-known compression algorithms in Sections 3. Detailed description of these and other methods are now available in several excellent recent books [Sal00; Say00; WiMB99; GBLL98 and others]. In Sections 4 through 6, we present our own research on text compression. In particular, we present a number of pre-processing techniques that transform the text in some intermediate forms that produce better compression performance. We give extensive test results of compression and timing performance with respect to text files in three corpuses: Canterbury, Calgary [Cant00] and Gutenberg [Gute71] corpus, along with discussion on storage overhead. In Section 7, we present a plausible information theoretic explanation of the compression performance of our algorithms. We conclude our chapter with a discussion of the compression utility website that is now integrated with Canterbury website for availability via the Internet.

2. Information Theory Background

The general approach to text compression is to find a representation of the text requiring less number of binary digits. In its uncompressed form each character in the text is represented by an 8-bit ASCII code[1]. It is common knowledge that such a representation is not very efficient because it treats frequent and less frequent characters equally. It makes intuitive sense to encode frequent characters with a smaller (less than 8) number of bits and less frequent characters with larger number of bits (possibly more than 8 bits) in order to reduce the *average* number of *bits per character* (BPC). In fact this principle was the basis of the invention of the so-called Morse code and the famous Huffman code developed in the early 50's. Huffman code typically reduces the size of the text file by about 50-60% or provides compression rate of 4-5 BPC [WiMB99] based on statistics of frequency of characters. In the late 1940's, Claude E. Shannon laid down the foundation of the information theory and modeled the text as the output of a source that generates a sequence of symbols from a finite alphabet A according to certain probabilities. Such a process is known as a *stochastic process* and in the special case when the probability of occurrence of the next symbol in the text depends on the previous symbols or its context it is called a *Markov process*. Furthermore, if the probability distribution of a typical sample represents the distribution of the text it is called an *ergodic process* [Sh48, ShW98]. The information content of the text source can then be quantified by the entity called *entropy H* given by

$$H = -\sum p_i \log p_i \qquad (1)$$

where p_i denotes the probability of occurrence of the ith symbol in the text, sum of all symbol probabilities is unity and the logarithm is with respect base 2 and $-\log p_i$ is the amount of *information* in bits for the event (occurrence of the ith symbol). The expression of H is simply the sum of the number of bits required to represent the symbols multiplied by their respective probabilities. Thus the entropy H can be looked upon as defining the *average number of BPC* required to represent or encode the symbols of the alphabet. Depending on how the probabilities are computed or modeled, the value of entropy may vary. If the probability of a symbol is computed as the ratio of the number of times it appears in the text to the total number of symbols in the text, the so-called *static* probability, it is called an Order(0) model. Under this model, it is also possible to compute the *dynamic* probabilities which can be roughly described as follows. At the beginning when no text symbol has emerged out of the source, assume that every symbol is equiprobable[2]. As new symbols of the text emerge out of the source, revise the probability values according to the actual frequency distribution of symbols at that time. In general, an Order(k) model can be defined where the probabilities are computed based on the probability of distribution of the ($k+1$)-grams of symbols or equivalently, by

[1] Most text files do not use more than 128 symbols which include the alphanumerics, punctuation marks and some special symbols. Thus, a 7-bit ASCII code should be enough.

[2] This situation gives rise to what is called the zero-frequency problem. One cannot assume the probabilities to be zero because that will imply an infinite number of bits to encode the first few symbols since –log o is infinity. There are many different methods of handling this problem but the equiprobabilty assumption is a fair and practical one.

taking into account the context of the preceding k symbols. A value of k = -1 is allowed and is reserved for the situation when all symbols are considered equiprobable, that is, $p_i = \frac{1}{|A|}$, where $|A|$ is the size of the alphabet A. When $k=1$ the probabilities are based on *bigram* statistics or equivalently on the context of just one preceding symbol and similarly for higher values of k. For each value of k, there are two possibilities, the static and dynamic model as explained above. For practical reasons, a static model is usually built by collecting statistics over a test *corpus* which is a collection of text samples representing a particular domain of application (viz. English literature, physical sciences, life sciences, etc.). If one is interested in a more precise static model for a given text, a *semi-static* model is developed in a two-pass process; in the first pass the text is read to collect statistics to compute the model and in the second pass an encoding scheme is developed. Another variation of the model is to use a specific text to *prime* or seed the model at the beginning and then build the model on top of it as new text files come in.

Independent of what the model is, there is an entropy associated with each file under that model. Shannon's fundamental noiseless source coding theorem says that entropy defines a lower limit of the average number of bits needed to encode the source symbols [ShW98]. The "worst" model from information theoretic point of view is the order(-1) model, the equiprobable model, giving the maximum value H_m of the entropy. Thus, for the 8-bit ASCII code, the value of this entropy is 8 bits. The redundancy R is defined to be the difference[3] between the maximum entropy H_m and the actual entropy H. As we build better and better models by going to higher order k, lower will be the value of entropy yielding a higher value of redundancy. The crux of lossless compression research boils down to developing compression algorithms that can find an encoding of the source using a model with minimum possible entropy and exploiting maximum amount of redundancy. But incorporating a higher order model is computationally expensive and the designer must be aware of other performance metrics such as decoding or decompression complexity (the process of decoding is the reverse of the encoding process in which the redundancy is restored so that the text is again human readable), speed of execution of compression and decompression algorithms and use of additional memory.

Good compression means less storage space to store or archive the data, and it also means less bandwidth requirement to transmit data from source to destination. This is achieved with the use of a *channel* which may be a simple point-to-point connection or a complex entity like the Internet. For the purpose of discussion, assume that the channel is noiseless, that is, it does not introduce error during transmission and it has a *channel capacity* C which is the maximum number of bits that can be transmitted per second. Since entropy H denotes the average number of bits required to encode a symbol, C/H denotes the average number of symbols that can be transmitted over the channel per second [ShW98]. A second fundamental theorem of Shannon says that however clever you may get developing a compression scheme, you will never be able to transmit on

[3] Shannon's original definition is R/H_m which is the fraction of the structure of the text message determined by the inherent property of the language that governs the generation of specific sequence or words in the text [ShW98].

average more than C/H symbols per second [ShW98]. In other words, to use the available bandwidth effectively, *H* should be as low as possible, which means employing a compression scheme that yields minimum BPC.

3. Classification of Lossless Compression Algorithms

The lossless algorithms can be classified into three broad categories : *statistical methods*, *dictionary methods* and *transform based methods*. We will give a very brief review of these methods in this section.

3.1 Statistical Methods

The classic method of statistical coding is Huffman coding [Huff52]. It formalizes the intuitive notion of assigning shorter codes to more frequent symbols and longer codes to infrequent symbols. It is built bottom-up as a binary tree as follows: given the model or the probability distribution of the list of symbols, the probability values are sorted in ascending order. The symbols are then assigned to the leaf nodes of the tree. Two symbols having the two lowest probability values are then combined to form a parent node representing a composite symbol which replaces the two child symbols in the list and whose probability equals the sum of the probabilities of the child symbols. The parent node is then connected to the child nodes by two edges with labels '0' and '1'in any arbitrary order. The process is now repeated with the new list (in which the composite node has replaced the child nodes) until the composite node is the only node remaining in the list. This node is called the root of the tree. The unique sequence of 0's and 1's in the path from the root to the leaf node is the Huffman code for the symbol represented by the leaf node. At the decoding end the same binary tree has to be used to decode the symbols from the compressed code. In effect, the tree behaves like a dictionary that has to be transmitted once from the sender to receiver and this constitute an initial overhead of the algorithm. *This overhead is usually ignored in publishing the BPC results for Huffman code in literature*. The Huffman codes for all the symbols have what is called the *prefix property* which is that no code of a symbol is the prefix of the code for another symbol, which makes the code *uniquely decipherable(UD)*. This allows forming a code for a sequence of symbols by just concatenating the codes of the individual symbols and the decoding process can retrieve the original sequence of symbols without ambiguity. Note that a prefix code is not necessarily a Huffman code nor may obey the Morse's principle and a uniquely decipherable code does not have to be a prefix code, but the beauty of Huffman code is that it is UD, prefix and is also optimum within one bit of the entropy H. Huffman code is indeed optimum if the probabilities are $1/2^k$ where k is a positive integer. There are also Huffman codes called *canonical* Huffman codes which uses a look up table or dictionary rather than a binary tree for fast encoding and decoding [WiMB99,Sal00].

Note in the construction of the Huffman code, we started with a model. Efficiency of the code will depend on how good this model is. If we use higher order models, the entropy will be smaller resulting in shorter average code length. As an example, a word–based Huffman code is constructed by collecting the statistics of words in the text and building

a Huffman tree based on the distribution of probabilities of words rather than the letters of the alphabet. It gives very good results but the overhead to store and transmit the tree is considerable. Since the leaf nodes contain all the distinct words in the text, the storage overhead is equal to having an English words dictionary shared between the sender and the receiver. We will return to this point later when we discuss our transforms. Adaptive Huffman codes takes longer time for both encoding and decoding because the Huffman tree has to be modified at each step of the process. Finally, Huffman code is sometimes referred to as a variable length code (VLC) because a message of a fixed length may have variable length representations depending on what letters of the alphabet are in the message.

In contrast, the *arithmetic code* encodes a variable size message into fixed length binary sequence[RiL79]. Arithmetic code is inherently adaptive, does not use any lookup table or dictionary and in theory can be optimal for a machine with unlimited precision of arithmetic computation. The basic idea can be explained as follows: at the beginning the semi-closed interval [0,1) is partitioned into $|A|$ equal sized semi-closed intervals under the equiprobabilty assumption and each symbol is assigned one of these intervals. The first symbol, say a_1 of the message can be represented by a point in the real number interval assigned to it. To encode the next symbol a_2 in the message, the new probabilities of all symbols are calculated recognizing that the first symbol has occurred one extra time and then the interval assigned to a_1 is partitioned (as if it were the entire interval) into $|A|$ sub-intervals in accordance with the new probability distribution. The sequence a_1a_2 can now be represented without ambiguity by any real number in the new sub-interval for a_2. The process can be continued for succeeding symbols in the message as long as the intervals are within the specified arithmetic precision of the computer. The number generated at the final iteration is then a code for the message received so far. The machine returns to its initial state and the process is repeated for the next block of symbol. A simpler version of this algorithm could use the same static distribution of probability at each iteration avoiding re-computation of probabilities. The literature on arithmetic coding is vast and the reader is referred to the texts cited above [Sal00; Say00;WiMB99] for further study.

The Huffman and arithmetic coders are sometimes referred to as the *entropy coder*. These methods normally use an order(0) model. If a good model with low entropy can be built external to the algorithms, these algorithms can generate the binary codes very efficiently. One of the most well known modeler is *"prediction by partial match" (PPM)* [ClW84; Moff90]. PPM uses a finite context Order(k) model where k is the maximum context that is specified ahead of execution of the algorithm. The program maintains all the previous occurrences of context at each level of k in a trie-like data structure with associated probability values for each context. If a context at a lower level is a suffix of a context at a higher level, this context is excluded at the lower level. At each level (except the level with k = -1), an *escape character* is defined whose frequency of occurrence is assumed to be equal to the number of distinct context encountered at that context level for the purpose of calculating its probability. During the encoding process, the algorithm estimates the probability of the occurrence of the *next character* in the text stream as follows: the algorithm tries to find the current context of maximum length k in the

context table or trie. If the context is not found, it passes the probability of the escape character at this level and goes down one level to *k-1* context table to find the current context of length *k-1*. If it continues to fail to find the context, it may go down ultimately to *k=-1* level corresponding to equiprobable level for which the probability of any next character is $1/|A|$. If a context of length q, $0<=q<=k$, is found, then the probability of the next character is estimated to be the product of probabilities of escape characters at levels $k, k-1, \ldots, q+1$ multiplied by the probability of the context found at the qth level. This probability value is then passed to the backend entropy coder (arithmetic coder) to obtain the encoding. Note, at the beginning there is no context available so the algorithm assumes a model with $k = -1$. The context lengths are shorter at the early stage of the encoding when only a few context have been seen. As the encoding proceeds, longer and longer context become available. In one version of PPM, called PPM*, an arbitrary length context is allowed which should give the optimal minimum entropy. In practice a model with $k = 5$ behaves as good as PPM* [ClTW95]. Although the algorithm performs very well in terms of high compression ratio or low BPC, it is very computation intensive and slow due to the enormous amount of computation that is needed as each character is processed for maintaining the context information and updating their probabilities.

Dynamic Markov Compression (DMC) is another modeling scheme that is equivalent to finite context model but uses finite state machine to estimate the probabilities of the input symbols which are bits rather than bytes as in PPM [CoH87]. The model starts with a single state machine with only one count of '0' and '1' transitions into itself (the zero frequency state) and then the machine adopts to future inputs by accumulating the transitions with 0's and 1's with revised estimates of probabilities. If a state is used heavily for input transitions (caused either by 1 or 0 input), it is *cloned* into two states by introducing a new state in which some of the transitions are directed and duplicating the output transitions from the original states for the cloned state in the same ratio of 0 and 1 transitions as the original state. The bit-wise encoding takes longer time and therefore DMC is very slow but the implementation is much simpler than PPM and it has been shown that the PPM and DMC models are equivalent [BeM89].

3.2 Dictionary Methods

The dictionary methods, as the name implies, maintain a *dictionary or codebook* of words or text strings previously encountered in the text input and data compression is achieved by replacing strings in the text by a reference to the string in the dictionary. The dictionary is *dynamic or adaptive* in the sense that it is constructed by adding new strings being read and it allows deletion of less frequently used strings if the size of the dictionary exceeds some limit. It is also possible to use a *static* dictionary like the word dictionary to compress the text. The most widely used compression algorithms (Gzip and Gif) are based on Ziv-Lempel or LZ77 coding [ZiL77] in which the text prior to the current symbol constitute the dictionary and a greedy search is initiated to determine whether the characters following the current character have already been encountered in the text before, and if yes, they are replaced by a reference giving its relative starting position in the text. Because of the pattern matching operation the encoding takes longer time but the process has been fine tuned with the use of hashing techniques and special

data structures. The decoding process is straightforward and fast because it involves a random access of an array to retrieve the character string. A variation of the LZ77 theme, called the LZ78 coding, includes one extra character to a previously coded string in the encoding scheme. A more popular variant of LZ78 family is the so-called LZW algorithm which lead to widely used *compress* algorithm. This method uses a suffix tree to store the strings previously encountered and the text is encoded as a sequence of node numbers in this tree. To encode a string the algorithm will traverse the existing tree as far as possible and a new node is created when the last character in the string fails to traverse a path any more. At this point the last encountered node number is used to compress the string up to that node and a new node is created appending the character that did not lead to a valid path to traverse. In other words, at every step of the process the length of the recognizable strings in the dictionary gets incrementally stretched and are made available to future steps. Many other variants of LZ77 and LZ78 compression family have been reported in the literature (See Sal00 and Say00 for further references).

3.3 Transform Based Methods: The Burrows-Wheeler Transform (BWT)

The word 'transform' has been used to describe this method because the text undergoes a transformation which performs a permutation of the characters in the text so that characters having similar lexical context will cluster together in the output. Given the text input, the forward Burrouws-Wheeler transform [BuWh94] forms all cyclic rotations of the characters in the text in the form of a matrix M whose rows are lexicographically sorted (with a specified ordering of the symbols in the alphabet). The last column L of this sorted matrix and an index r of the row where the original text appears in this matrix is the output of the transform. The text could be divided into blocks or the entire text could be considered as one block. The transformation is applied to individual blocks separately, and for this reason the method is referred to as *block sorting* transform [Fenw96]. The repetition of the same character in the block might slow down the sorting process; to avoid this, a run-length encoding (RLE) step could be preceded before the transform step. The Bzip2 compression algorithm based on BWT transform uses this step and other steps as follows: the output of the BWT transform stage then undergoes a final transformation using either move-to-front (MTF) [BSTW86] encding or distance coding (DC) [Arna00] which exploits the clustering of characters in the BWT output to generate a sequence of numbers dominated by small values (viz. 0,1 or 2) out of possible maximum value of $| A |$. This sequence of numbers is then sent to an entropy coder (Huffman or Arithmetic) to obtain the final compressed form. The inverse operation of recovering the original text from the compressed output proceeds by decoding the inverse of the entropy decoder, then inverse of MTF or DC and then an inverse of BWT. The inverse of BWT obtains the original text given (L , r). This is done easily by noting that the first column of M, denoted as F, is simply a sorted version of L. Define an index vector Tr of size $|L|$ such that $Tr[j]=i$ if and only if both $L[j]$ and $F[i]$ denote the kth occurrence of a symbol from A. Since the rows of M are cyclic rotations of the text, the elements of L precedes the respective elements of F in the text. Thus $F[Tr[j]]$ cyclically precedes $L[j]$ in the text which leads to a simple algorithm to reconstruct the original text.

3.4 Comparison of Performance of Compression Algorithms

An excellent discussion of performance comparison of the important compression algorithms can be found in [WiMB99]. In general, the performance of compression methods depends on the type of data being compressed and there is a tradeoff between compression performance, speed and the use of additional memory resources. The authors report the following results with respect to the Canterbury corpus: In order of increasing compression performance (decreasing BPC), the algorithms can be listed as order zero arithmetic, order zero Huffman giving over 4 BPC; the LZ family of algorithms come next whose performance range from 4 BPC to around 2.5 BPC (gzip) depending on whether the algorithm is tuned for compression or speed. Order zero word based Huffman (2.95 BPC) is a good contender for this group in terms of compression performance but it is two to three times slower in speed and needs a word dictionary to be shared between the compressor and decompressor. The best performing compression algorithms are bzip2 (based on BWT), DMC and PPM all giving BPC ranging from 2.4 to 2.1 BPC. PPM is theoretically the best but is extremely slow as is DMC, bzip2 strikes a middle ground, it gives better than gzip but is not an on-line algorithm because it needs the entire text or blocks of text in memory to perform the BWT transform. LZ77 methods (gzip) are fastest for decompression, then LZ78 technique, then Huffman coders, and the methods using arithmetic coding are the slowest. Huffman coding is better for static applications whereas arithmetic coding is preferable in adaptive and online coding. Bzip2 decodes faster than most of other methods and it achieves good compression as well. A lot of new research on bzip2 (see Section 4) has been carried on recently to push the performance envelope of bzip2 both in terms of compression ratio and speed and as a result bzip2 has become a strong contender to replace the popularity of gzip and compress.

New research is going on to improve the compression performance of many of the algorithms. However, these efforts seem to have come to a point of saturation regarding lowering the compression ratio. To get a significant further improvement in compression, other means like transforming the text before actual compression and use of grammatical and semantic information to improve prediction models should be looked into. Shannon made some experiments with native speakers of English language and estimated that the English language has entropy of around 1.3BPC [Sh51]. Thus, it seems that lossless text compression research is now confronted with the challenge of bridging a gap of about 0.8 BPC in terms of compression ratio. Of course, combining compression performance with other performance metric like speed, memory overhead and on-line capabilities seem to pose even a bigger challenge.

4. Transform Based Methods: Star(*), LIPT , LIT and NIT transforms

In this section we present our research on new transformation techniques that can be used as preprocessing steps for the compression algorithms described in the previous section. The basic idea is to transform the text into some intermediate form, which can be compressed with better efficiency. The transformation is designed to exploit the natural redundancy of the language. We have developed a class of such transformations, each

giving better compression performance over the previous ones and most of them giving better compression over current and classical compression algorithms discussed in the previous section. We first present a brief description of the first transform called Star Transform (also denoted by *-encoding). We then present four new transforms called LIPT, ILPT, LIT and NIT, which produce better results.

The algorithms use a fixed amount of storage overhead in the form of a word dictionary for the particular corpus of interest and must be shared by the sender and receiver of the compressed files. Word based Huffman method also make use of a static word dictionary but there are important differences as we will explain later. Because of this similarity, we specifically compare the performance of our preprocessing techniques with that of the word-based Huffman. Typical size of dictionary for the English language is about 0.5 MB and can be downloaded along with application programs. If the compression algorithms are going to be used over and over again, which is true in all practical applications, the amortized storage overhead for the dictionary is negligibly small. We will present experimental results measuring the performance (compression ratio, compression times, decompression times and memory overhead) of our proposed preprocessing techniques using three corpuses: Calgary, Canterbury and Gutenberg corpus. Finally, we present an information theory based explanation of the performance of our algorithms.

4.1 Star (*) Transformation

The basic idea underlying the star transformations is to define a unique signature of a word by replacing letters in a word by a special placeholder character (*) and keeping a minimum number of characters to identify the word uniquely [FrMu96]. For an English language dictionary D of size 60,000 words, we observed that we needed at most two characters of the original words to keep their identity intact. In fact, it is not necessary to keep any letters of the original word as long as an unique representation can be defined. The dictionary is divided into sub-dictionaries D_s containing words of length, $1<=s<=22$, because the maximum length of a word in English dictionary is 22 and there are two words of length 1 viz ‘a’ and ‘I’ (See Figure 1).

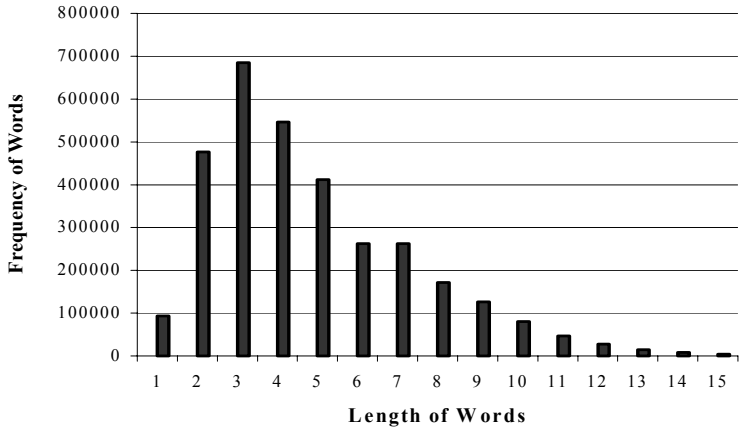


Figure 1: Frequency of English words versus length of words in the test corpus

The following encoding scheme is used for the words in D_s : The first word is represented as sequence of *s* stars. The next 52 words are represented by a sequence of *s-1* stars followed by a single letter from the alphabet $\sum$*=(a,b, ,z,A,B,…Z).* The next 52 words have a similar encoding except that the single letter appears in the last but one position. This will continue until all the letters occupy the first position in the sequence. The following group of words have *s-2* *'s and the remaining two positions are taken by unique pairs of letters from the alphabet. This process can be continued to obtain a total of 53^s unique encodings which is more than sufficient for English words. A large fraction of these combinations are never used; for example for *s* = 2, there are only 17 words and for *s*=8, there are about 9000 words in English dictionary. As an example of star encoding the following sentence: '*Our philosophy of compression is to transform the text into some intermediate form which can be compressed with better efficiency and which exploits the natural redundancy of the language in making this transformation'* can be *-encoded as **a ********** ** ********* *a *b ******** *** ***a ***c ***b ************ ***d ***** **b *c **********a ***e ****** *********a **c ***** ******* *** ******a *********b ** *** ******** *d *****a **** **********

There are exactly five two letter words in the sentence (of, is, to, be, in) which can be uniquely encoded as (**, *a, *b, *c, *d) and similarly for the other groups of words. Given such an encoding, the original word can be retrieved from the dictionary that contains a one-to-one mapping between encoded words and original words. The encoding produces an abundance of * characters in the transformed text making it the most frequently occurring character. If the word in the input text is not in the English dictionary (viz. a new word in the lexicon) it will be passed to the transformed text unaltered. The transformed text must also be able to handle special characters, punctuation marks and capitalization. The space character is used as word separator. The character '~' at the end of an encoded word denotes that the first letter of the input text word is capitalized. The character '`' denotes that all the characters in the input word are capitalized. A capitalization mask, preceded by the character '^', is placed at the end of encoded word to denote capitalization of characters other than the first letter and all capital letters. The character '\' is used as escape character for encoding the occurrences of '*','~','`','^', and '\' in the input text. The transformed text can now be the input to any available losslesss text compression algorithm, including Bzip2 where the text undergoes two transformation, first the *-transform and then a BWT transform.

4.2 LIPT:Length-Index Preserving Transform

A different twist to our transformation comes from the observation that the frequency of occurrence of words in the corpus as well as the predominance of certain lengths of words in English language might play an important role in revealing additional redundancy to be exploited by the backend algorithm. The frequency of occurrence of symbols, k-grams and words in the form of probability models, of course, forms the corner stone of all compression algorithms but none of these algorithms considered the distribution of the length of words directly in the models. We were motivated to consider length of words as an important factor in English text as we gathered word frequency data according to lengths for the Calgary, Canterbury [Cant00], and Gutenberg Corpus [Gute71]. A plot showing the total word frequency versus the word length results for all the text files in our test corpus (combined) is shown in Figure 1. It can be seen that most

words lie in the range of length 1 to 10. The maximum number words have length 2 to 4. The word length and word frequency results provided a basis to build context in the transformed text. We call this Length Index Preserving Transform (LIPT). LIPT can be regarded as the first step of a multi-step compression algorithm such as Bzip2 which includes run length encoding, BWT, move to front encoding, and Huffman coding. LIPT can be used as an additional component in the Bzip2 before run length encoding or simply replace it. Compared to the *-transform, we also made a couple of modifications to improve the timing performance of LIPT. For *-tranform, searching for a transformed word for a given word in the dictionary during compression and doing the reverse during decompression takes time which degrades the execution times. The situation can be improved by pre-sorting the words lexicographically and doing a binary search on the sorted dictionary both during compression and decompression stages. The other new idea that we introduce is to be able to access the words during decompression phase in a random access manner so as to obtain fast decoding. This is achieved by generating the address of the words in the dictionary by using, not numbers, but the letters of the alphabet. We need a maximum of three letters to denote an address and these letters introduce artificial but useful context for the backend algorithms to further exploit the redundancy in the intermediate transformed form of the text. LIPT encoding scheme makes use of recurrence of same length of words in the English language to create context in the transformed text that the entropy coders can exploit.

A dictionary D of words in the corpus is partitioned into disjoint dictionaries D_i, each containing words of length i, where $i = 1,2\ldots n$. Each dictionary D_i is partially sorted according to the frequency of words in the corpus. Then a mapping is used to generate the encoding for all words in each dictionary D_i. $D_i[j]$ denotes the j^{th} word in the dictionary D_i. In LIPT, the word $D_i[j]$, in the dictionary D is transfored as * $c_{len}[c][c][c]$ (the square brackets denote the optional occurrence of a letter of the alphabet enclosed and are not part of the transformed representation) in the transform dictionary D_{LIPT} where c_{len} stands for a letter in the alphabet [a-z, A-Z] each denoting a corresponding length [1-26, 27-52] and each c is in [a-z, A-Z]. If $j = 0$ then the encoding is $*c_{len}$. For $j>0$, the encoding is * c_{len} $c[c][c]$. Thus, for $1 \le j \le 52$ the encoding is $*c_{len}c$; for $53 \le j \le 2756$ it is $*c_{len}cc$, and for $2757 \le j \le 140608$ it is $*c_{len}ccc$. Thus, the 0^{th} word of length 10 in the dictionary D will be encoded as "*j" in D_{LIPT}, $D_{10}[1]$ as "*ja", $D_{10}[27]$ as "*jA", $D_{10}[53]$ as "*jaa", $D_{10}[79]$ as "*jaA", $D_{10}[105]$ as "*jba" , $D_{10}[2757]$ as "*jaaa", $D_{10}[2809]$ as "*jaba", and so on.

The transform must also be able to handle special characters, punctuation marks and capitalization. The character '*' is used to denote the beginning of an encoded word. The handling of capitalization and special characters are same as in *-encoding. Our scheme allows for a total of 140608 encodings for each word length. Since the maximum length of English words is around 22 and the maximum number of words in any D_i in our English dictionary is less than 10,000, our scheme covers all English words in our dictionary and leaves enough room for future expansion. If the word in the input text is not in the English dictionary (viz. a new word in the lexicon) it will be passed to the transformed text unaltered.

The decoding steps are as follows: The received encoded text is first decoded using the same backend compressor used at the sending end and the transformed LIPT text is recovered. The words with '*' represent transformed words and those without '*' represent non-transformed words and do not need any reverse transformation. The length character in the transformed words gives the length block and the next three characters give the offset in the respective block. The words are looked up in the original dictionary *D* in the respective length block and at the respective position in that block as given by the offset characters. The transformed words are replaced with the respective words from dictionary *D*. The capitalization mask is then applied.

4.3 Experimental Results

The performance of LIPT is measured using Bzip2 –9 [BuWh94; Chap00; Lar98; Sewa00], PPMD (order 5) [Moff90; ClTW95; Sal00] and Gzip -9 [Sal00; WiMB99] as the backend algorithms in terms of average BPC (bits per character). Note these results include some amount of pre-compression because the size of the LIPT text is smaller than the size of the original text file. By average BPC we mean the un-weighted average (simply taking the average of the BPC of all files) over the entire text corpus. The test corpus is shown in Table 1. Note that all the files given in Table 1 are text files extracted from the corpuses.

File Names	Actual Sizes
Calgary	
Bib	111261
book1	768771
book2	610856
News	377109
paper1	53161
paper2	82199
paper3	46526
paper4	13286
paper5	11954
paper6	38105
Progc	39611
Progl	71646
Progp	49379
Trans	93695

File Names	Actual Sizes
Canterbury	
alice29.txt	152089
asyoulik.txt	125179
cp.html	24603
fields.c	11150
grammar.lsp	3721
lcet10.txt	426754
plrabn12.txt	481861
xargs.1	4227
bible.txt	4047392
kjv.Gutenberg	4846137
world192.txt	2473400
Project Gutenberg	
1musk10.txt	1344739
anne11.txt	586960
world95.txt	2988578

Table 1: Text files used in our tests (a) Table showing text files and their sizes from Calgary Corpus (b) Table showing text files and their sizes from Canterbury Corpus and Project Gutenberg

We used SunOS Ultra-5 to run all our programs and to obtain results. LIPT achieves a sort of pre-compression for all the text files. We are using a 60,000 words English dictionary that takes 557,537 bytes. The LIPT dictionary takes only 330,636 bytes compared to *-encoded dictionary which takes as much storage as that of the original dictionary. Figure 2 shows the comparison of actual file sizes and file sizes obtained after applying LIPT and also after *-Encoding, for some of the text files extracted from Calgary, Canterbury, and Project Gutenberg. From Figure 2 it can be seen that LIPT

achieves a bit of compression in addition to preprocessing the text before application to any compressor.

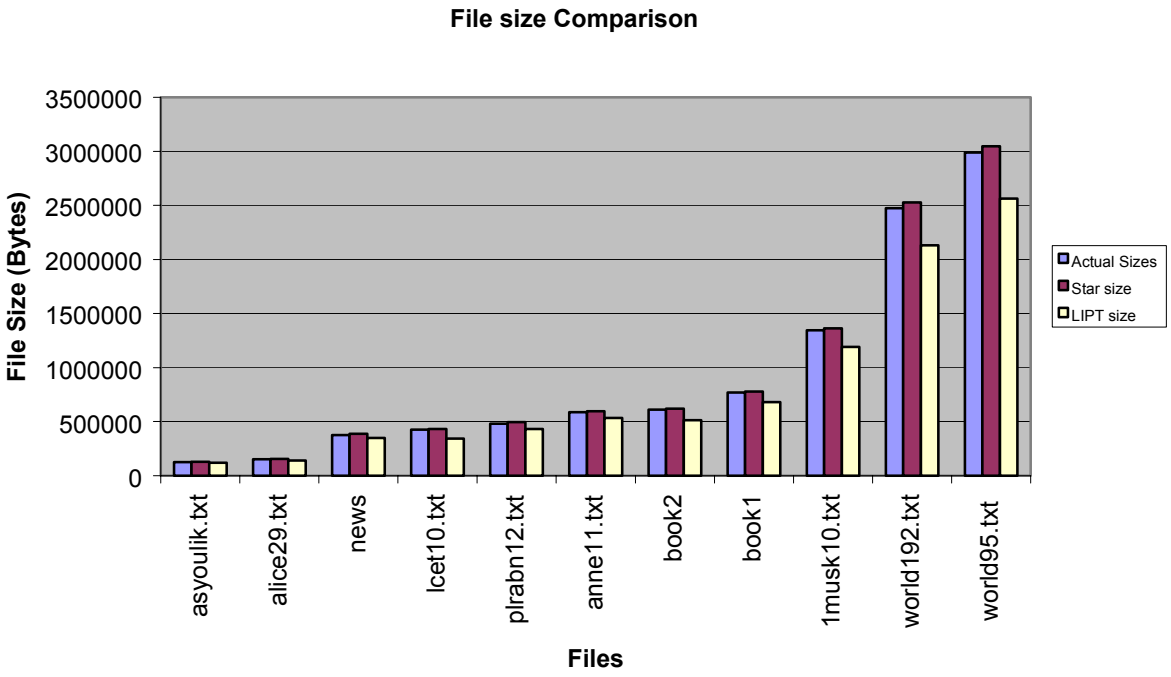


Figure 2: Chart showing the comparison between actual file sizes, Star-Encoding, and the LIPT file sizes for a few of the text files extracted from our test corpus

The results can be summarized as follows: The average BPC using original Bzip2 is 2.28, and using Bzip2 with LIPT gives average BPC of 2.16, a 5.24% improvement (Table 2). The average BPC using original PPMD (order 5) is 2.14, and using PPMD with LIPT gives average BPC of 2.04, and overall improvement of 4.46% (Table 3). The average BPC using original Gzip-9 is 2.71, and using Gzip-9 with LIPT the average BPC is 2.52, a 6.78% improvement (Table 4). The files in Tables 2,3 and 4 are listed in ascending order of file size. Note that for normal text files, the BPC decreases as the file size increases. This can clearly be seen from the Tables especially part (c) of every table that has three text files from Project Gutenberg.

(a)

FileNames	Bzip2 (BPC)	Bzip2 with LIPT (BPC)
Calgary		
paper5	3.24	**2.95**
paper4	3.12	**2.74**
paper6	2.58	**2.40**
Progc	2.53	**2.44**
paper3	2.72	**2.45**
Progp	1.74	**1.72**
paper1	2.49	**2.33**
Progl	1.74	**1.66**
paper2	2.44	**2.26**
Trans	1.53	**1.47**
Bib	1.97	**1.93**
News	2.52	**2.45**
Book2	2.06	**1.99**
Book1	2.42	**2.31**
Average BPC	2.36	**2.22**

(b)

FileNames	Bzip2 (BPC)	Bzip2 with LIPT (BPC)
Canterbury		
grammar.lsp	2.76	**2.58**
xargs.1	3.33	**3.10**
fields.c	2.18	**2.14**
cp.html	2.48	**2.44**
asyoulik.txt	2.53	**2.42**
alice29.txt	2.27	**2.13**
lcet10.txt	2.02	**1.91**
plrabn12.txt	2.42	**2.33**
world192.txt	1.58	**1.52**
bible.txt	1.67	**1.62**
kjv.gutenberg	1.66	**1.62**
Average BPC	2.26	**2.17**

(c)

FileNames	Bzip2 (BPC)	Bzip2 with LIPT (BPC)
Gutenberg		
Anne11.txt	2.22	**2.12**
1musk10.txt	2.08	**1.98**
World95.txt	1.54	**1.49**
Average BPC	1.95	**1.86**

Table 2: Tables a – c show BPC comparison between original Bzip2 –9, and Bzip2 –9 with LIPT for the files in three corpuses

(a)

FileNames	PPMD (BPC)	PPMD with LIPT
Calgary		
paper5	2.98	**2.74**
paper4	2.89	**2.57**
paper6	2.41	**2.29**
Progc	2.36	**2.30**
paper3	2.58	**2.37**
Progp	1.70	**1.68**
paper1	2.33	**2.21**
Progl	1.68	**1.61**
paper2	2.32	**2.17**
Trans	1.47	**1.41**
Bib	1.86	**1.83**
news	2.35	**2.31**
book2	1.96	**1.91**
book1	2.30	**2.23**
Average BPC	2.23	**2.12**

(b)

FileNames	PPMD (BPC)	PPMD with LIPT (BPC)
Canterbury		
grammar.lsp	2.36	**2.21**
xargs.1	2.94	**2.73**
fields.c	2.04	**1.97**
cp.html	2.26	**2.22**
asyoulik.txt	2.47	**2.35**
alice29.txt	2.18	**2.06**
lcet10.txt	1.93	**1.86**
plrabn12.txt	2.32	**2.27**
world192.txt	1.49	**1.45**
bible.txt	1.60	**1.57**
kjv.gutenberg	1.57	**1.55**
Average BPC	2.11	**2.02**

(c)

FileNames	PPMD (BPC)	PPMD with LIPT
Gutenberg		
Anne11.txt	2.13	**2.04**
1musk10.txt	1.91	**1.85**
World95.txt	1.48	**1.45**
Average BPC	1.84	**1.78**

Table 3: Tables a – c show BPC comparison between original PPMD (order 5), and PPMD (order 5) with LIPT for the files in three corpuses.

(a)

FileNames	Gzip (BPC)	Gzip with LIPT (BPC)
Calgary		
paper5	3.34	**3.05**
paper4	3.33	**2.95**
paper6	2.77	**2.61**
Progc	2.68	**2.61**
paper3	3.11	**2.76**
Progp	1.81	**1.81**
paper1	2.79	**2.57**
Progl	1.80	**1.74**
paper2	2.89	**2.62**
Trans	1.61	**1.56**
bib	2.51	**2.41**
news	3.06	**2.93**
book2	2.70	**2.48**
book1	3.25	**2.96**
Average BPC	2.69	**2.51**

(b)

FileNames	Gzip (BPC)	Gzip with LIPT (BPC)
Canterbury		
grammar.lsp	2.68	**2.61**
xargs.1	3.32	**3.13**
fields.c	2.25	**2.21**
Cp.html	2.60	**2.55**
asyoulik.txt	3.12	**2.89**
alice29.txt	2.85	**2.60**
lcet10.txt	2.71	**2.42**
plrabn12.txt	3.23	**2.96**
world192.txt	2.33	**2.18**
bible.txt	2.33	**2.18**
kjv.gutenberg	2.34	**2.19**
Average BPC	2.70	**2.54**

(c)

FileNames	Gzip (BPC)	Gzip with LIPT (BPC)
Gutenberg		
Anne11.txt	3.02	**2.75**
1musk10.txt	2.91	**2.62**
World95.txt	2.31	**2.15**
Average BPC	2.75	**2.51**

Table 4: Tables a – c show BPC comparison between original Gzip -9, and Gzip –9 with LIPT for the files in three corpuses.

Figure 3 gives a bar chart comparing the BPC of the original Bzip2,PPMD and the compressors in conjunction with LIPT for a few text files extracted from our test corpus. From Figure 3 it can be seen that Bzip2 with LIPT (second bar in Figure 3) is close to the original PPMD (third bar in Figure 3) in bits per character (BPC).

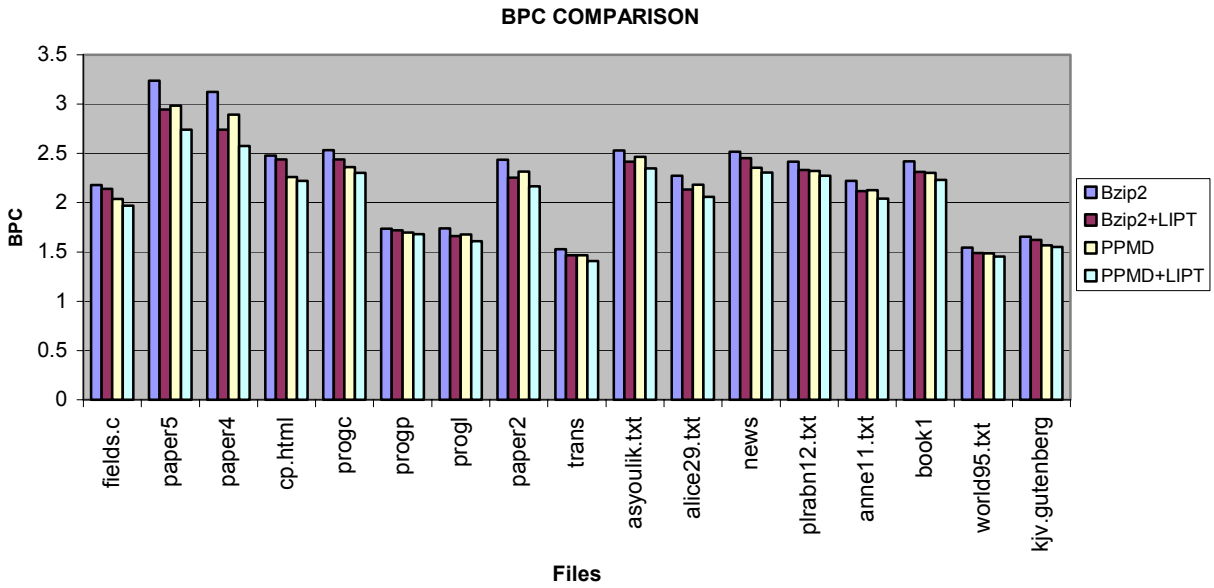


Figure 3: Bar Chart giving comparison of Bzip2, Bzip2 with LIPT, PPMD, and PPPMD with LIPT

From Figure 3 it can also be seen that in instances like paper5, paper4, progl, paper2, asyoulik.txt, and alice29.txt, Bzip2 with LIPT is beating the original PPMD in terms of BPC. The difference between average BPC for Bzip2 with LIPT (2.16) and original PPMD (2.1384) is only around 0.02 bits i.e. average BPC for Bzip2 with LIPT is only around 1% more than the original PPMD. This observation is important as it contributes towards the efforts being made by different researchers to obtain PPMD BPC performance with a faster compressor. It is shown later on timing results that Bzip2 with LIPT is much faster than the original PPMD. (Note that although Bzip2 with LIPT gives lower BPC than the original Bzip2, the former is much slower than the later as discussed in later in this report). Table 5 gives a summary comparison of BPC for the original Bzip2 –9, PPMD (order 5), Gzip –9, Huffman (character based), word-based Arithmetic coding, and these compressors with Star-Encoding and LIPT. The data in Table 5 shows that LIPT performs much better over Star-Encoding and original algorithms except for character based Huffman and Gzip –9.

	Original (BPC)	*-encoded (BPC)	LIPT (BPC)
Huffman (character based)	4.87	**4.12**	4.49
Arithmetic (word based)	2.71	2.90	**2.61**
Gzip-9	2.70	**2.52**	**2.52**
Bzip2	2.28	2.24	**2.16**
PPMD	2.13	2.13	**2.04**

Table 5: Summary of BPC Results

Table 5 also shows that Star-encoding gives a better average BPC performance for character-based Huffman, Gzip, and Bzip2 but gives worse average BPC performance for word-based arithmetic coding and PPMD. This is due to the presence of the non-English words and special symbols in the text. For a pure text file, for example the dictionary itself, the star dictionary has a BPC of 1.88 and original BPC is 2.63 for PPMD. The improvement is 28.5% in this case. Although the average BPC for Star-encoding is worse than original PPMD, there are 16 files that show improved BPC, and 12 files show worse BPC. The number of words in the input text that are also found in English dictionary *D* is an important factor for the final compression ratio. For character based Huffman, Star-encoding performs better than the original Huffman and LIPT with Huffman. This is because in Star-encoding there are repeated occurrences of the character '*' which gets the highest frequency in the Huffman code book and is thus encoded with lowest number of bits resulting in better compression results than the original and the LIPT files.

4.4 Comparison with Recent Improvements of BWT and PPM

We focus our attention on LIPT over Bzip2 (which uses BWT), Gzip and PPM algorithms because Bzip2 and PPM outperform other compression methods and Gzip is commercially available and commonly used. Of these, BWT based approach has proved to be the most efficient and a number of efforts have been made to improve its efficiency. The latest efforts include Balkenhol, Kurtz, and Shtarkov [BaKS99], Seward [Sewa00], Chapin [Chap00], and Arnavut [Arna00]. PPM on the other hand gives better compression ratio than BWT but is very slow in execution time. A number of efforts have

been made to reduce the time for PPM and also to improve the compression ratio. Sadakane, Okazaki, and Imai [SaOH00] have given a method where they have combined PPM and CTW [WiST95] to get better compression. Effros [Effo00] has given a new implementation of PPM* with the complexity of BWT. Tables 6 and 7 give a comparison of compression performance of our proposed transform which shows that LIPT has better BPC for most of the files and it has better average BPC than all the other methods cited. Some data in Table 6 and Table 7 have been taken from the references given in the respective columns.

File	MBSWIC [Arna00]	BKS98 [BaKS99]	Best x of $2x - 1$ [Chap00]	Bzip2 with LIPT
Bib	2.05	1.94	1.94	**1.93**
book1	**2.29**	2.33	**2.29**	2.31
book2	2.02	2.00	2.00	**1.99**
news	2.55	2.47	2.48	**2.45**
paper1	2.59	2.44	2.45	**2.33**
paper2	2.49	2.39	2.39	**2.26**
progc	2.68	2.47	2.51	**2.44**
progl	1.86	1.70	1.71	**1.66**
progp	1.85	**1.69**	1.71	1.72
trans	1.63	**1.47**	1.48	**1.47**
Average BPC	2.21	2.105	2.11	**2.07**

Table 6: BPC comparison of approaches based on BWT

File	Multi-alphabet CTW order 16 [SaOH00]	NEW Effros [Effo00]	PPMD (order 5) with LIPT
bib	1.86	1.84	**1.83**
book1	**2.22**	2.39	2.23
book2	1.92	1.97	**1.91**
News	2.36	2.37	**2.31**
paper1	2.33	2.32	**2.21**
paper2	2.27	2.33	**2.17**
Progc	2.38	2.34	**2.30**
Progl	1.66	**1.59**	1.61
Progp	1.64	**1.56**	1.68
Trans	1.43	**1.38**	1.41
Average BPC	2.021	2.026	**1.98**

Table 7: BPC comparison of new approaches based on Prediction Models

4.5 Comparison with Word-based Huffman

Huffman compression method also needs sharing of the same static dictionary at both the sender and receiver end, as does our method. Canonical Huffman [WiMB99] method assigns variable length addresses to words using bits and LIPT assigns variable length

offset in each length block using letters of alphabet. Due to these similarities we compare the word-based Huffman with LIPT (we used Bzip2 as the compressor). Huffman and LIPT both sort the dictionary according to frequency of use of words. Canonical Huffman assigns a variable address to the input word, building a tree of locations of words in the dictionary and assigning 0 or 1 to each branch of the path. LIPT exploits the structural information of the input text by including the length of the word in encoding. LIPT also achieves a pre-compression due to the variable offset scheme. In Huffman, if new words are added, the whole frequency distribution has to be recomputed as well as the Huffman codes for them.

A typical word-based Huffman model is a zero-order word-based semi-static model [WiMB99]. Text is parsed at the first pass of scan to extract zero-order words and non-words as well as their frequency distributions. Words are typically defined as consecutive characters and non-words are typically defined as punctuation, space and control characters. If an unseen word or non-word occurred, normally some escape symbol is transmitted, and then the string is transmitted as sequence of single characters. Some special type of strings can be considered for special representation, for example, the numerical sequences. To handle arbitrarily large sequence of numbers, one way of encoding is to break them in to smaller pieces e.g. groups of four digits. Word-based models can generate a large number of symbols. For example, in our text corpus with the size of 12918882 bytes, there are 70661 words and 5504 non-words. We can not make sure that these may include all or most of the possible words in a huge database since the various words may be generated by the definition of words here. Canonical Huffman code [Sewa00] is selected to encode the words. The main reason for using canonical Huffman code is to provide efficient data structures to deal with huge dictionary generated and for fast decompression so that the retrieval is made faster.

Table 8 shows the BPC comparison. For LIPT, we extract the strings of characters in the text and build the LIPT dictionary for each file. In contrast to the approach given in [WiMB99], we do not include the words composed of digits and mixture of alphabets and digits as well as other special characters. We try to make a fair comparison, however, word-based Huffman still uses a broader definition of “words”. Comparing the average BPC, the Managing Gigabyte [WiMB99] word-based Huffman model has a 2.506 BPC for our test corpus. LIPT with Bzip2 has a BPC of 2.17. The gain is 13.44%. LIPT does not give improvement over word based Huffman for files with mixed text such as source files for programming languages. For files with more English word, LIPT shows consistent gain.

We also have compared LIPT with some of the newer compression reported in the literature and websites such as YBS [YBS00],RK [RK00-1,RK00-2] and PPMonstr [PPMDH00]. The average BPC using LIPT along with these methods is around 2.00BPC which is better than any of these algorithms as well as the original Bzip2 and PPMD. For details, the reader is referred to [AwMu01;Awan01].

File	Filesize	Bzip2 with LIPT	Word-based Huffman	% GAIN
cp.html	21333	2.03	2.497	18.696
paper6	32528	2.506	2.740	8.550
progc	33736	2.508	2.567	2.289
paper1	42199	2.554	2.750	7.134
progp	44862	1.733	2.265	23.497
progl	62367	1.729	2.264	23.618
trans	90985	1.368	1.802	24.090
bib	105945	1.642	2.264	27.477
asyoulik.txt	108140	2.525	2.564	1.527
alice29.txt	132534	2.305	2.333	1.194
lcet10.txt	307026	2.466	2.499	1.314
news	324476	2.582	2.966	12.936
book2	479612	2.415	2.840	14.963
anne11.txt	503408	2.377	2.466	3.597
1musk10.txt	1101083	2.338	2.454	4.726
world192.txt	1884748	1.78	2.600	31.551
world95.txt	2054715	1.83	2.805	34.750
kjv.gutenberg	4116876	1.845	2.275	18.914
AVERAGE		**2.169**	**2.506**	**13.439**

Table 8: BPC comparison for test files using Bzip2 with LIPT, and word-based Huffman

4.6 Timing Performance Measurements

The improved compression performance of our proposed transform comes with a penalty of degraded timing performance. For off-line and archival storage applications, such penalties in timing are quite acceptable if a substantial savings in storage space can be achieved. The increased compression/decompression times are due to frequent access to a dictionary and its transform dictionary. To alleviate the situation, we have developed efficient data structures to expedite access to the dictionaries and memory management techniques using caching. Realizing that certain on-line algorithms might prefer not to use a pre-assigned dictionary, we also have been working on a new family of algorithms, called M5zip to obtain the transforms dynamically with no dictionary and with small dictionaries (7947 words and 10000 words), which will be reported in future papers.

The experiments were carried out on 360MHz Ultra Sparc-IIi Sun Microsystems machine housing SunOS 5.7 Generic_106541-04. In our experiments we compare compression times of Bzip2, Gzip and PPMD against Bzip2 with LIPT, Gzip with LIPT and PPMD with LIPT. During the experiments we have used –9 option for Gzip. This option supports for better compression. Average compression time, for our test corpus, using LIPT with Bzip2 -9, Gzip -9, and PPMD is 1.79 times slower, 3.23 times slower and fractionally (1.01 times) faster compared to original Bzip2, Gzip and PPMD respectively. The corresponding results for decompression times are 2.31 times slower, 6.56 times slower and almost the same compared to original Bzip2, Gzip and PPMD respectively. Compression using Bzip2 with LIPT is 1.92 times faster and decompression is 1.98 times

faster than original PPMD (order 5). The increase in time over standard methods is due to time spent in preprocessing the input file. Gzip uses –9 option to achieve maximum compression therefore we find that the times for compression using Bzip2 are less than Gzip. When maximum compression option is not used, Gzip runs much faster than Bzip2.

Decompression time for methods using LIPT includes decompression using compression techniques plus reverse transformation time. Bzip2 with LIPT decompresses 2.31 times slower than original Bzip2, Gzip with LIPT decompresses 6.56 times slower than Gzip, and PPMD with LIPT is almost the same.

5. Dictionary Organization and Memory Overhead

LIPT uses a static English language dictionary of 59951 words having a size of around 0.5 MB. LIPT uses transform dictionary of around 0.3 MB. . The transformation process requires two files namely English dictionary, which consist of most frequently used words, and a transform dictionary, which contains corresponding transforms for the words in English dictionary. There is one-to-one mapping of word from English to transform dictionary. The words not found in the dictionary are passed as they are. To generate the LIPT dictionary (which is done offline), we need the source English dictionary to be sorted on blocks of lengths and words in each block should be sorted according to frequency of their use. On the other hand we need a different organization of dictionary for encoding and decoding procedures (which are done online) in order to achieve efficient timing. We use binary search which on average needs log$w$ comparisons where w is the number of words in the English dictionary D. To use binary search, we need to sort the dictionary lexicographically. We sort the blocks once on loading the dictionary into memory using Quicksort. For successive searching the access time is $M \times \log w$, where M is number of words in the input file and w is number of words in dictionary. The total number of comparison is $w \times \log w + M \times \log w$. In secondary storage, our dictionary structure is based on first level blocking according length and then within each block we sort the words according to their frequency of use. In memory, we organize the dictionary into two levels. In Level 1, we classify the words in dictionary based on the length of the word and sort these blocks in ascending order of frequency of use. Then in level 2, we sort the words within each block of length lexicographically. This sorting is done once upon loading of dictionaries into the memory. It is subjected to resorting only when there is modification to the dictionary like adding or deleting words from dictionary. In order to search a word of length l and starting character as z, the search domain is only confined to a small block of words which have length l and start with z. It is necessary to maintain a version system for different versions of the English dictionaries being used. A simple method works well with our existing dictionary system. When new words are added they are added at the end of the respective word length blocks. Adding words at the end has two advantages: previous dictionary word-transform mapping is preserved scalability without distortion is maintained in the dictionary.

It is important to note that the dictionary is installed with the executable and is not transmitted every time with the encoded files. The only other time it is transmitted is when there is an update or new version release. LIPT encoding needs to load original

English dictionary (currently 55K bytes) and LIPT dictionary D_{LIPT} (currently 33K). There is an additional overhead of 1.5 K for the two level index tables we are using in our dictionary organization in memory. So currently, in total, LIPT incurs about 89K bytes. Bzip2 is claimed to use 400K+(7 × Block size) for compression [Bzip2]. We use "–9 option" for Bzip2 and Bzip2 –9 uses 900K of block size for the test. Therefore, in total we need about 6700K for Bzip2. For decompression, it takes around 4600K and 2305K with –s option [Bzip2]. For PPMD it takes as about 5100K + file size (this is the size we fixed in the source code for PPMD). So LIPT takes only a small additrional overhead compared to Bzip2 and PPM in memory usage.

6. Three New Transforms – ILPT, NIT and LIT

We will briefly describe our three new lossless reversible text transforms based on LIPT. We give experimental results for the new transforms and discuss them briefly. For details on these transformations and experimental results the reader is referred to [Awan01]. Note that there is no significant effect on the time performance as the dictionary loading method remains the same and the number of words also remain the same in the static English dictionary *D* and transform dictionaries. Hence we will only give the BPC results obtained with different approaches for the corpus.

Initial Letter Preserving transform (ILPT) is similar to LIPT except that we sort the dictionary into blocks based on the lexicographic order of starting letter of the words. We sort the words in each letter block according to descending order of frequency of use. The character denoting length in LIPT (character after '*') is replaced by the starting letter of the input word i.e. instead of $*c_{len}[c][c][c]$, for ILPT it becomes $*c_{init}[c][c][c]$ where c_{init} denotes the initial (staring) letter of the word. Everything else is handled the same way as in LIPT. Bzip2 with ILPT has an average BPC of 2.12 for all the files in all the corpuses combined. This means that Bzip2 –9 with ILPT shows 6.83% improvement over the original Bzip2 –9. Bzip2 –9 with ILPT shows 1.68 % improvement over Bzip2 with LIPT.

Number Index Transform (NIT) scheme uses variable addresses based on letters of alphabet instead of numbers. We wanted to compare this using a simple linear addressing scheme with numbers i.e. giving addresses 0 - 59950 to 59951 words in our dictionary. Using this scheme on our English dictionary *D*, sorted according to length of words and then sorted according to frequency within each length block, gave deteriorated performance compared to LIPT. So we sorted the dictionary globally according to descending order of word usage frequency. No blocking was used in the new frequency sorted dictionary. The transformed words are still denoted by staring character '*'. The first word in the dictionary is encoded as "*0", the 1000th word is encoded as "*999", and so on. Special character handling is same as in LIPT. We compare the BPC results for Bzip2 –9 with the new transform NIT with Bzip2 –9 with LIPT. Bzip2 with NIT has an average BPC of 2.12 for all the files in all the corpuses combined. This means that Bzip2 –9 with NIT shows 6.83% improvement over the original Bzip2 –9. Bzip2 –9 with NIT shows 1.68 % improvement over Bzip2 with LIPT.

Combining the approach taken in NIT and the using letters to denote offset , we arrive at another transform which is same as NIT except that now we use letters of the alphabet [a-zA-Z] to denote the index or the linear address of the words in the dictionary, instead of numbers. This scheme is called *Letter Index Transform (LIT)*. Bzip2 with LIT has an average BPC of 2.11 for all the files in all corpuses combined. This means that Bzip2 –9 with LIT shows 7.47% improvement over the original Bzip2 –9. Bzip2 –9 with LIT shows 2.36 % improvement over Bzip2 with LIPT.

The transform dictionary sizes vary with the transform. The original dictionary takes 557537 bytes. The transformed dictionaries for LIPT, ILPT, NIT and LIT are 330636, 311927, 408547 and 296947 bytes, respectively. Note that LIT has the smallest size dictionary and shows uniform compression improvement over other transforms for almost all the files.

Because of its better performance compared to the performance of other transforms, we have compared LIT with PPMD, YBS, RK and PPMonstr. PPMD (order 5) with LIT has an average BPC of 1.99 for all the files in all corpuses combined. This means that PPMD (order 5) with LIT shows 6.88% improvement over the original PPMD (order 5). PPMD with LIT shows 2.53 % improvement over PPMD with LIPT. RK with LIT has average BPC lower than RK with LIPT. LIT performs well with YBS, RK, and PPMonstr giving 7.47%, 5.84%, and 7.0% improvement over the original methods, respectively. It is also important to note that LIT with YBS outperforms Bzip2 by 10.7% and LIT with PPPMonstr outperforms PPMD by 10%.

7. Theoretical Explanation

In this section we give theoretical explanation for the compression performance based on entropy calculations. We give mathematical equations to giving factors that affects compression performance. We provide experimental results in support of our proposed theoretical explanation..

7.1 Qualitative Explanation for Better Compression with LIPT

LIPT introduces frequent occurrences of common characters for BWT and good context for PPM as well as it compresses the original text. Cleary, Teahan, and Witten [CLTW95], and Larsson [Lar98] have discussed the similarity between PPM and Bzip2. PPM uses a probabilistic model based on the context depth and uses the context information explicitly. On the other hand the frequency of similar patterns and local context affect the performance of BWT implicitly. Fenwick [Fenw96] also explains how BWT exploits the structure in the input text. LIPT introduces added structure along with smaller file size leading to better compression after applying Bzip2 or PPMD.
The offset characters in LIPT represent a variable-length encoding similar to Huffman encoding and produce some initial compression of the text but the difference from Huffman encoding is significant. The address of the word in the dictionary is generated at the modeling rather than entropy encoding level and LIPT exploits the distribution of words in English language based on the length of the words as given in Figure 1. The

sequence of letters to denote the address also has some inherent context depending on how many words are in a single group, which also opens another opportunity to be exploited by the backend algorithm at the entropy level.

There are repeated occurrences of words with same length in a usual text file. This factor contributes in introducing good and frequent context and thus higher probability of occurrence of same characters (space, '*', and characters denoting length of words) that enhance the performance of Bzip2 (which uses BWT) and PPM as proved by results given earlier in the report. LIPT generates encoded file, which is smaller in size than the original text file. Because of the small input file along with a set of artificial but well defined deterministic context, both BWT and PPM can exploit the context information very effectively producing a compressed file that is smaller than the file without using LIPT. In order to verify our conjecture that LIPT may produce effective context information based on the frequent word length recurrence in the text, we made some measurement on order statistics of the file entropy with and without using LIPT and calculated the effective BPC over context length up to 5. The results are given in Table 9 with respect to a typical file alice29.txt for the purpose of illustration.

Context Length (Order)	Original (count)	Original BPC-Entropy	LIPT (count)	LIPT BPC-Entropy
5	114279	1.98	108186	2.09
4	18820	2.50	15266	3.12
3	12306	2.85	7550	**2.03**
2	5395	3.36	6762	**2.18**
1	1213	4.10	2489	**3.50**
0	75	6.20	79	6.35
Total	152088		**140332**	

Table 9: Comparison of bits per character (BPC) in each context length used by PPMD, and PPMD with LIPT for the file alice29.txt

The data in Table 9 shows that LIPT uses less number of bits for context order 3, 2, and 1 as compared to the original file. The reason for this can be derived from our discussion on frequency of recurrence of length throughout the input text. For instance in alice29.txt there are 6184 words of length 3 and 5357 words of length 4 (there are words of other lengths as well but here we are taking lengths 3 and 4 to illustrate our point). Out of these, 5445 words of length 3 and 4066 words of length 4 are found in our English dictionary. This means that the sequence, space followed by '*c', will be found 5445 times and the sequence, space followed by '*d', will be found 4066 times in the transformed text for alice29.txt using LIPT. There can be other sequences of offset letters after 'c' or 'd' but at least these three characters (including space) will be found in this sequences this many times. Here these three characters define a context of length three. Similarly there will be other repeated lengths in the text. The sequence space followed by

'*' with two characters defines context of length two that is found in the transformed text very frequently. We can see from Table 9 that the BPC for context length 2 and 3 is much lower than the rest. Apart from the space, '*', and the length character sequence the offset letters also provide added probability for finding similar contexts. LIPT also achieves a pre-compression at the transformation stage (before actual compression). This is because transformation for a word can use at most 6 characters and hence words of length 7 and above are encoded with fewer characters. Also the words with other lengths can be encoded with fewer characters depending on the frequency of their usage (their offset will have fewer characters in this case). Due to sorting of dictionary according to frequency and length blocking we have been able to achieve reduction in size of the original files in the range of 7% to 20%.

7.2 Entropy Based Explanation for Better Compression Performance of Our Transforms

Let n denote the total number of unique symbols (characters) in a message Q and P_i denote the probability of occurrence of each symbol i. Then entropy S of message Q [ShW98] is given by:

$$S = -\sum_{1}^{n} P_i \log_2 P_i \tag{2}$$

Entropy is highest when all the events are equally likely that is when all P_i are $1/n$. The difference between the entropy at this peak (maximum entropy) where probability P is $1/n$ and the actual entropy given by equation (2) is called the redundancy R of the source data.

$$R = -\log_2(1/P) - (-\sum_{1}^{n} P_i \log_2 P_i) = \log_2 n + \sum_{1}^{n} P_i \log_2 P_i \tag{3}$$

This implies that when entropy S is equal to maximum entropy $S_{max} = \log_2 n$ then there is no redundancy and the source text cannot be compressed further. The performance of compression method is measured by the average number of bits it takes to encode a message. Compression performance can also be measured by *compression factor*, which is the ratio of the number of characters in the source message to the number of characters in the compressed message.

7.3 Explanation Based on First Order (Context Level Zero) Entropy

First order entropy is sum of entropies for each symbol in a text. It is based on context level zero i.e. no prediction based on preceding symbols. Compressed file size can be given in terms of redundancy and original file size. We can rewrite Equation (3) as:

$$R = S_{\max} - S \tag{4}$$

R is the total redundancy found in the file. Now compressed file size F_c is given as:

$$F_c = F - \frac{R \times F}{S_{\max}} \tag{5}$$

where F is the size of uncompressed file. Notice that $\frac{R}{S_{\max}}$ gives the per symbol redundancy for the file.

Substituting value of R from Equation (3) into Equation (5) we get:

$$F_c = F \times \frac{S}{S_{\max}} \tag{6}$$

The quantity $S/S_{\max}$ is called the relative entropy and gives a measure of fraction of file that can be compressed. As we are using ASCII character set as our alphabet so S_{max} is 8 for our discussion. Hence equation (6) becomes:

$$F_c = F \times \frac{S}{8} \tag{7}$$

Compression Factor C is given by

$$C = \frac{F}{F_c} \tag{8}$$

Substituting value of F_c from Equation (6) into Equation (8) we get:

$$C = \frac{S_{\max}}{S} \tag{9}$$

This relationship is also verified in Table 10 in compression factor column. Table 10 gives the first-order entropy (Equation (2)) and redundancy (Equation (3)) data for Calgary Corpus. The maximum entropy is 8 bits/symbol as we are using 256 character set as our alphabet. Table 10 also shows theoretical compression factors for respective files based on first order entropy S and compression factors based on experimental results using first-order character based Huffman coder (context level zero). The last two columns of Table 10 show that the theoretical and experimental compression factors are very close and the compression factor is inversely proportional to redundancy based on first order entropy. Higher compression factor means more compression as compression factor is given by the ratio of uncompressed file size to compressed file size. Redundancy shows how much a file can be compressed. Lower entropy means less number of bits to encode the file resulting in better compression. Table 10 verifies all these observations.

File	first order entropy *S* (bits/character)	Redundancy *R* (bits/character)	Compression factor (based on first order entropy *S*)	Compression factor (based on experimental results using Huffman Coding)
Trans	5.53	2.47	1.45	1.44
Progc	5.20	2.80	1.54	1.53
Bib	5.20	2.80	1.54	1.53
News	5.19	2.81	1.54	1.53
Paper6	5.01	2.99	1.60	1.59
Paper1	4.98	3.02	1.61	1.59
Paper5	4.94	3.06	1.62	1.61
Progp	4.87	3.13	1.64	1.63
book2	4.79	3.21	1.67	1.66
Progl	4.77	3.23	1.68	1.67
Paper4	4.70	3.30	1.70	1.69
Paper3	4.67	3.33	1.71	1.71
Paper2	4.60	3.40	1.74	1.73
book1	4.53	3.47	1.77	1.75

Table 10: Entropy and redundancy in relation with compression factor (Calgary Corpus- original files)

Now let F_t represent the size of the file transformed with any one of our transforms, which has the size F before application of the transform. Remember that our transforms produce a reduced file size for the intermediate text by a factor s, $0<s<1$ (for *-transform for which s is slightly greater than 1 implying an expansion). Hence we can write:

$$F_t = F \times s \tag{10}$$

The compressed file size F_c is given by

$$F_c = \frac{F_t \times S_t}{S_{\max}} \tag{11}$$

where S_t is entropy of the transformed file. Intermediate compression factor C_{int} is given by

$$C_{\text{int}} = \frac{F_t}{F_c} \tag{12}$$

Substituting the value of F_c in Equation (8):

$$C = \frac{F}{F_t \times S_t \Big/ S_{\max}} \tag{13}$$

Finally, substituting the value of F_t, we get

$$C = \frac{S_{\max}}{s \times S_t} \tag{14}$$

Equation (14) shows that the compression given by *compression factor C is inversely proportional to the product of file size reduction factor s achieved by a transform, and entropy* S_t. Smaller s and proportionally smaller entropy of the transformed file means higher compression. Equation (14) is same as Equation (9) which is for uncompressed (original) files with s= 1 and $S_t = S$.

The results for the products $s \times S$ for original files and $s \times S_t$ of all our transforms for Calgary corpus are given in Table 11.

File	Original $s \times S$	C	LIPT $s \times S_t$	C	ILPT $s \times S_t$	C	NIT $s \times S_t$	C	LIT $s \times S_t$	C
Trans	5.53	1.44	5.10	1.56	4.39	1.62	4.61	1.61	**4.38**	**1.64**
Bib	5.20	1.53	4.83	1.65	4.63	1.72	4.74	1.68	**4.51**	**1.77**
Progc	5.20	1.53	5.12	1.55	2.20	1.61	2.36	1.52	**2.13**	**1.62**
News	5.19	1.53	4.96	1.60	4.71	1.69	4.93	1.61	**4.60**	**1.73**
Paper6	5.01	1.59	4.58	1.73	4.28	1.85	4.22	1.88	**4.11**	**1.93**
Paper1	4.98	1.59	4.42	1.80	4.11	1.93	4.13	1.92	**3.99**	**1.99**
Paper5	4.94	1.61	4.41	1.80	4.14	1.92	4.24	1.87	**3.98**	**1.99**
Progp	4.87	1.63	4.93	1.61	4.67	1.65	4.94	1.53	**4.64**	**1.66**
book2	4.79	1.66	4.20	1.89	3.86	2.05	3.91	2.03	**3.71**	**2.14**
Progl	4.77	1.67	4.61	1.72	4.71	1.82	5.09	1.67	**4.69**	**1.83**
Paper4	4.70	1.69	4.00	1.98	3.67	2.16	3.70	2.13	**3.52**	**2.25**
Paper3	4.67	1.71	3.98	1.99	3.62	2.19	3.74	2.12	**3.50**	**2.26**
Paper2	4.60	1.73	4.00	1.98	3.63	2.18	3.70	2.14	**3.48**	**2.28**
book1	4.53	1.75	4.05	1.95	3.66	2.17	3.72	2.13	**3.50**	**2.27**

Table 11: Product $s \times S$ for original and $s \times S_T$ for all transformed files in Calgary Corpus

We also give experimental compression factor C obtained using Huffman Coding (character based). From Table 11 we can see that LIT has the lowest $s \times S_t$ for most of the files and thus will give the best compression performance. This is also depicted in Figure 4 which shows that compression is inversely proportional to product of file size factor s and entropy of the file S_t (Only the graph for LPIT is shown).

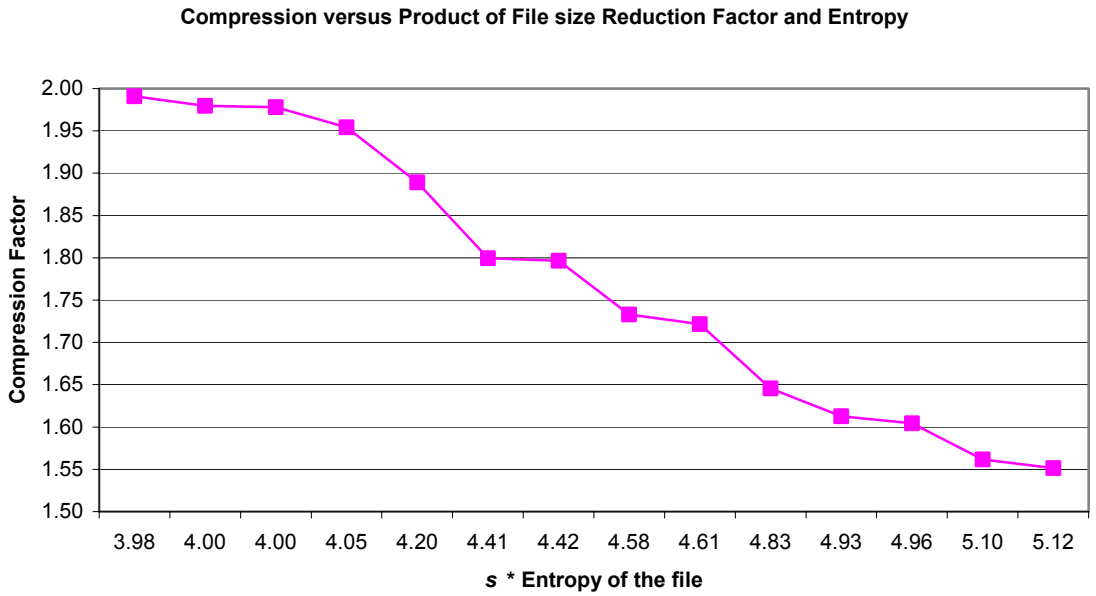


Figure 4: Graph showing Compression versus Product of File size reduction factor s and File Entropy S_t

7.4 Explanation based on higher order context-based entropy

A context-based text compression method uses the context of a symbol to predict it. This context has certain length k called the context order and the method is said to be following order-k Markov Model [ShW98]. PPMD (order 5) used in experiments in this report uses order-5 Markov Model. Burrows Wheeler method is also a context-based method [Fenw96; Lar98] using Markov Model. Burrows Wheeler method is a context-based compression method but performs a transformation on the input data and then applies a statistical model.

If we define each context level c_i as a finite state of Markov model then the average entropy S_c of a context-based method is given by:

$$S_c = \sum_{i=0}^{k} P(c_i) S(c_i) \qquad (15)$$

where i is the context level, k is the length of maximum context (context order), $P(c_i)$ is the probability of the model to be in context level c_i , and $S(c_i)$ is the entropy of context level c_i . In practice, this average entropy S_c can also be calculated by totaling the bits taken in each context level and dividing it by the total count of characters encoded at that level. Now based on Equation (15), let us give the entropies for a few files in Calgary Corpus. Table 12 gives the comparison of the product of file size reduction s and average context-based entropy S_c for Bzip2, Bzip2 compression factors C_{bzip2} and PPMD compression factors C_{PPMD} for original, LIPT, and LIT files. The maximum context length is 5 (order 5). These calculations are derived from similar information as given in Table 11 for respective files. The files in Table 12 are listed in ascending order of

compression factors. From the data given in Table 12 it can be seen that compression factor for bzip2, C_{bzip2} and also for PPMD, C_{PPMD} are inversely proportional to $s \times S_c$ .One also notes that LIT has the lowest $s \times S_c$ but highest compression factors. PPMD with LIT has the highest compression factors. Hence lower $s \times S_c$ justifies the better compression performance for LIT. The same argument can be applied to other transforms.

Files	C_{bzip2} (original)	C_{PPMD} (original)	original $s \times S_c$	C_{bzip2} (LIPT)	C_{PPMD} (LIPT)	LIPT $s \times S_c$	C_{bzip2} (LIT)	C_{PPMD} (LIT)	LIT $s \times S_c$
paper6	3.10	3.32	2.41	3.33	3.50	2.28	3.42	**3.62**	**2.21**
progc	3.16	3.39	2.36	3.28	3.48	2.30	3.33	**3.55**	**2.25**
news	3.18	3.40	2.35	3.26	3.47	2.31	3.31	**3.52**	**2.27**
book1	3.31	3.47	2.30	3.46	3.58	2.23	3.49	**3.61**	**2.22**
book2	3.88	4.07	1.96	4.02	4.19	1.91	4.08	**4.26**	**1.88**
bib	4.05	4.30	1.86	4.14	4.37	1.83	4.24	**4.47**	**1.79**
trans	4.61	5.45	1.47	5.46	5.69	1.41	5.55	**5.84**	**1.37**

Table 12: Relationship between Compression Factors (using Bzip2) and file size and entropy $k \times S_c$ for a few files from Calgary Corpus

Now let us compare the context level entropies of alice29.txt (Canterbury Corpus) transformed with LIPT and LIT. Table 13 gives level-wise entropies (column 3 and 5). The first column of Table 13 gives the order of the context (context level). The second and the fourth columns give the total count of predicting next input character using the respective context order for respective methods. For instance the count 108186 for LIPT in the row for context level 5 denotes that 108186 characters were predicted using up to five preceding characters using PPMD method. The third and fifth columns give the entropies or the bits needed to encode each character in the respective context level.

Context Length (Order)	LIPT (count)	LIPT (Entropy-BPC)	LIT (count)	LIT (Entropy-BPC)
5	108186	2.09	84534	2.31
4	15266	3.12	18846	3.22
3	7550	2.03	9568	2.90
2	6762	2.18	6703	1.84
1	2489	3.50	2726	3.52
0	79	6.35	79	6.52
Total /Average	140332	2.32	122456	2.50

Table 13: Entropy comparison for LIPT and LIT for alice29.txt from Canterbury Corpus

The entropy in each context level is calculated by multiplying the probability of the context level, which is the count for that context level divided by the total count (size of file in bytes), by the total number of bits needed to encode all the characters in the respective context. Bits needed to encode each character in respective context is calculated dynamically based on the probability of occurrence of input character given a certain preceding string of characters with length equal to respective context order. The $\log_2$ of this probability gives the number of bits to encode that particular character. The sum off all number of bits needed to encode each character using the probability of

occurrence of that character after a certain string in the respective context gives the total bits needed to encode all the characters in that context level.

Note that in Table 13 by average (in last row of the table) we mean the weighted average which is the sum of BPC multiplied by respective count in each row, divided by the total count. Note also that the average entropy for PPMD with LIT is more than the entropy for PPMD with LIPT, whereas LIT outperforms LIPT in average BPC performance. Although the entropy is higher but note that the total size of the respective transform files are different. LIPT has larger file size compared to LIT. Multiplying entropy which is average bits/symbol or character with total count from Table 13, for LIPT we have 2.32× 140332= 325570.24 bits, which is equal to 40696.28 bytes. From Table 13 for LIT, we have 2.50×122456= 306140 bits, which is equal to 38267.5 bytes. We see that the total number of bits and hence bytes for LIT is less than LIPT. This argument can be applied to justify performance of our transforms with Bzip2 as it is BWT based method and BWT exploits context. This can be shown for other transforms and for other files as well. Here we have only given one file as an example to explain. We have obtained this data for all the files in Calgary corpus as well.

Our transforms keep the word level context of the original text file but adopt a new context structure on the character level. The frequency of the repeated words remains the same in both the original and transformed file. The frequency of characters is different. Both these factors along with the reduction in file size contribute towards the better compression with our transforms. The context structure affects the entropy S or S_t and reduction in file size affects s. We have already discussed that compression is inversely proportional to the product of these two variables. In all our transforms, to generate our transformed dictionary, we have sorted the words according to frequency of usage in our English Dictionary D. For LIPT, words in each length block in English Dictionary D are sorted in descending order according to frequency. For ILPT, there is a sorting based on descending order of frequency inside each initial letter block of English Dictionary D. For NIT, there is no blocking of words. The whole dictionary is one block. The words in the dictionary are sorted in descending order of frequency. LIT uses the same structure of dictionary as NIT. Sorting of words according to frequency plays a vital role in the size of transformed file and also its entropy. Arranging the words in descending order of usage frequency results in shorter codes for more frequently occurring words and larger for less frequently occurring words. This fact leads to smaller file sizes.

8. Conclusions

In this chapter, we have given an overview of classical and recent lossless text compression algorithms and then presented our research on text compression based on transformation of text as a preprocessing step for use by the available compression algorithms. For comparison purposes, we were primarily concerned with gzip, Bzip2, a version of PPM called PPMD and word based Huffman. We give theoretical explanation of why our transforms improved the compression performance of the algorithms. We have developed an web site (http://vlsi.cs.ucf.edu) as a test bed for all compression

algorithms. To use this, one has to click the "online compression utility" and the client could then submit any text file for compression using all the classical compression algorithms, some of the most recent algorithms including Bzip2, PPMD, YBS, RK and PPMonstr and, of course, all the transformed based algorithms that we developed and reported in this chapter. The site is still under construction and is evolving. One nice feature is that the client can submit a text file and obtain statistics of all compression algorithms presented in the form of tables and bar charts. The site is being integrated with the Canterbury website.

Acknowledgement

The research reported in this chapter is based on a research grant supported by NSF Award No. IIS-9977336. Several members of the M5 Research Group at the School of Electrical Engineering and Computer Science of University of Central Florida participated in this research. The contributions of Dr. Robert Franceschini and Mr. Holger Kruse with respect to star transform work are acknowledged. The collaboration of Mr. Nitin Motgi and Mr. Nan Zhang in obtaining some of the experimental results is also gratefully acknowledged. Mr. Nitin Motgi's help in developing the compression utility website is also acknowledged.

References

[Arna00] Z. Arnavut. Move-to-Front and Inversion Coding. *Proceedings of Data Compression Conference*, Snowbird Utah, pp. 193-202, 2000.

[AwMu01] F. Awan, A. Mukherjee. LIPT: A Lossless Text Transform to Improve Compression. *International Conference on Information Theory: Coding and Computing*, Las Vegas Nevada, IEEE Computer Society, April 2001, pp 452- 460.

[Awan01] F. Awan, Lossless Reversible Text Transforms, *MS thesis*, University of Central Florida, July, 2001.

[BaKS99] B. Balkenhol, S. Kurtz , and Y. M. Shtarkov. Modifications of the Burrows Wheeler Data Compression Algorithm . *Proceedings of Data Compression Conference*, Snowbird Utah, pp. 188-197, 1999.

[BeM89] T. C. Bell and A. Moffat. *A note on the DMC data compression scheme*. The British Computer Journal, 32(1):16--20, 1989.

[BuWh94] M. Burrows and D.J. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. *SRC Research Report* 124, Digital Systems Research Center, Palo Alto, CA., 1994.

[Bzip2] Bzip2 memory usage http://krypton.mnsu.edu/krypton/software/bzip2.html

[Cant00] Calgary and Canterbury Corpi http://corpus.canterbury.ac.nz

[Chap00] B. Chapin. Switching Between Two On-line List Update Algorithms for Higher Compression of Burrows-Wheeler Transformed Data. *Proceedings of Data Compression Conference*, Snowbird Utah, pp. 183-191, 2000.

[ClW84] J.G. Cleary, and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. Comm. COM*-32, 4 (1984), pp. 396-402.

[ClTW95] J.G. Cleary, W.J. Teahan, and Ian H. Witten. Unbounded Length Contexts for PPM, *Proceedings of Data Compression Conference*, March 1995, pp. 52-61.

[CoH87] G.V. Cormack and R.N. Horspool, Data Compression Using Dynamic Markov Modeling, *Computer Journal*, Vol. 30, No. 6, 1987,pp. 541-550.

[Effo00] M. Effros. PPM Performance with BWT Complexity: A New Method for Lossless Data Compression. *Proceedings of Data Compression Conference*, Snowbird Utah, pp. 203-212, 2000.

[Fenw96] P. Fenwick. Block Sorting Text Compression. *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australia, January 31 – February 2, 1996.

[FrMu96] R Franceschini and A. Mukherjee. Data Compression Using Encrypted Text. Proceedings of the third Forum on Research and Technology, Advances on Digital Libraries, ADL 96, pp. 130-138.

[GBLL98] J. D. Gibson, T. Berger, T. Lookabaugh, D. Lindbergh and R.L Baker. Digital Compression for Multimedia: Principles and Standards. Morgan Kaufmann, 1998.

[Gute71] http://www.promo.net/pg/

[Howa93] P.G.Howard. The Design and Analysis of Efficient Lossless Data Compression Systems (Ph.D. thesis). Providence, RI:Brown University, 1993.

[Huff52] D. A. Huffman, A Method for the Construction of Minimum Redundancy Codes, *Proceedings of the Institute of Radio Engineers* 40, 1952, pp.1098-1101.

[KrMu97-1] H. Kruse and A. Mukherjee. Data Compression Using Text Encryption. *Proceedings of Data Compression Conference*, 1997, IEEE Computer Society Press, pp. 447.

[KrMu97-2] H. Kruse and A. Mukherjee. Preprocessing Text to Improve Compression Ratios. *Proceedings of Data Compression Conference*, 1998, IEEE Computer Society Press 1997, pp. 556.

[Lar98] N.J. Larsson. The Context Trees of Block Sorting Compression. N. Jesper Larsson: The Context Trees of Block Sorting Compression. *Proceedings of Data Compression Conference*, 1998, pp 189- 198.

[Moff90] A. Moffat. Implementing the PPM data Compression Scheme, IEEE Transaction on Communications, 38(11), pp.1917-1921, 1990

[PPMDH] PPMDH ftp://ftp.elf.stuba.sk/pub/pc/pack/

[RK-1] RK archiver http://rksoft.virtualave.net/

[RK-2] RK archiver http://www.geocities.com/SiliconValley/Lakes/1401/compress.html

[SaOH00] K. Sadakane, T. Okazaki, and H. Imai. Implementing the Context Tree Weighting Method for Text Compression. *Proceedings of Data Compression Conference*, Snowbird Utah, pp. 123-132, 2000.

[Sal00] D. Salomon. Data Compression: The Complete Reference. 2nd Edition, Springer Verlag, 2000.

[Say00] K. Sayood. Introduction to Data Compression. Morgan Kaufman Publishers, 1996.

[Sewa00] J. Seward. On the Performance of BWT Sorting Algorithms. *Proceedings of Data Compression Conference*, Snowbird Utah, pp. 173-182, 2000.

[ShW98] C. E. Shannon, W. Weaver, The Mathematical Theory of Communication, University of Illinois Press, 1998.

[Sh48] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, Vol. 27, pp.379-423,623-656, 1948.

[TREC00] http://trec.nist.gov/data.html

[WiST95] F. Willems, Y. M. Shtarkov, and T .J.Tjalkens. The Context-Tree Weighting Method: Basic Properties. *IEEE Transaction on Information Theory*, IT-41(3), pp. 653-664, 1995.

[WiMB99] I. H. Witten, A. Moffat, T. Bell. Managing Gigabyte, Compressing and Indexing Documents and Images. 2nd Edition, Morgan Kaufmann Publishers, 1999.

[YBS] YBS compression algorithm http://artest1.tripod.com/texts18.html

[ZiL77] A. Lempel and J. Ziv. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, pages 337--343, May 1977.

A Dictionary-Based Multi-Corpora Text Compression System

Weifeng Sun Amar Mukherjee Nan Zhang
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL. 32816
{wsun, amar, nzhang}@cs.ucf.edu

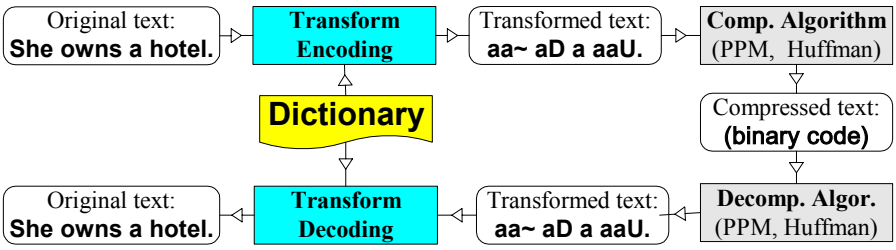
Abstract *In this paper we introduce StarZip, a multi-corpora lossless text compression utility which incorporates StarNT, our newly proposed transform algorithm. StarNT is a dictionary-based fast lossless text transform algorithm which utilizes ternary search tree to expedite transform encoding. For large files, viz. 400 Kbytes or more, our experiments show that the compression time is no worse than those obtained by bzip2 and gzip, and much faster than PPMD. However, if the file size is small, our algorithm is 28.1% and 50.4% slower than bzip2 -9, gzip -9 respectively and 21.1% faster compared to PPMD. We also achieve a superior compression ratio than almost all the other recent efforts based on BWT and PPM. StarNT is especially suitable for domain-specific lossless text compression used for archival storage and retrieval. Using domain-specific dictionaries, StarZip achieve an average improvement (in terms of BPC) of 13% over bzip2 –9, 19% over Gzip -9, and 10% over PPMD.*

7. Introduction

It is well known that there are theoretical predictions on how far a source file can be losslessly compressed, but no existing compression methods consistently attain these bounds over wide classes of text files. Researchers in the field of lossless text compression have developed several sophisticated approaches, such as Huffman encoding, arithmetic encoding, the Ziv-Lempel family, Dynamic Markov Compression, Prediction by Partial Matching (PPM [Moff90]), and Burrow-Wheeler Transform (BWT [BuWh94]) based algorithms, etc. LZ-family methods are dictionary based compression algorithm. Variants of LZ algorithm form the basis of UNIX compress, gzip and pkzip tools. PPM encodes each symbol by considering the preceding k symbols (an "order k" context). PPM achieves better compression than any existing compression algorithms, but it is intolerably slow and also consumes large amount of memory to store context information. Several efforts have been made to improve PPM [Howa93, Effr00]. BWT rearranges the symbols of a data sequence that share the same unbounded context by cyclic rotation followed by lexicographic sort operations. BWT utilize move-to-front and an entropy coder as the backend compressor. A number of efforts have been made to improve its efficiency [BKSh99, Chap00, Arna00].

In the recent past, the M5 Data compression group, University of Central Florida (http://vlsi.cs.ucf.edu/) has developed a family of reversible Star-transformations[ArMu01, FrMu96] which applied to a source text along with a backend compression algorithm, achieves better compression. The transformation is designed to make it easier to compress the source file. *Figure 1* illustrates the paradigm. The basic idea of the transform module is to transform the text into some intermediate form, which can be compressed with better efficiency. The transformed text is provided to a backend compressor which compresses the transformed text. The decompress process is just the reverse of the compression process. The drawback of these compression algorithms is

Figure 1. Text transform paradigm



that execution time performance and runtime memory expenditure is relatively high compared with the backend compression algorithm such as bzip2 and gzip.

In this paper we first introduce a fast algorithm for transform encoding and transform decoding, called *StarNT*. Facilitated with our proposed transform algorithm, bzip2 -9, gzip –9 and PPMD all achieve a better compression performance in comparison to most of the recent efforts based on PPM and BWT. Results shows that, for Calgary corpus, Canterbury corpus and Gutenberg corpus, StarNT achieves an average improvement in compression ratio of 11.2% over bzip2 –9, 16.4% over gzip –9, and 10.2% over PPMD. This algorithm utilizes *Ternary Search Tree* [BeSe97] in the encoding module. With a finely tuned dictionary mapping mechanism, we can find a word in the dictionary at time complexity *O(1)* in the transform decoding module. Results shows that for all corpora, the average compression time using the transform algorithm with bzip2 –9, gzip –9 and PPMD is 28.1% slower, 50.4% slower and 21.2% faster compared to the original bzip2 –9, gzip –9 and PPMD respectively. The average decompression time using the new transform algorithm with bzip2 –9, gzip –9 and PPMD is 100% slower, 600% slower and 18.6% faster compared to the original bzip2 –9, gzip –9 and PPMD respectively. However, since the decoding process is fairly fast, this increase is negligible. We draw a significant conclusion that bzip2 in conjunction with StarNT is better than both gzip and PPMD both in time complexity and compression performance.

Based on this transform, we developed StarZip[1], a domain-specific lossless text compression utility for archival storage and retrieval. StarZip uses specific dictionaries for specific domains. In our experiment, we created five corpora from publicly available website, and derived five domain-specific dictionaries. Results show that the average

[1] The name StarZip was suggested by Dr. Mark R. Nelson of Dr. Dobb's Journal. A featured article by Dr. Nelson appears in Dr. Dobb's Journal, August 2002, pp. 94-96.

BPC improved 13% over bzip2 –9, 19% over Gzip -9, and 10% over PPMD for these five corpora.

8. StarNT: the Dictionary-Based Fast Transform

In this section, we will first discuss three considerations from which this transformation is derived. After that a brief discussion about ternary search tree is presented, followed by detailed description about how the transform works.

8.1. Why Another New Transform?

There are three considerations that lead us to this transform algorithm.

First, we gathered data of word frequency and length of words information from our collected corpora (All these corpora are publicly available), as depicted in *Figure 2*. It is clear that almost more than 82% of the words in English text have the lengths greater than three. If we can recode each English word with a representation of no more than three symbols, then we can achieve a certain kind of "*pre-compression*". This consideration can be implemented with a fine-tuned transform encoding algorithm, as is described later.

The second consideration is that the transformed output should be compressible to the backend compression algorithm. In other words, the transformed immediate output should maintain some of the original context information as well as provide some kind of "artificial" but strong context. The reason behind this is that we choose BWT and PPM algorithms as our backend compression tools. Both of them predict symbols based on context information.

Finally, the transformed codewords can be treated as the offset of words in the transform dictionary. Thus, in the transform decoding phrase we can use a hash function to achieve *O(1)* time complexity for searching a word in the dictionary. Based on this consideration, we use a continuously addressed dictionary in our algorithm. In contrast, the dictionary is split into 22 sub-blocks in LIPT [AwMu01]. Results show that the new transform is better than LIPT not only in time complexity but also in compression performance.

The performance of search operation in the dictionary is the key for fast transform encoding. Here there are several approaches: the first one is hash table, which is really fast, but designing a fine-skewed hash function is very difficult. And unsuccessful searches using hash table are disproportionately slow. Another option is digital search tries, which are also very fast, however, they have exorbitant space requirements: suppose each node has 52-way branching, then one node will typically occupy at least 213 bytes. A three-level digital search tries with this kind of nodes will consume (1+52+52*52)*213=587,241 bytes! Or, we can use binary search tree (LIPT take this approach), which is really space efficient. Unfortunately, the search cost is quite high for binary search. In this transform, we utilize ternary search trees, which provide a very fast transform encoding speed at a low

Figure 2: Frequency of words versus length of words in English text

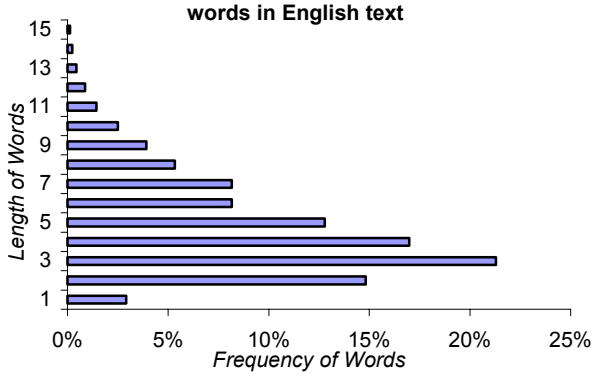
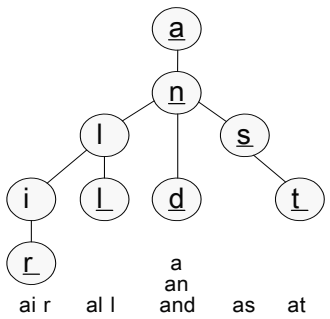


Figure 3: a Ternary Search Tree for seven words



storage overhead.

8.2. Ternary Search Tree

Ternary search trees [BeSe97] are similar to digital search tries in that strings are split in the trees with each character stored in a single node as *split char*. Besides, three pointers are included in one node: left, middle and right. All elements less than the split character are stored in the left child, those greater than the split character are stored in the right child, while the middle child contains all elements with the same character.

Search operations in ternary search trees are quite straightforward: current character in the search string is compared with the *split char* at the node. If the search character is less than the *split char*, then go to the left child; if the search character is greater than the *split char*, go to the right child; otherwise, if the search character is equal to the *split char*, just go to the middle child, and proceed to the next character in the search string. Searching for a string of length k in a ternary search tree with n strings will require at most $O(\log n+k)$ comparisons. The construction time for the ternary tree takes $O(n\log n)$ time.

Furthermore, ternary search trees are quite space-efficient. In *Figure 3*, seven strings are stored in this ternary search tree. Only nine nodes are needed. If multiple strings have same prefix, then the corresponding nodes to these prefixes can be reused, thus memory requirements is reduced in scenarios with large amounts of data.

In the transform encoding module, words in the dictionary are stored in the ternary search trees with the address of corresponding codewords. The ternary search tree is split into 26 distinct ternary search trees. An array is used to store the addresses of these ternary search trees corresponding to the letters [a..z] of the alphabet in the main root node. Words have the same starting character are stored in same sub-tree, viz. all words starting with ‘a’ in the dictionary exist in the first sub-tree, while all words start with ‘b’ in second sub-tree, and so on.

In each leaf node of the ternary search tree, there is a pointer which points to the corresponding codeword. All codewords are stored in a global memory that is prepared in advance. Using this technique we can avoid storing the codeword in the node, which enables a lot of flexibility as well as space-efficiency. To expedite the tree-build operation, we allocate a big pool of nodes to avoid overhead time for allocating storage for nodes in sequence.

Ternary search tree is sensitive to insertion order: if we insert nodes in a good order (middle element first), we end up with a balanced tree for which the construction time is the small; if we insert nodes in the order of the frequency of words in the dictionary, then the result would be a skinny tree that is very costly to build but efficient to search. In our experiment, we confirmed that insertion order has a lot of performance impact in the transform encoding phase. Our approach is just to follow the *natural* order of words in the dictionary. Result shows that this approach works very well.

8.3. Dictionary Mapping

The dictionary used in this experiment is prepared in advance, and shared by both the transform encoding module and the transform decoding module. In view of the three considerations mentioned in section 2.2, words in the dictionary ***D*** are sorted using the following rules:

- Most frequently used words are listed at the beginning of the dictionary. There are 312 words in this group.

- The remaining words are stored in ***D*** according to their lengths. Words with longer lengths are stored after words with shorter lengths. Words with same length are sorted according to their frequency of occurrence.
- To achieve better compression performance for the backend data compression algorithm, only letters [a..zA..Z] are used to represent the codeword.

With the ordering specified above, each word in ***D*** is assigned a corresponding codeword. The first 26 words in ***D*** are assigned "a", "b", … , "z" as their codewords. The next 26 words are assigned "A", "B", … , "Z". The 53rd word is assigned "aa", 54th "ab". Following this order, "ZZ" is assigned to the 2756th word in ***D***. The 2757th word in ***D*** is assigned "aaa", the following 2758th word is assigned "aab", and so on. Hence, the most frequently occurred words are assigned codewords form "a" to "eZ". Using this mapping mechanism, totally 52+52*52+52*52*52 = 143,364 words can be included in ***D***.

8.4. Transform Encoding

In the transform encoding module, the shared static base dictionary is read into main memory and the corresponding ternary search tree is constructed.

A *replacer* is initiated to read in the input text character by character, which performs the replace operation when it recognizes that the string of a certain length of input symbols (or the lower case form of this string sequence) exists in the dictionary. Then it outputs the corresponding codeword (appended with a special symbol if needed) and continues. If the input symbol sequence does not exist in the dictionary, it will be output with a prefix escape character "*". Currently only single words are stored in the dictionary. For future implementation, the transform module will contain phrases or sequence of words with all kinds of symbols (such as upper case letter, smaller case letters, blank symbols, punctuations, etc) in the dictionary.

The proposed new transform differs from our earlier Star-family transforms with respect to the meaning of the character '*'. Originally it was used to indicate the beginning of a codeword. In our new transform, it denotes that the following word does not exist in the dictionary ***D***. The main reason for this change is to minimize the size of the transformed intermediate text file, because smaller size can expedite the backend encoding/decoding.

Currently only lower-case words are stored in the dictionary ***D***. Special operations were designed to handle the first-letter capitalized words and all-letter capitalized words. The character '~' appended at the end of an encoded word denotes that the first letter of the input text word is capitalized. The appended character '`' denotes that all letters of the word are capitalized. The character '\' is used as escape character for encoding the occurrences of '*', '~', '`', and '\' in the input text.

8.5. Transform Decoding

The transform decoding module performs the inverse operation of the transform-encoding module. The escape character and special symbols ('*', '~', '`', and '\') are recognized and processed, and the transformed words are replaced with their original forms, which are stored continuously in a pre-allocated memory to minimize the memory usage. And their addresses are stored in an array sequentially. There is a very important property of the dictionary mapping mechanism that can be used to achieve O(1) time complexity to search a word in the dictionary. Because codewords in the dictionary are assigned sequentially, and only letters [a..zA..Z] are used, these codewords can be

deemed as the address in the dictionary. In our implementation, we use a very simple one-to-one mapping to calculate the index of the corresponding original words in the array that stores the address of the dictionary words.

9. Performance Evaluation

Our experiments were carried out on a 360MHz Ultra Sparc-IIi Sun Microsystems machine housing SunOS 5.7 Generic_106541-04. We choose bzip2 (-9), PPMD (order 5) and gzip (-9) as the backend compression tool. In this section, all compression results are derived from the Canterbury-Calgary and Gutenburg Corpus. The reason why we choose bzip2, gzip and PPMD as our backend compressor is that bzip2 and PPM outperform other compression algorithms, and gzip is widely used.

9.1. Timing Performance

9.1.1. Timing Performance with Backend Compression Algorithm

When backend data compression algorithms, i.e. bzip2 -9, gzip -9 and PPMD (k=5) are used along with the new transform, the compression system also provides a favorable timing performance. Following conclusions can be drawn form *Table 1* and *Table 2*:

1. The average compression time using the new transform algorithm with bzip2 –9, gzip –9 and PPMD is 28.1% slower, 50.4% slower and 21.2% faster compared to the original bzip2 –9, gzip –9 and PPMD respectively.
2. The average decompression time using the new transform algorithm with bzip2 –9, gzip –9 and PPMD is 1 and 6 times slower, and is 18.6% faster compared to the original bzip2 –9, gzip –9 and PPMD respectively. However, since the decoding process is fairly fast for bzip2 and gzip, this increase is negligible.

It must be pointed out that when the new transform is concatenated with backend compressor, the increase in compression time is imperceptible to human being, if the size of the input text file is small, viz. less than 100Kbytes. As the size of the input text file increases, the relative impact the new transform introduced to backend compressor decreases proportionally. Especially, if the text is very large, compression using the transform will be faster than that without using the transform. An example in case is file bible.txt (size 4,047,392), the runtime decreases when the transform is applied on bzip2. The reason behind this is the file size decrease introduced by the transform, which makes backend compressor runs faster. It also should be pointed out that our programs have not yet been well optimized, there is potential for less time and smaller memory usage.

Table 1: Comparison of Encoding Speed of Various Compressor with/without Transform (in seconds)

Corpus	bzip2	bzip2 + StarNT	Bzip2 + LIPT	Gzip	gzip + StarNT	gzip + LIPT	PPMD	PPMD + StarNT	PPMD + LIPT
Calgary	0.36	0.76	1.33	0.23	0.86	1.7	9.58	7.94	9.98
Canterbury	2.73	3.04	5.22	2.46	3.36	6.59	68.3	55.7	69.2
Gutenburg	4.09	4.4	7.01	2.28	3.78	9.67	95.4	75.2	90.9
AVERAGE	1.69	2.05	3.47	1.33	2.06	4.47	41.9	33.9	41.9

Table 2: Comparison of Decoding Speed of Various Compressor with/without Transform (in seconds)

Corpus	bzip2	bzip2 + StarNT	bzip2 + LIPT	gzip	Gzip + StarNT	gzip + LIPT	PPMD	PPMD + StarNT	PPMD + LIPT
Calgary	0.13	0.33	1.66	0.04	0.27	1.64	9.65	8.07	10.9
Canterbury	0.82	1.53	6.77	0.22	1.16	9.15	71.2	57.8	77.2
Gutenburg	1.15	2.22	8.46	0.29	1.44	7.99	95.4	76.9	98.7
AVERAGE	0.51	1	4.4	0.14	0.72	5.27	43	35	46.4

The data in *Table 2* and *Table 3* also shows that the new transform works better than LIPT when they are applied with backend compression algorithm both in the compression phase and decompression phase.

9.1.2. Improvement upon LIPT

Table 1 illustrates the transform encoding/decoding time of our transform algorithm against LIPT without any backend compression algorithm involved. All data are un-weighted average value of 10 runs. The results can be summarized as follows.

1. For all corpora, the average transform encoding and decoding times using the new transform decrease about 76.3% and 84.9%, respectively, in comparison to times taken by LIPT.
2. The decoding module runs faster than encoding module by 39.3% on average. The main reason is that the hash function used in the decoding phase is more efficient than the ternary search tree in the encoding module.

Table 3: Comparison of Transform Encoding and Decoding Speed (in seconds)

Corpora	StarNT		LIPT	
	Transform Encoding	Transform Decoding	Transform Encoding	Transform Decoding
Calgary	0.42	0.18	1.66	1.45
Canterbury	1.26	0.85	5.7	5.56
Gutenburg	1.68	1.12	6.89	6.22
AVERAGE	0.89	0.54	3.75	3.58

9.2. Compression Performance of StarNT

We compared the compression performance (in terms of BPC, bits per character) of our proposed transform with the results of the recent improvements on BWT and PPM as listed on *Table 4* and *Table 5,* taken from the references given in the respective columns. From *Table 4* and *Table 5*, it can be seen that our transform algorithm outperforms almost all the other improvements. The detailed compression results in terms of BPC for our test corpus are listed in the Appendix. Following is the summary:

1. Facilitated with StarNT, bzip2 -9, gzip –9 and PPMD an average improvement in compression ratio of 11.2% over bzip2 –9, 16.4% over gzip –9, and 10.2% over PPMD.
2. The StarNT works better than LIPT when is applied with backend compressor.
3. In conjunction with bzip2, our transform algorithm achieves a better compression performance than the original PPMD. Combined with the timing performance, we conclude that bzip2+StarNT is better than PPMD both in time complexity and compression performance.

9.3. Space Complexity

In our implementation the transform dictionary is a static dictionary shared by both transform encoder and transform decoder. The typical size of the off-line dictionary is about 0.5MB. The run-time overhead is as follows: the memory requirement of the ternary search tree for our dictionary in the transform encoding phase is about 1M bytes. In the transform decoding phase, less memory is needed. Bzip2 –9 needs about 7600K. PPM is programmed to use about 5100K + file size. So our transform algorithm takes insignificant overhead compared to bzip2 and PPM in memory usage.

Table 1: BPC comparison of new approaches based on BWT

File	Size (byte)	Mbswic [Arna00]	bks98 [BKSh99]	best x of 2x-1 [Chap00]	bzip2+ LIPT	bzip2+ StarNT
bib	111261	2.05	1.94	1.94	1.93	**1.71**
book1	768771	2.29	2.33	2.29	2.31	**2.28**
book2	610856	2.02	2.00	2.00	1.99	**1.92**
news	377109	2.55	2.47	2.48	2.45	**2.29**
paper1	53161	2.59	2.44	2.45	2.33	**2.21**
paper2	82199	2.49	2.39	2.39	2.26	**2.14**
progc	39611	2.68	2.47	2.51	2.44	**2.32**
progl	71646	1.86	1.70	1.71	1.66	**1.58**
progp	49379	1.85	**1.69**	1.71	1.72	**1.69**
trans	93695	1.63	1.47	1.48	1.47	**1.22**
Average		2.20	2.09	2.10	2.06	**1.94**

Table 2: BPC comparison of new approaches based on PPM

File	Size (byte)	Multi-alphabet CTW order 16 [SOIm00]	NEW [Effr00]	PPMD+ LIPT	PPMD+ StarNT
Bib	111261	1.86	1.84	1.83	**1.62**
book1	768771	**2.22**	2.39	2.23	2.24
book2	610856	1.92	1.97	1.91	**1.85**
news	377109	2.36	2.37	2.31	**2.16**
paper1	53161	2.33	2.32	2.21	**2.10**
paper2	82199	2.27	2.33	2.17	**2.07**
progc	39611	2.38	2.34	2.30	**2.17**
progl	71646	1.66	1.59	1.61	**1.51**
progp	49379	1.64	**1.56**	1.68	1.64
trans	93695	1.43	1.38	1.41	**1.14**
Average		2.01	2.01	1.97	**1.85**

10. StarZip: A Multi-corpora Text Compression Tool

We have developed a multi-corpora text compression system StarZip, which incorporates StarNT as the basic transform. One of the key features of our StarZip compression system is to develop domain specific dictionaries and provide tools to develop such dictionaries. We made preliminary studies on specific dictionaries and obtained some encouraging results. We experimented with five corpora derived form ibiblio.com [literature (3064 files, 1.2 G), History (233 files, 9.11M), Political Science (969 files, 33.4M), Psychology (55 files, 13.3M) and Computer Network (RFC, 3237 files, 145M)] and created separate domain-specific dictionaries for each corpus with entries of 60533, 39740, 38464, 45165 and 13987, respectively. Each dictionary is

created using the algorithm mentioned in section 2.3. For Bzip2, the average BPC using domain-specific dictionary shows an improvement of 13% compared with the original Bzip2. Using domain-specific dictionary gives a 6% improvement in BPC compared with the case when a common English dictionary is used. For Gzip, the average BPC using domain-specific dictionary gives an improvement of 19% and 7% over using a common dictionary. For PPMD, the average BPC using domain-specific dictionary has an improvement of 10% compared with the original PPMD and gives 5% gain in BPC when a common dictionary is used. We propose to conduct similar experiments on a large number of corpora to evaluate the effectiveness of our approach.

11. Conclusion

In this paper, we have proposed a new transform algorithm which utilizes ternary search tree to expedite transform encoding operation. This transform algorithm also includes an efficient dictionary mapping mechanism based on which searching operation can be performed with time complexity O(1) in the transform decoding module.

We compared the compression effectiveness versus compression/decompression speed and compression ratio when bzip2 –9, gzip –9 and PPMD are used as backend compressor with our transform algorithm, as illustrated in *Figure 4* and *Figure 5*. It is very clear that bzip2 + StarNT could provide a better compression performance that maintains an appealing compression and decompression speed.

Figure 4: Compression effectiveness versus compression speed

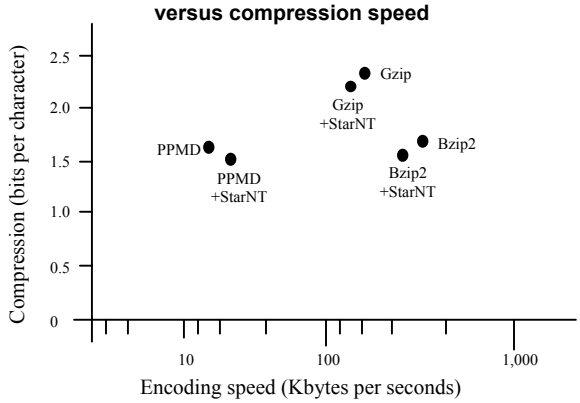
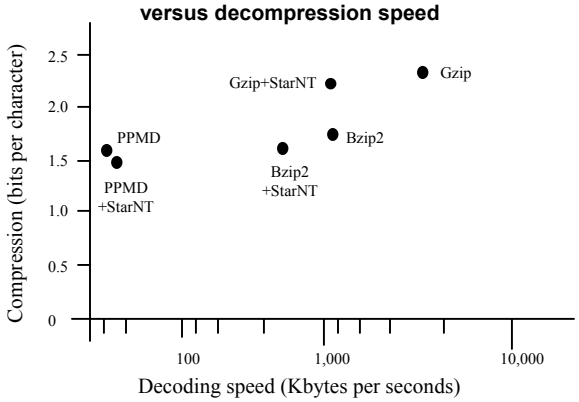


Figure 5: Compression effectiveness versus decompression speed



Acknowledgement

This research is partially supported by NSF grant number: IIS-9977336 and IIS-0207819.

References

[AwMu01] F. Awan and A. Mukherjee, "LIPT: A Lossless Text Transform to improve compression", Proceedings of International Conference on Information and Theory : Coding and Computing, IEEE Computer Society, Las Vegas Nevada, 2001.

[Arna00] Z. Arnavut, "Move-to-Front and Inversion Coding", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird, Utah, March 2000, pp. 193-202.

[BKSh99] B. Balkenhol, S. Kurtz , and Y. M. Shtarkov, "Modifications of the Burrows Wheeler Data Compression Algorithm", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, March 1999,pp. 188-197.

[BeSe97] J. L. Bentley and Robert Sedgewick, "Fast Algorithms for Sorting and Searching Strings", Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, January, 1997

[BuWh94] M. Burrows and D.J. Wheeler, "A Block-Sorting Lossless Data Compression Algorithm", *SRC Research Report* 124, Digital Systems Research Center, Palo Alto, CA, 1994.

[Chap00] B. Chapin, "Switching Between Two On-line List Update Algorithms for Higher Compression of Burrows-Wheeler Transformed Data", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, March 2000, pp. 183-191.

[Effr00] M. Effros, "PPM Performance with BWT Complexity: A New Method for Lossless Data Compression", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, March 2000, pp. 203-212.

[FrMu96] R. Franceschini and A. Mukherjee, "Data Compression Using Encrypted Text", *Proceedings of the third Forum on Research and Technology*, Advances on Digital Libraries, ADL 96, pp. 130-138.

[Howa93] P.G.Howard, "The Design and Analysis of Efficient Lossless Data Compression Systems", Ph.D. thesis. Providence, RI:Brown University, 1993.

[KrMu98] H. Kruse and A. Mukherjee, "Preprocessing Text to Improve Compression Ratios", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, 1998, pp. 556.

[Moff90] A. Moffat, "Implementing the PPM data Compression Scheme", *IEEE Transaction on Communications*, 38(11), pp.1917-1921, 1990

[SOIm00] K. Sadakane, T. Okazaki, and H. Imai, "Implementing the Context Tree Weighting Method for Text Compression", *Proceedings of Data Compression Conference,* IEEE Computer Society, Snowbird Utah, March 2000, pp. 123-132.

[Sewa00] J. Seward, "On the Performance of BWT Sorting Algorithms", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird, March 2000, pp. 173-182.

Appendix: Compression Results (BPC) Using StarNT

File	Size (byte)	bzip2 –9	bzip2 –9 +StarNT	gzip –9	gzip –9 +StarNT	PPMD	PPMD +StarNT
paper5	11954	3.24	2.76	3.34	2.78	2.98	2.56
paper4	13286	3.12	2.46	3.33	2.55	2.89	2.34
paper6	38105	2.58	2.29	2.77	2.40	2.41	2.17
progc	39611	2.53	2.32	2.68	2.45	2.36	2.17
paper3	46526	2.72	2.28	3.11	2.47	2.58	2.24
progp	49379	1.74	1.69	1.81	1.76	1.70	1.64
paper1	53161	2.49	2.21	2.79	2.35	2.33	2.10
progl	71646	1.74	1.58	1.80	1.65	1.68	1.51
paper2	82199	2.44	2.14	2.89	2.35	2.32	2.07
trans	93695	1.53	1.22	1.61	1.25	1.47	1.14
bib	111261	1.97	1.71	2.51	2.12	1.86	1.62
news	377109	2.52	2.29	3.06	2.57	2.35	2.16
book2	610856	2.06	1.92	2.70	2.24	1.96	1.85
book1	768771	2.42	2.28	3.25	2.66	2.30	2.24
grammar.lsp	3721	2.76	2.42	2.68	2.38	2.36	2.06
xargs.1	4227	3.33	2.90	3.32	2.87	2.94	2.57
fields.c	11150	2.18	1.98	2.25	2.03	2.04	1.81
cp.html	24603	2.48	2.01	2.60	2.13	2.26	1.85
asyoulik.txt	125179	2.53	2.27	3.12	2.58	2.47	2.24
alice29.txt	152089	2.27	2.06	2.85	2.38	2.18	2.00
lcet10.txt	426754	2.02	1.81	2.71	2.14	1.93	1.78
plrabn12.txt	481861	2.42	2.23	3.23	2.60	2.32	2.22
world192.txt	2473400	1.58	1.36	2.33	1.87	1.49	1.30
bible.txt	4047392	1.67	1.53	2.33	1.87	1.60	1.47
kjv.gutenberg	4846137	1.66	1.55	2.34	1.94	1.57	1.47
anne11.txt	586960	2.22	2.05	3.02	2.47	2.13	2.01
1musk10.txt	1344739	2.08	1.88	2.91	2.34	1.91	1.82
world95.txt	2736128	1.57	1.34	2.37	1.89	1.49	1.29
average		2.28	2.02	2.70	2.25	2.14	1.92

Text Compression

Amar Mukherjee and Fauzia Awan
School of Electrical Engineering and Computer Science, University of Central Florida
amar@cs.ucf.edu

1. Introduction

In recent times, we have seen an unprecedented explosion of textual information through the use of Internet, digital libraries and information retrieval system. The advent of office automation systems, newspapers, journals and magazine repositories have brought the issue of maintaining archival storage for search and retrieval to the forefront of research. As an example, the TREC [TREC00] database holds around 800 million static pages having 6 trillion bytes of plain text equal to the size of a million books. Text compression is concerned with techniques for representing the digital text data in alternate representations that takes less space. Not only it helps conserve the storage space for archival and online data, it also helps system performance by requiring less number of secondary storage (disk or CD Rom) access and improves the network transmission bandwidth utilization by reducing the transmission time. Data compression methods are generally classified as *lossless or lossy*. Lossless compression allows the original data to be recovered exactly. Although used primarily for text data, lossless compression algorithms are useful in special classes of images such as medical imaging, finger print data, astronomical images and data bases containing mostly vital numerical data, tables and text information. In contrast, lossy compression schemes allow some deterioration and are generally used for video, audio and still image applications. The deterioration of the quality of lossy images are usually not detectable by human perceptual system, and the compression systems exploit this by a process called '*quantization*' to achieve compression by a factor of 10 to a couple of hundreds. Many lossy algorithms use lossless methods at the final stage of the encoding stage underscoring the importance of lossless methods for both lossy and lossless compression applications. This chapter will be concerned with lossless algorithms for textual information. We will first review in Section 2 the basic concepts of information theory applicable in the context of lossless text compression. We will then briefly describe the well-known compression algorithms in Sections 3. Detailed description of these and other methods are now available in several excellent recent books [Sal00; Say00; WiMB99; GBLL98 and others]. In Sections 4 through 6, we present our own research on text compression. In particular, we present a number of pre-processing techniques that transform the text in some intermediate forms that produce better compression performance. We give extensive test results of compression and timing performance with respect to text files in three corpuses: Canterbury, Calgary [Cant00] and Gutenberg [Gute71] corpus, along with discussion on storage overhead. In Section 7, we present a plausible information theoretic explanation of the compression performance of our algorithms. We conclude our chapter with a discussion of the compression utility website that is now integrated with Canterbury website for availability via the Internet.

2. Information Theory Background

The general approach to text compression is to find a representation of the text requiring less number of binary digits. In its uncompressed form each character in the text is represented by an 8-bit ASCII code[1]. It is common knowledge that such a representation is not very efficient because it treats frequent and less frequent characters equally. It makes intuitive sense to encode frequent characters with a smaller (less than 8) number of bits and less frequent characters with larger number of bits (possibly more than 8 bits) in order to reduce the *average* number of *bits per character* (BPC). In fact this principle was the basis of the invention of the so-called Morse code and the famous Huffman code developed in the early 50's. Huffman code typically reduces the size of the text file by about 50-60% or provides compression rate of 4-5 BPC [WiMB99] based on statistics of frequency of characters. In the late 1940's, Claude E. Shannon laid down the foundation of the information theory and modeled the text as the output of a source that generates a sequence of symbols from a finite alphabet A according to certain probabilities. Such a process is known as a *stochastic process* and in the special case when the probability of occurrence of the next symbol in the text depends on the previous symbols or its context it is called a *Markov process*. Furthermore, if the probability distribution of a typical sample represents the distribution of the text it is called an *ergodic process* [Sh48, ShW98]. The information content of the text source can then be quantified by the entity called *entropy H* given by

$$H = -\sum p_i \log p_i \tag{1}$$

where p_i denotes the probability of occurrence of the ith symbol in the text, sum of all symbol probabilities is unity and the logarithm is with respect base 2 and $-\log p_i$ is the amount of *information* in bits for the event (occurrence of the ith symbol). The expression of H is simply the sum of the number of bits required to represent the symbols multiplied by their respective probabilities. Thus the entropy H can be looked upon as defining the *average number of BPC* required to represent or encode the symbols of the alphabet. Depending on how the probabilities are computed or modeled, the value of entropy may vary. If the probability of a symbol is computed as the ratio of the number of times it appears in the text to the total number of symbols in the text, the so-called *static* probability, it is called an Order(0) model. Under this model, it is also possible to compute the *dynamic* probabilities which can be roughly described as follows. At the beginning when no text symbol has emerged out of the source, assume that every symbol is equiprobable[2]. As new symbols of the text emerge out of the source, revise the probability values according to the actual frequency distribution of symbols at that time. In general, an Order(k) model can be defined where the probabilities are computed based on the probability of distribution of the ($k+1$)-grams of symbols or equivalently, by

[1] Most text files do not use more than 128 symbols which include the alphanumerics, punctuation marks and some special symbols. Thus, a 7-bit ASCII code should be enough.

[2] This situation gives rise to what is called the zero-frequency problem. One cannot assume the probabilities to be zero because that will imply an infinite number of bits to encode the first few symbols since –log o is infinity. There are many different methods of handling this problem but the equiprobabilty assumption is a fair and practical one.

taking into account the context of the preceding k symbols. A value of k = -1 is allowed and is reserved for the situation when all symbols are considered equiprobable, that is, $p_i = \frac{1}{|A|}$, where $|A|$ is the size of the alphabet A. When $k=1$ the probabilities are based on *bigram* statistics or equivalently on the context of just one preceding symbol and similarly for higher values of k. For each value of k, there are two possibilities, the static and dynamic model as explained above. For practical reasons, a static model is usually built by collecting statistics over a test *corpus* which is a collection of text samples representing a particular domain of application (viz. English literature, physical sciences, life sciences, etc.). If one is interested in a more precise static model for a given text, a *semi-static* model is developed in a two-pass process; in the first pass the text is read to collect statistics to compute the model and in the second pass an encoding scheme is developed. Another variation of the model is to use a specific text to *prime* or seed the model at the beginning and then build the model on top of it as new text files come in.

Independent of what the model is, there is an entropy associated with each file under that model. Shannon's fundamental noiseless source coding theorem says that entropy defines a lower limit of the average number of bits needed to encode the source symbols [ShW98]. The "worst" model from information theoretic point of view is the order(-1) model, the equiprobable model, giving the maximum value H_m of the entropy. Thus, for the 8-bit ASCII code, the value of this entropy is 8 bits. The redundancy R is defined to be the difference[3] between the maximum entropy H_m and the actual entropy H. As we build better and better models by going to higher order k, lower will be the value of entropy yielding a higher value of redundancy. The crux of lossless compression research boils down to developing compression algorithms that can find an encoding of the source using a model with minimum possible entropy and exploiting maximum amount of redundancy. But incorporating a higher order model is computationally expensive and the designer must be aware of other performance metrics such as decoding or decompression complexity (the process of decoding is the reverse of the encoding process in which the redundancy is restored so that the text is again human readable), speed of execution of compression and decompression algorithms and use of additional memory.

Good compression means less storage space to store or archive the data, and it also means less bandwidth requirement to transmit data from source to destination. This is achieved with the use of a *channel* which may be a simple point-to-point connection or a complex entity like the Internet. For the purpose of discussion, assume that the channel is noiseless, that is, it does not introduce error during transmission and it has a *channel capacity* C which is the maximum number of bits that can be transmitted per second. Since entropy H denotes the average number of bits required to encode a symbol, C/H denotes the average number of symbols that can be transmitted over the channel per second [ShW98]. A second fundamental theorem of Shannon says that however clever you may get developing a compression scheme, you will never be able to transmit on

[3] Shannon's original definition is R/H_m which is the fraction of the structure of the text message determined by the inherent property of the language that governs the generation of specific sequence or words in the text [ShW98].

average more than C/H symbols per second [ShW98]. In other words, to use the available bandwidth effectively, *H* should be as low as possible, which means employing a compression scheme that yields minimum BPC.

3. Classification of Lossless Compression Algorithms

The lossless algorithms can be classified into three broad categories : *statistical methods*, *dictionary methods* and *transform based methods*. We will give a very brief review of these methods in this section.

3.1 Statistical Methods

The classic method of statistical coding is Huffman coding [Huff52]. It formalizes the intuitive notion of assigning shorter codes to more frequent symbols and longer codes to infrequent symbols. It is built bottom-up as a binary tree as follows: given the model or the probability distribution of the list of symbols, the probability values are sorted in ascending order. The symbols are then assigned to the leaf nodes of the tree. Two symbols having the two lowest probability values are then combined to form a parent node representing a composite symbol which replaces the two child symbols in the list and whose probability equals the sum of the probabilities of the child symbols. The parent node is then connected to the child nodes by two edges with labels '0' and '1'in any arbitrary order. The process is now repeated with the new list (in which the composite node has replaced the child nodes) until the composite node is the only node remaining in the list. This node is called the root of the tree. The unique sequence of 0's and 1's in the path from the root to the leaf node is the Huffman code for the symbol represented by the leaf node. At the decoding end the same binary tree has to be used to decode the symbols from the compressed code. In effect, the tree behaves like a dictionary that has to be transmitted once from the sender to receiver and this constitute an initial overhead of the algorithm. *This overhead is usually ignored in publishing the BPC results for Huffman code in literature*. The Huffman codes for all the symbols have what is called the *prefix property* which is that no code of a symbol is the prefix of the code for another symbol, which makes the code *uniquely decipherable(UD)*. This allows forming a code for a sequence of symbols by just concatenating the codes of the individual symbols and the decoding process can retrieve the original sequence of symbols without ambiguity. Note that a prefix code is not necessarily a Huffman code nor may obey the Morse's principle and a uniquely decipherable code does not have to be a prefix code, but the beauty of Huffman code is that it is UD, prefix and is also optimum within one bit of the entropy H. Huffman code is indeed optimum if the probabilities are $1/2^k$ where k is a positive integer. There are also Huffman codes called *canonical* Huffman codes which uses a look up table or dictionary rather than a binary tree for fast encoding and decoding [WiMB99,Sal00].

Note in the construction of the Huffman code, we started with a model. Efficiency of the code will depend on how good this model is. If we use higher order models, the entropy will be smaller resulting in shorter average code length. As an example, a word–based Huffman code is constructed by collecting the statistics of words in the text and building

a Huffman tree based on the distribution of probabilities of words rather than the letters of the alphabet. It gives very good results but the overhead to store and transmit the tree is considerable. Since the leaf nodes contain all the distinct words in the text, the storage overhead is equal to having an English words dictionary shared between the sender and the receiver. We will return to this point later when we discuss our transforms. Adaptive Huffman codes takes longer time for both encoding and decoding because the Huffman tree has to be modified at each step of the process. Finally, Huffman code is sometimes referred to as a variable length code (VLC) because a message of a fixed length may have variable length representations depending on what letters of the alphabet are in the message.

In contrast, the *arithmetic code* encodes a variable size message into fixed length binary sequence[RiL79]. Arithmetic code is inherently adaptive, does not use any lookup table or dictionary and in theory can be optimal for a machine with unlimited precision of arithmetic computation. The basic idea can be explained as follows: at the beginning the semi-closed interval [0,1) is partitioned into $|A|$ equal sized semi-closed intervals under the equiprobabilty assumption and each symbol is assigned one of these intervals. The first symbol, say a_1 of the message can be represented by a point in the real number interval assigned to it. To encode the next symbol a_2 in the message, the new probabilities of all symbols are calculated recognizing that the first symbol has occurred one extra time and then the interval assigned to a_1 is partitioned (as if it were the entire interval) into $|A|$ sub-intervals in accordance with the new probability distribution. The sequence a_1a_2 can now be represented without ambiguity by any real number in the new sub-interval for a_2. The process can be continued for succeeding symbols in the message as long as the intervals are within the specified arithmetic precision of the computer. The number generated at the final iteration is then a code for the message received so far. The machine returns to its initial state and the process is repeated for the next block of symbol. A simpler version of this algorithm could use the same static distribution of probability at each iteration avoiding re-computation of probabilities. The literature on arithmetic coding is vast and the reader is referred to the texts cited above [Sal00; Say00;WiMB99] for further study.

The Huffman and arithmetic coders are sometimes referred to as the *entropy coder*. These methods normally use an order(0) model. If a good model with low entropy can be built external to the algorithms, these algorithms can generate the binary codes very efficiently. One of the most well known modeler is *"prediction by partial match" (PPM)* [ClW84; Moff90]. PPM uses a finite context Order(k) model where k is the maximum context that is specified ahead of execution of the algorithm. The program maintains all the previous occurrences of context at each level of k in a trie-like data structure with associated probability values for each context. If a context at a lower level is a suffix of a context at a higher level, this context is excluded at the lower level. At each level (except the level with k = -1), an *escape character* is defined whose frequency of occurrence is assumed to be equal to the number of distinct context encountered at that context level for the purpose of calculating its probability. During the encoding process, the algorithm estimates the probability of the occurrence of the *next character* in the text stream as follows: the algorithm tries to find the current context of maximum length k in the

context table or trie. If the context is not found, it passes the probability of the escape character at this level and goes down one level to *k-1* context table to find the current context of length *k-1*. If it continues to fail to find the context, it may go down ultimately to *k=-1* level corresponding to equiprobable level for which the probability of any next character is $1/|A|$. If a context of length q, $0<=q<=k$, is found, then the probability of the next character is estimated to be the product of probabilities of escape characters at levels $k, k-1, \ldots, q+1$ multiplied by the probability of the context found at the qth level. This probability value is then passed to the backend entropy coder (arithmetic coder) to obtain the encoding. Note, at the beginning there is no context available so the algorithm assumes a model with $k = -1$. The context lengths are shorter at the early stage of the encoding when only a few context have been seen. As the encoding proceeds, longer and longer context become available. In one version of PPM, called PPM*, an arbitrary length context is allowed which should give the optimal minimum entropy. In practice a model with $k = 5$ behaves as good as PPM* [ClTW95]. Although the algorithm performs very well in terms of high compression ratio or low BPC, it is very computation intensive and slow due to the enormous amount of computation that is needed as each character is processed for maintaining the context information and updating their probabilities.

Dynamic Markov Compression (DMC) is another modeling scheme that is equivalent to finite context model but uses finite state machine to estimate the probabilities of the input symbols which are bits rather than bytes as in PPM [CoH87]. The model starts with a single state machine with only one count of '0' and '1' transitions into itself (the zero frequency state) and then the machine adopts to future inputs by accumulating the transitions with 0's and 1's with revised estimates of probabilities. If a state is used heavily for input transitions (caused either by 1 or 0 input), it is *cloned* into two states by introducing a new state in which some of the transitions are directed and duplicating the output transitions from the original states for the cloned state in the same ratio of 0 and 1 transitions as the original state. The bit-wise encoding takes longer time and therefore DMC is very slow but the implementation is much simpler than PPM and it has been shown that the PPM and DMC models are equivalent [BeM89].

3.2 Dictionary Methods

The dictionary methods, as the name implies, maintain a *dictionary or codebook* of words or text strings previously encountered in the text input and data compression is achieved by replacing strings in the text by a reference to the string in the dictionary. The dictionary is *dynamic or adaptive* in the sense that it is constructed by adding new strings being read and it allows deletion of less frequently used strings if the size of the dictionary exceeds some limit. It is also possible to use a *static* dictionary like the word dictionary to compress the text. The most widely used compression algorithms (Gzip and Gif) are based on Ziv-Lempel or LZ77 coding [ZiL77] in which the text prior to the current symbol constitute the dictionary and a greedy search is initiated to determine whether the characters following the current character have already been encountered in the text before, and if yes, they are replaced by a reference giving its relative starting position in the text. Because of the pattern matching operation the encoding takes longer time but the process has been fine tuned with the use of hashing techniques and special

data structures. The decoding process is straightforward and fast because it involves a random access of an array to retrieve the character string. A variation of the LZ77 theme, called the LZ78 coding, includes one extra character to a previously coded string in the encoding scheme. A more popular variant of LZ78 family is the so-called LZW algorithm which lead to widely used *compress* algorithm. This method uses a suffix tree to store the strings previously encountered and the text is encoded as a sequence of node numbers in this tree. To encode a string the algorithm will traverse the existing tree as far as possible and a new node is created when the last character in the string fails to traverse a path any more. At this point the last encountered node number is used to compress the string up to that node and a new node is created appending the character that did not lead to a valid path to traverse. In other words, at every step of the process the length of the recognizable strings in the dictionary gets incrementally stretched and are made available to future steps. Many other variants of LZ77 and LZ78 compression family have been reported in the literature (See Sal00 and Say00 for further references).

3.3 Transform Based Methods: The Burrows-Wheeler Transform (BWT)

The word 'transform' has been used to describe this method because the text undergoes a transformation which performs a permutation of the characters in the text so that characters having similar lexical context will cluster together in the output. Given the text input, the forward Burrouws-Wheeler transform [BuWh94] forms all cyclic rotations of the characters in the text in the form of a matrix M whose rows are lexicographically sorted (with a specified ordering of the symbols in the alphabet). The last column L of this sorted matrix and an index r of the row where the original text appears in this matrix is the output of the transform. The text could be divided into blocks or the entire text could be considered as one block. The transformation is applied to individual blocks separately, and for this reason the method is referred to as *block sorting* transform [Fenw96]. The repetition of the same character in the block might slow down the sorting process; to avoid this, a run-length encoding (RLE) step could be preceded before the transform step. The Bzip2 compression algorithm based on BWT transform uses this step and other steps as follows: the output of the BWT transform stage then undergoes a final transformation using either move-to-front (MTF) [BSTW86] encding or distance coding (DC) [Arna00] which exploits the clustering of characters in the BWT output to generate a sequence of numbers dominated by small values (viz. 0,1 or 2) out of possible maximum value of $| A |$. This sequence of numbers is then sent to an entropy coder (Huffman or Arithmetic) to obtain the final compressed form. The inverse operation of recovering the original text from the compressed output proceeds by decoding the inverse of the entropy decoder, then inverse of MTF or DC and then an inverse of BWT. The inverse of BWT obtains the original text given (L , r). This is done easily by noting that the first column of M, denoted as F, is simply a sorted version of L. Define an index vector Tr of size $|L|$ such that $Tr[j]=i$ if and only if both $L[j]$ and $F[i]$ denote the kth occurrence of a symbol from A. Since the rows of M are cyclic rotations of the text, the elements of L precedes the respective elements of F in the text. Thus $F[Tr[j]]$ cyclically precedes $L[j]$ in the text which leads to a simple algorithm to reconstruct the original text.

3.4 Comparison of Performance of Compression Algorithms

An excellent discussion of performance comparison of the important compression algorithms can be found in [WiMB99]. In general, the performance of compression methods depends on the type of data being compressed and there is a tradeoff between compression performance, speed and the use of additional memory resources. The authors report the following results with respect to the Canterbury corpus: In order of increasing compression performance (decreasing BPC), the algorithms can be listed as order zero arithmetic, order zero Huffman giving over 4 BPC; the LZ family of algorithms come next whose performance range from 4 BPC to around 2.5 BPC (gzip) depending on whether the algorithm is tuned for compression or speed. Order zero word based Huffman (2.95 BPC) is a good contender for this group in terms of compression performance but it is two to three times slower in speed and needs a word dictionary to be shared between the compressor and decompressor. The best performing compression algorithms are bzip2 (based on BWT), DMC and PPM all giving BPC ranging from 2.4 to 2.1 BPC. PPM is theoretically the best but is extremely slow as is DMC, bzip2 strikes a middle ground, it gives better than gzip but is not an on-line algorithm because it needs the entire text or blocks of text in memory to perform the BWT transform. LZ77 methods (gzip) are fastest for decompression, then LZ78 technique, then Huffman coders, and the methods using arithmetic coding are the slowest. Huffman coding is better for static applications whereas arithmetic coding is preferable in adaptive and online coding. Bzip2 decodes faster than most of other methods and it achieves good compression as well. A lot of new research on bzip2 (see Section 4) has been carried on recently to push the performance envelope of bzip2 both in terms of compression ratio and speed and as a result bzip2 has become a strong contender to replace the popularity of gzip and compress.

New research is going on to improve the compression performance of many of the algorithms. However, these efforts seem to have come to a point of saturation regarding lowering the compression ratio. To get a significant further improvement in compression, other means like transforming the text before actual compression and use of grammatical and semantic information to improve prediction models should be looked into. Shannon made some experiments with native speakers of English language and estimated that the English language has entropy of around 1.3BPC [Sh51]. Thus, it seems that lossless text compression research is now confronted with the challenge of bridging a gap of about 0.8 BPC in terms of compression ratio. Of course, combining compression performance with other performance metric like speed, memory overhead and on-line capabilities seem to pose even a bigger challenge.

4. Transform Based Methods: Star(*), LIPT , LIT and NIT transforms

In this section we present our research on new transformation techniques that can be used as preprocessing steps for the compression algorithms described in the previous section. The basic idea is to transform the text into some intermediate form, which can be compressed with better efficiency. The transformation is designed to exploit the natural redundancy of the language. We have developed a class of such transformations, each

giving better compression performance over the previous ones and most of them giving better compression over current and classical compression algorithms discussed in the previous section. We first present a brief description of the first transform called Star Transform (also denoted by *-encoding). We then present four new transforms called LIPT, ILPT, LIT and NIT, which produce better results.

The algorithms use a fixed amount of storage overhead in the form of a word dictionary for the particular corpus of interest and must be shared by the sender and receiver of the compressed files. Word based Huffman method also make use of a static word dictionary but there are important differences as we will explain later. Because of this similarity, we specifically compare the performance of our preprocessing techniques with that of the word-based Huffman. Typical size of dictionary for the English language is about 0.5 MB and can be downloaded along with application programs. If the compression algorithms are going to be used over and over again, which is true in all practical applications, the amortized storage overhead for the dictionary is negligibly small. We will present experimental results measuring the performance (compression ratio, compression times, decompression times and memory overhead) of our proposed preprocessing techniques using three corpuses: Calgary, Canterbury and Gutenberg corpus. Finally, we present an information theory based explanation of the performance of our algorithms.

4.1 Star (*) Transformation

The basic idea underlying the star transformations is to define a unique signature of a word by replacing letters in a word by a special placeholder character (*) and keeping a minimum number of characters to identify the word uniquely [FrMu96]. For an English language dictionary D of size 60,000 words, we observed that we needed at most two characters of the original words to keep their identity intact. In fact, it is not necessary to keep any letters of the original word as long as an unique representation can be defined. The dictionary is divided into sub-dictionaries D_s containing words of length, $1<=s<=22$, because the maximum length of a word in English dictionary is 22 and there are two words of length 1 viz ‘a’ and ‘I’ (See Figure 1).

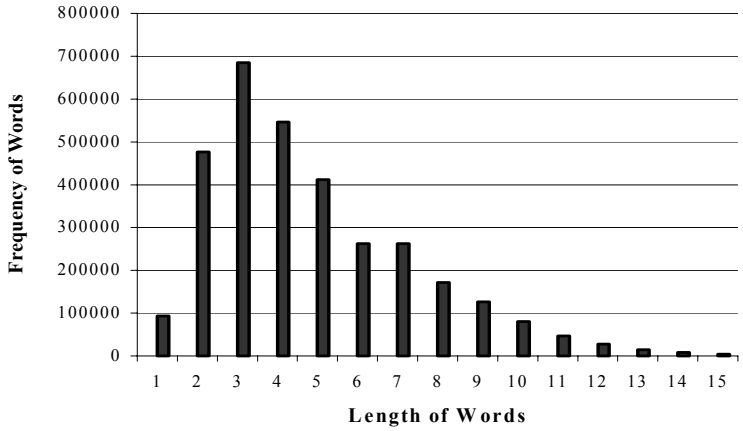


Figure 1: Frequency of English words versus length of words in the test corpus

The following encoding scheme is used for the words in D_s : The first word is represented as sequence of *s* stars. The next 52 words are represented by a sequence of *s-1* stars followed by a single letter from the alphabet $\sum=(a,b, \ldots,z,A,B,\ldots Z)$. The next 52 words have a similar encoding except that the single letter appears in the last but one position. This will continue until all the letters occupy the first position in the sequence. The following group of words have *s-2* *'s and the remaining two positions are taken by unique pairs of letters from the alphabet. This process can be continued to obtain a total of 53^s unique encodings which is more than sufficient for English words. A large fraction of these combinations are never used; for example for $s = 2$, there are only 17 words and for s=8, there are about 9000 words in English dictionary. As an example of star encoding the following sentence: '*Our philosophy of compression is to transform the text into some intermediate form which can be compressed with better efficiency and which exploits the natural redundancy of the language in making this transformation'* can be *-encoded as **a ********** ** ********* *a *b ******** *** ***a ***c ***b ********** ***d ***** **b *c **********a ***e ****** *********a **c ***** ******* *** ******a *********b ** *** ******** *d *****a **** **********

There are exactly five two letter words in the sentence (of, is, to, be, in) which can be uniquely encoded as (**, *a, *b, *c, *d) and similarly for the other groups of words. Given such an encoding, the original word can be retrieved from the dictionary that contains a one-to-one mapping between encoded words and original words. The encoding produces an abundance of * characters in the transformed text making it the most frequently occurring character. If the word in the input text is not in the English dictionary (viz. a new word in the lexicon) it will be passed to the transformed text unaltered. The transformed text must also be able to handle special characters, punctuation marks and capitalization. The space character is used as word separator. The character '~' at the end of an encoded word denotes that the first letter of the input text word is capitalized. The character '`' denotes that all the characters in the input word are capitalized. A capitalization mask, preceded by the character '^', is placed at the end of encoded word to denote capitalization of characters other than the first letter and all capital letters. The character '\' is used as escape character for encoding the occurrences of '*','~','`','^', and '\' in the input text. The transformed text can now be the input to any available losslesss text compression algorithm, including Bzip2 where the text undergoes two transformation, first the *-transform and then a BWT transform.

4.2 LIPT:Length-Index Preserving Transform

A different twist to our transformation comes from the observation that the frequency of occurrence of words in the corpus as well as the predominance of certain lengths of words in English language might play an important role in revealing additional redundancy to be exploited by the backend algorithm. The frequency of occurrence of symbols, k-grams and words in the form of probability models, of course, forms the corner stone of all compression algorithms but none of these algorithms considered the distribution of the length of words directly in the models. We were motivated to consider length of words as an important factor in English text as we gathered word frequency data according to lengths for the Calgary, Canterbury [Cant00], and Gutenberg Corpus [Gute71]. A plot showing the total word frequency versus the word length results for all the text files in our test corpus (combined) is shown in Figure 1. It can be seen that most

words lie in the range of length 1 to 10. The maximum number words have length 2 to 4. The word length and word frequency results provided a basis to build context in the transformed text. We call this Length Index Preserving Transform (LIPT). LIPT can be regarded as the first step of a multi-step compression algorithm such as Bzip2 which includes run length encoding, BWT, move to front encoding, and Huffman coding. LIPT can be used as an additional component in the Bzip2 before run length encoding or simply replace it. Compared to the *-transform, we also made a couple of modifications to improve the timing performance of LIPT. For *-tranform, searching for a transformed word for a given word in the dictionary during compression and doing the reverse during decompression takes time which degrades the execution times. The situation can be improved by pre-sorting the words lexicographically and doing a binary search on the sorted dictionary both during compression and decompression stages. The other new idea that we introduce is to be able to access the words during decompression phase in a random access manner so as to obtain fast decoding. This is achieved by generating the address of the words in the dictionary by using, not numbers, but the letters of the alphabet. We need a maximum of three letters to denote an address and these letters introduce artificial but useful context for the backend algorithms to further exploit the redundancy in the intermediate transformed form of the text. LIPT encoding scheme makes use of recurrence of same length of words in the English language to create context in the transformed text that the entropy coders can exploit.

A dictionary D of words in the corpus is partitioned into disjoint dictionaries D_i, each containing words of length i, where $i = 1,2\ldots n$. Each dictionary D_i is partially sorted according to the frequency of words in the corpus. Then a mapping is used to generate the encoding for all words in each dictionary D_i. $D_i[j]$ denotes the j^{th} word in the dictionary D_i. In LIPT, the word $D_i[j]$, in the dictionary D is transfored as * $c_{len}[c][c][c]$ (the square brackets denote the optional occurrence of a letter of the alphabet enclosed and are not part of the transformed representation) in the transform dictionary D_{LIPT} where c_{len} stands for a letter in the alphabet [a-z, A-Z] each denoting a corresponding length [1-26, 27-52] and each c is in [a-z, A-Z]. If $j = 0$ then the encoding is $*c_{len}$. For $j>0$, the encoding is * c_{len} $c[c][c]$. Thus, for $1 \le j \le 52$ the encoding is $*c_{len}c$; for $53 \le j \le 2756$ it is $*c_{len}cc$, and for $2757 \le j \le 140608$ it is $*c_{len}ccc$. Thus, the 0^{th} word of length 10 in the dictionary D will be encoded as "*j" in D_{LIPT}, $D_{10}[1]$ as "*ja", $D_{10}[27]$ as "*jA", $D_{10}[53]$ as "*jaa", $D_{10}[79]$ as "*jaA", $D_{10}[105]$ as "*jba" , $D_{10}[2757]$ as "*jaaa", $D_{10}[2809]$ as "*jaba", and so on.

The transform must also be able to handle special characters, punctuation marks and capitalization. The character '*' is used to denote the beginning of an encoded word. The handling of capitalization and special characters are same as in *-encoding. Our scheme allows for a total of 140608 encodings for each word length. Since the maximum length of English words is around 22 and the maximum number of words in any D_i in our English dictionary is less than 10,000, our scheme covers all English words in our dictionary and leaves enough room for future expansion. If the word in the input text is not in the English dictionary (viz. a new word in the lexicon) it will be passed to the transformed text unaltered.

The decoding steps are as follows: The received encoded text is first decoded using the same backend compressor used at the sending end and the transformed LIPT text is recovered. The words with '*' represent transformed words and those without '*' represent non-transformed words and do not need any reverse transformation. The length character in the transformed words gives the length block and the next three characters give the offset in the respective block. The words are looked up in the original dictionary *D* in the respective length block and at the respective position in that block as given by the offset characters. The transformed words are replaced with the respective words from dictionary *D*. The capitalization mask is then applied.

4.3 Experimental Results

The performance of LIPT is measured using Bzip2 –9 [BuWh94; Chap00; Lar98; Sewa00], PPMD (order 5) [Moff90; ClTW95; Sal00] and Gzip -9 [Sal00; WiMB99] as the backend algorithms in terms of average BPC (bits per character). Note these results include some amount of pre-compression because the size of the LIPT text is smaller than the size of the original text file. By average BPC we mean the un-weighted average (simply taking the average of the BPC of all files) over the entire text corpus. The test corpus is shown in Table 1. Note that all the files given in Table 1 are text files extracted from the corpuses.

File Names	Actual Sizes
Calgary	
Bib	111261
book1	768771
book2	610856
News	377109
paper1	53161
paper2	82199
paper3	46526
paper4	13286
paper5	11954
paper6	38105
Progc	39611
Progl	71646
Progp	49379
Trans	93695

File Names	Actual Sizes
Canterbury	
alice29.txt	152089
asyoulik.txt	125179
cp.html	24603
fields.c	11150
grammar.lsp	3721
lcet10.txt	426754
plrabn12.txt	481861
xargs.1	4227
bible.txt	4047392
kjv.Gutenberg	4846137
world192.txt	2473400
Project Gutenberg	
1musk10.txt	1344739
anne11.txt	586960
world95.txt	2988578

Table 1: Text files used in our tests (a) Table showing text files and their sizes from Calgary Corpus (b) Table showing text files and their sizes from Canterbury Corpus and Project Gutenberg

We used SunOS Ultra-5 to run all our programs and to obtain results. LIPT achieves a sort of pre-compression for all the text files. We are using a 60,000 words English dictionary that takes 557,537 bytes. The LIPT dictionary takes only 330,636 bytes compared to *-encoded dictionary which takes as much storage as that of the original dictionary. Figure 2 shows the comparison of actual file sizes and file sizes obtained after applying LIPT and also after *-Encoding, for some of the text files extracted from Calgary, Canterbury, and Project Gutenberg. From Figure 2 it can be seen that LIPT

achieves a bit of compression in addition to preprocessing the text before application to any compressor.

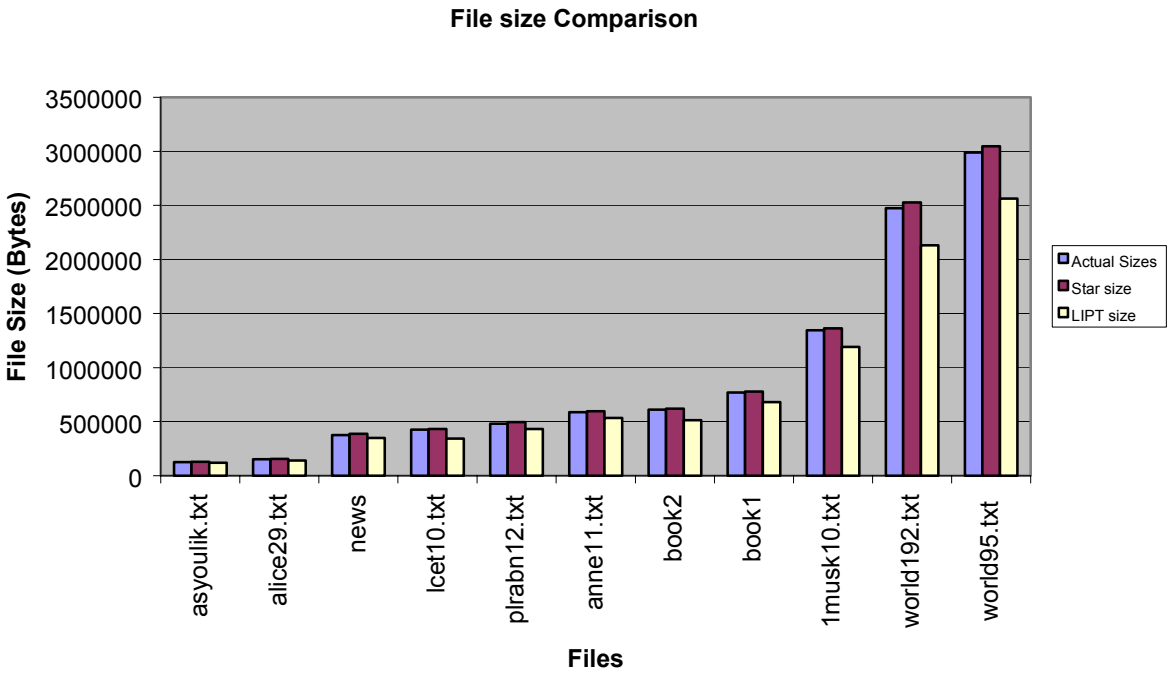


Figure 2: Chart showing the comparison between actual file sizes, Star-Encoding, and the LIPT file sizes for a few of the text files extracted from our test corpus

The results can be summarized as follows: The average BPC using original Bzip2 is 2.28, and using Bzip2 with LIPT gives average BPC of 2.16, a 5.24% improvement (Table 2). The average BPC using original PPMD (order 5) is 2.14, and using PPMD with LIPT gives average BPC of 2.04, and overall improvement of 4.46% (Table 3). The average BPC using original Gzip-9 is 2.71, and using Gzip-9 with LIPT the average BPC is 2.52, a 6.78% improvement (Table 4). The files in Tables 2,3 and 4 are listed in ascending order of file size. Note that for normal text files, the BPC decreases as the file size increases. This can clearly be seen from the Tables especially part (c) of every table that has three text files from Project Gutenberg.

(a)

FileNames	Bzip2 (BPC)	Bzip2 with LIPT (BPC)
Calgary		
paper5	3.24	**2.95**
paper4	3.12	**2.74**
paper6	2.58	**2.40**
Progc	2.53	**2.44**
paper3	2.72	**2.45**
Progp	1.74	**1.72**
paper1	2.49	**2.33**
Progl	1.74	**1.66**
paper2	2.44	**2.26**
Trans	1.53	**1.47**
Bib	1.97	**1.93**
News	2.52	**2.45**
Book2	2.06	**1.99**
Book1	2.42	**2.31**
Average BPC	2.36	**2.22**

(b)

FileNames	Bzip2 (BPC)	Bzip2 with LIPT (BPC)
Canterbury		
grammar.lsp	2.76	**2.58**
xargs.1	3.33	**3.10**
fields.c	2.18	**2.14**
cp.html	2.48	**2.44**
asyoulik.txt	2.53	**2.42**
alice29.txt	2.27	**2.13**
lcet10.txt	2.02	**1.91**
plrabn12.txt	2.42	**2.33**
world192.txt	1.58	**1.52**
bible.txt	1.67	**1.62**
kjv.gutenberg	1.66	**1.62**
Average BPC	2.26	**2.17**

(c)

FileNames	Bzip2 (BPC)	Bzip2 with LIPT (BPC)
Gutenberg		
Anne11.txt	2.22	**2.12**
1musk10.txt	2.08	**1.98**
World95.txt	1.54	**1.49**
Average BPC	1.95	**1.86**

Table 2: Tables a – c show BPC comparison between original Bzip2 –9, and Bzip2 –9 with LIPT for the files in three corpuses

(a)

FileNames	PPMD (BPC)	PPMD with LIPT
Calgary		
paper5	2.98	**2.74**
paper4	2.89	**2.57**
paper6	2.41	**2.29**
Progc	2.36	**2.30**
paper3	2.58	**2.37**
Progp	1.70	**1.68**
paper1	2.33	**2.21**
Progl	1.68	**1.61**
paper2	2.32	**2.17**
Trans	1.47	**1.41**
Bib	1.86	**1.83**
news	2.35	**2.31**
book2	1.96	**1.91**
book1	2.30	**2.23**
Average BPC	2.23	**2.12**

(b)

FileNames	PPMD (BPC)	PPMD with LIPT (BPC)
Canterbury		
grammar.lsp	2.36	**2.21**
xargs.1	2.94	**2.73**
fields.c	2.04	**1.97**
cp.html	2.26	**2.22**
asyoulik.txt	2.47	**2.35**
alice29.txt	2.18	**2.06**
lcet10.txt	1.93	**1.86**
plrabn12.txt	2.32	**2.27**
world192.txt	1.49	**1.45**
bible.txt	1.60	**1.57**
kjv.gutenberg	1.57	**1.55**
Average BPC	2.11	**2.02**

(c)

FileNames	PPMD (BPC)	PPMD with LIPT
Gutenberg		
Anne11.txt	2.13	**2.04**
1musk10.txt	1.91	**1.85**
World95.txt	1.48	**1.45**
Average BPC	1.84	**1.78**

Table 3: Tables a – c show BPC comparison between original PPMD (order 5), and PPMD (order 5) with LIPT for the files in three corpuses.

(a)

FileNames	Gzip (BPC)	Gzip with LIPT (BPC)
Calgary		
paper5	3.34	**3.05**
paper4	3.33	**2.95**
paper6	2.77	**2.61**
Progc	2.68	**2.61**
paper3	3.11	**2.76**
Progp	1.81	**1.81**
paper1	2.79	**2.57**
Progl	1.80	**1.74**
paper2	2.89	**2.62**
Trans	1.61	**1.56**
bib	2.51	**2.41**
news	3.06	**2.93**
book2	2.70	**2.48**
book1	3.25	**2.96**
Average BPC	2.69	**2.51**

(b)

FileNames	Gzip (BPC)	Gzip with LIPT (BPC)
Canterbury		
grammar.lsp	2.68	**2.61**
xargs.1	3.32	**3.13**
fields.c	2.25	**2.21**
Cp.html	2.60	**2.55**
asyoulik.txt	3.12	**2.89**
alice29.txt	2.85	**2.60**
lcet10.txt	2.71	**2.42**
plrabn12.txt	3.23	**2.96**
world192.txt	2.33	**2.18**
bible.txt	2.33	**2.18**
kjv.gutenberg	2.34	**2.19**
Average BPC	2.70	**2.54**

(c)

FileNames	Gzip (BPC)	Gzip with LIPT (BPC)
Gutenberg		
Anne11.txt	3.02	**2.75**
1musk10.txt	2.91	**2.62**
World95.txt	2.31	**2.15**
Average BPC	2.75	**2.51**

Table 4: Tables a – c show BPC comparison between original Gzip -9, and Gzip –9 with LIPT for the files in three corpuses.

Figure 3 gives a bar chart comparing the BPC of the original Bzip2,PPMD and the compressors in conjunction with LIPT for a few text files extracted from our test corpus. From Figure 3 it can be seen that Bzip2 with LIPT (second bar in Figure 3) is close to the original PPMD (third bar in Figure 3) in bits per character (BPC).

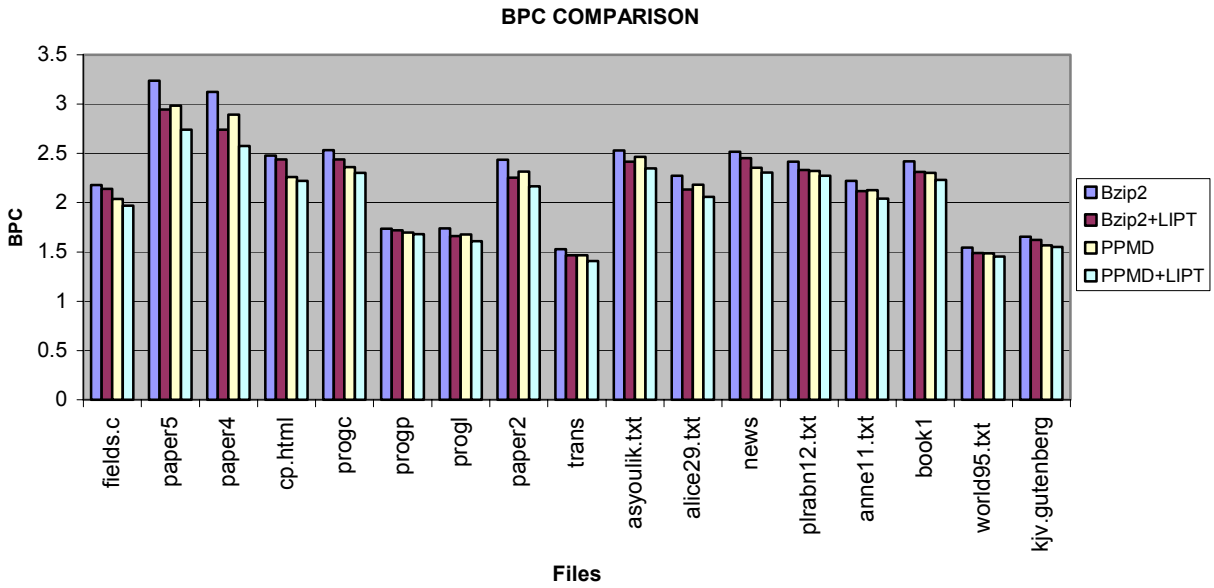


Figure 3: Bar Chart giving comparison of Bzip2, Bzip2 with LIPT, PPMD, and PPPMD with LIPT

From Figure 3 it can also be seen that in instances like paper5, paper4, progl, paper2, asyoulik.txt, and alice29.txt, Bzip2 with LIPT is beating the original PPMD in terms of BPC. The difference between average BPC for Bzip2 with LIPT (2.16) and original PPMD (2.1384) is only around 0.02 bits i.e. average BPC for Bzip2 with LIPT is only around 1% more than the original PPMD. This observation is important as it contributes towards the efforts being made by different researchers to obtain PPMD BPC performance with a faster compressor. It is shown later on timing results that Bzip2 with LIPT is much faster than the original PPMD. (Note that although Bzip2 with LIPT gives lower BPC than the original Bzip2, the former is much slower than the later as discussed in later in this report). Table 5 gives a summary comparison of BPC for the original Bzip2 –9, PPMD (order 5), Gzip –9, Huffman (character based), word-based Arithmetic coding, and these compressors with Star-Encoding and LIPT. The data in Table 5 shows that LIPT performs much better over Star-Encoding and original algorithms except for character based Huffman and Gzip –9.

	Original (BPC)	*-encoded (BPC)	LIPT (BPC)
Huffman (character based)	4.87	**4.12**	4.49
Arithmetic (word based)	2.71	2.90	**2.61**
Gzip-9	2.70	**2.52**	**2.52**
Bzip2	2.28	2.24	**2.16**
PPMD	2.13	2.13	**2.04**

Table 5: Summary of BPC Results

Table 5 also shows that Star-encoding gives a better average BPC performance for character-based Huffman, Gzip, and Bzip2 but gives worse average BPC performance for word-based arithmetic coding and PPMD. This is due to the presence of the non-English words and special symbols in the text. For a pure text file, for example the dictionary itself, the star dictionary has a BPC of 1.88 and original BPC is 2.63 for PPMD. The improvement is 28.5% in this case. Although the average BPC for Star-encoding is worse than original PPMD, there are 16 files that show improved BPC, and 12 files show worse BPC. The number of words in the input text that are also found in English dictionary *D* is an important factor for the final compression ratio. For character based Huffman, Star-encoding performs better than the original Huffman and LIPT with Huffman. This is because in Star-encoding there are repeated occurrences of the character '*' which gets the highest frequency in the Huffman code book and is thus encoded with lowest number of bits resulting in better compression results than the original and the LIPT files.

4.4 Comparison with Recent Improvements of BWT and PPM

We focus our attention on LIPT over Bzip2 (which uses BWT), Gzip and PPM algorithms because Bzip2 and PPM outperform other compression methods and Gzip is commercially available and commonly used. Of these, BWT based approach has proved to be the most efficient and a number of efforts have been made to improve its efficiency. The latest efforts include Balkenhol, Kurtz, and Shtarkov [BaKS99], Seward [Sewa00], Chapin [Chap00], and Arnavut [Arna00]. PPM on the other hand gives better compression ratio than BWT but is very slow in execution time. A number of efforts have

been made to reduce the time for PPM and also to improve the compression ratio. Sadakane, Okazaki, and Imai [SaOH00] have given a method where they have combined PPM and CTW [WiST95] to get better compression. Effros [Effo00] has given a new implementation of PPM* with the complexity of BWT. Tables 6 and 7 give a comparison of compression performance of our proposed transform which shows that LIPT has better BPC for most of the files and it has better average BPC than all the other methods cited. Some data in Table 6 and Table 7 have been taken from the references given in the respective columns.

File	MBSWIC [Arna00]	BKS98 [BaKS99]	Best x of $2x - 1$ [Chap00]	Bzip2 with LIPT
Bib	2.05	1.94	1.94	**1.93**
book1	**2.29**	2.33	**2.29**	2.31
book2	2.02	2.00	2.00	**1.99**
news	2.55	2.47	2.48	**2.45**
paper1	2.59	2.44	2.45	**2.33**
paper2	2.49	2.39	2.39	**2.26**
progc	2.68	2.47	2.51	**2.44**
progl	1.86	1.70	1.71	**1.66**
progp	1.85	**1.69**	1.71	1.72
trans	1.63	**1.47**	1.48	**1.47**
Average BPC	2.21	2.105	2.11	**2.07**

Table 6: BPC comparison of approaches based on BWT

File	Multi-alphabet CTW order 16 [SaOH00]	NEW Effros [Effo00]	PPMD (order 5) with LIPT
bib	1.86	1.84	**1.83**
book1	**2.22**	2.39	2.23
book2	1.92	1.97	**1.91**
News	2.36	2.37	**2.31**
paper1	2.33	2.32	**2.21**
paper2	2.27	2.33	**2.17**
Progc	2.38	2.34	**2.30**
Progl	1.66	**1.59**	1.61
Progp	1.64	**1.56**	1.68
Trans	1.43	**1.38**	1.41
Average BPC	2.021	2.026	**1.98**

Table 7: BPC comparison of new approaches based on Prediction Models

4.5 Comparison with Word-based Huffman

Huffman compression method also needs sharing of the same static dictionary at both the sender and receiver end, as does our method. Canonical Huffman [WiMB99] method assigns variable length addresses to words using bits and LIPT assigns variable length

offset in each length block using letters of alphabet. Due to these similarities we compare the word-based Huffman with LIPT (we used Bzip2 as the compressor). Huffman and LIPT both sort the dictionary according to frequency of use of words. Canonical Huffman assigns a variable address to the input word, building a tree of locations of words in the dictionary and assigning 0 or 1 to each branch of the path. LIPT exploits the structural information of the input text by including the length of the word in encoding. LIPT also achieves a pre-compression due to the variable offset scheme. In Huffman, if new words are added, the whole frequency distribution has to be recomputed as well as the Huffman codes for them.

A typical word-based Huffman model is a zero-order word-based semi-static model [WiMB99]. Text is parsed at the first pass of scan to extract zero-order words and non-words as well as their frequency distributions. Words are typically defined as consecutive characters and non-words are typically defined as punctuation, space and control characters. If an unseen word or non-word occurred, normally some escape symbol is transmitted, and then the string is transmitted as sequence of single characters. Some special type of strings can be considered for special representation, for example, the numerical sequences. To handle arbitrarily large sequence of numbers, one way of encoding is to break them in to smaller pieces e.g. groups of four digits. Word-based models can generate a large number of symbols. For example, in our text corpus with the size of 12918882 bytes, there are 70661 words and 5504 non-words. We can not make sure that these may include all or most of the possible words in a huge database since the various words may be generated by the definition of words here. Canonical Huffman code [Sewa00] is selected to encode the words. The main reason for using canonical Huffman code is to provide efficient data structures to deal with huge dictionary generated and for fast decompression so that the retrieval is made faster.

Table 8 shows the BPC comparison. For LIPT, we extract the strings of characters in the text and build the LIPT dictionary for each file. In contrast to the approach given in [WiMB99], we do not include the words composed of digits and mixture of alphabets and digits as well as other special characters. We try to make a fair comparison, however, word-based Huffman still uses a broader definition of “words”. Comparing the average BPC, the Managing Gigabyte [WiMB99] word-based Huffman model has a 2.506 BPC for our test corpus. LIPT with Bzip2 has a BPC of 2.17. The gain is 13.44%. LIPT does not give improvement over word based Huffman for files with mixed text such as source files for programming languages. For files with more English word, LIPT shows consistent gain.

We also have compared LIPT with some of the newer compression reported in the literature and websites such as YBS [YBS00],RK [RK00-1,RK00-2] and PPMonstr [PPMDH00]. The average BPC using LIPT along with these methods is around 2.00BPC which is better than any of these algorithms as well as the original Bzip2 and PPMD. For details, the reader is referred to [AwMu01;Awan01].

File	Filesize	Bzip2 with LIPT	Word-based Huffman	% GAIN
cp.html	21333	2.03	2.497	18.696
paper6	32528	2.506	2.740	8.550
progc	33736	2.508	2.567	2.289
paper1	42199	2.554	2.750	7.134
progp	44862	1.733	2.265	23.497
progl	62367	1.729	2.264	23.618
trans	90985	1.368	1.802	24.090
bib	105945	1.642	2.264	27.477
asyoulik.txt	108140	2.525	2.564	1.527
alice29.txt	132534	2.305	2.333	1.194
lcet10.txt	307026	2.466	2.499	1.314
news	324476	2.582	2.966	12.936
book2	479612	2.415	2.840	14.963
anne11.txt	503408	2.377	2.466	3.597
1musk10.txt	1101083	2.338	2.454	4.726
world192.txt	1884748	1.78	2.600	31.551
world95.txt	2054715	1.83	2.805	34.750
kjv.gutenberg	4116876	1.845	2.275	18.914
AVERAGE		**2.169**	**2.506**	**13.439**

Table 8: BPC comparison for test files using Bzip2 with LIPT, and word-based Huffman

4.6 Timing Performance Measurements

The improved compression performance of our proposed transform comes with a penalty of degraded timing performance. For off-line and archival storage applications, such penalties in timing are quite acceptable if a substantial savings in storage space can be achieved. The increased compression/decompression times are due to frequent access to a dictionary and its transform dictionary. To alleviate the situation, we have developed efficient data structures to expedite access to the dictionaries and memory management techniques using caching. Realizing that certain on-line algorithms might prefer not to use a pre-assigned dictionary, we also have been working on a new family of algorithms, called M5zip to obtain the transforms dynamically with no dictionary and with small dictionaries (7947 words and 10000 words), which will be reported in future papers.

The experiments were carried out on 360MHz Ultra Sparc-IIi Sun Microsystems machine housing SunOS 5.7 Generic_106541-04. In our experiments we compare compression times of Bzip2, Gzip and PPMD against Bzip2 with LIPT, Gzip with LIPT and PPMD with LIPT. During the experiments we have used –9 option for Gzip. This option supports for better compression. Average compression time, for our test corpus, using LIPT with Bzip2 -9, Gzip -9, and PPMD is 1.79 times slower, 3.23 times slower and fractionally (1.01 times) faster compared to original Bzip2, Gzip and PPMD respectively. The corresponding results for decompression times are 2.31 times slower, 6.56 times slower and almost the same compared to original Bzip2, Gzip and PPMD respectively. Compression using Bzip2 with LIPT is 1.92 times faster and decompression is 1.98 times

faster than original PPMD (order 5). The increase in time over standard methods is due to time spent in preprocessing the input file. Gzip uses –9 option to achieve maximum compression therefore we find that the times for compression using Bzip2 are less than Gzip. When maximum compression option is not used, Gzip runs much faster than Bzip2.

Decompression time for methods using LIPT includes decompression using compression techniques plus reverse transformation time. Bzip2 with LIPT decompresses 2.31 times slower than original Bzip2, Gzip with LIPT decompresses 6.56 times slower than Gzip, and PPMD with LIPT is almost the same.

5. Dictionary Organization and Memory Overhead

LIPT uses a static English language dictionary of 59951 words having a size of around 0.5 MB. LIPT uses transform dictionary of around 0.3 MB. . The transformation process requires two files namely English dictionary, which consist of most frequently used words, and a transform dictionary, which contains corresponding transforms for the words in English dictionary. There is one-to-one mapping of word from English to transform dictionary. The words not found in the dictionary are passed as they are. To generate the LIPT dictionary (which is done offline), we need the source English dictionary to be sorted on blocks of lengths and words in each block should be sorted according to frequency of their use. On the other hand we need a different organization of dictionary for encoding and decoding procedures (which are done online) in order to achieve efficient timing. We use binary search which on average needs log$w$ comparisons where w is the number of words in the English dictionary D. To use binary search, we need to sort the dictionary lexicographically. We sort the blocks once on loading the dictionary into memory using Quicksort. For successive searching the access time is $M \times \log w$, where M is number of words in the input file and w is number of words in dictionary. The total number of comparison is $w \times \log w + M \times \log w$. In secondary storage, our dictionary structure is based on first level blocking according length and then within each block we sort the words according to their frequency of use. In memory, we organize the dictionary into two levels. In Level 1, we classify the words in dictionary based on the length of the word and sort these blocks in ascending order of frequency of use. Then in level 2, we sort the words within each block of length lexicographically. This sorting is done once upon loading of dictionaries into the memory. It is subjected to resorting only when there is modification to the dictionary like adding or deleting words from dictionary. In order to search a word of length l and starting character as z, the search domain is only confined to a small block of words which have length l and start with z. It is necessary to maintain a version system for different versions of the English dictionaries being used. A simple method works well with our existing dictionary system. When new words are added they are added at the end of the respective word length blocks. Adding words at the end has two advantages: previous dictionary word-transform mapping is preserved scalability without distortion is maintained in the dictionary.

It is important to note that the dictionary is installed with the executable and is not transmitted every time with the encoded files. The only other time it is transmitted is when there is an update or new version release. LIPT encoding needs to load original

English dictionary (currently 55K bytes) and LIPT dictionary D_{LIPT} (currently 33K). There is an additional overhead of 1.5 K for the two level index tables we are using in our dictionary organization in memory. So currently, in total, LIPT incurs about 89K bytes. Bzip2 is claimed to use 400K+(7 × Block size) for compression [Bzip2]. We use "–9 option" for Bzip2 and Bzip2 –9 uses 900K of block size for the test. Therefore, in total we need about 6700K for Bzip2. For decompression, it takes around 4600K and 2305K with –s option [Bzip2]. For PPMD it takes as about 5100K + file size (this is the size we fixed in the source code for PPMD). So LIPT takes only a small additrional overhead compared to Bzip2 and PPM in memory usage.

6. Three New Transforms – ILPT, NIT and LIT

We will briefly describe our three new lossless reversible text transforms based on LIPT. We give experimental results for the new transforms and discuss them briefly. For details on these transformations and experimental results the reader is referred to [Awan01]. Note that there is no significant effect on the time performance as the dictionary loading method remains the same and the number of words also remain the same in the static English dictionary *D* and transform dictionaries. Hence we will only give the BPC results obtained with different approaches for the corpus.

Initial Letter Preserving transform (ILPT) is similar to LIPT except that we sort the dictionary into blocks based on the lexicographic order of starting letter of the words. We sort the words in each letter block according to descending order of frequency of use. The character denoting length in LIPT (character after '*') is replaced by the starting letter of the input word i.e. instead of $*c_{len}[c][c][c]$, for ILPT it becomes $*c_{init}[c][c][c]$ where c_{init} denotes the initial (staring) letter of the word. Everything else is handled the same way as in LIPT. Bzip2 with ILPT has an average BPC of 2.12 for all the files in all the corpuses combined. This means that Bzip2 –9 with ILPT shows 6.83% improvement over the original Bzip2 –9. Bzip2 –9 with ILPT shows 1.68 % improvement over Bzip2 with LIPT.

Number Index Transform (NIT) scheme uses variable addresses based on letters of alphabet instead of numbers. We wanted to compare this using a simple linear addressing scheme with numbers i.e. giving addresses 0 - 59950 to 59951 words in our dictionary. Using this scheme on our English dictionary *D*, sorted according to length of words and then sorted according to frequency within each length block, gave deteriorated performance compared to LIPT. So we sorted the dictionary globally according to descending order of word usage frequency. No blocking was used in the new frequency sorted dictionary. The transformed words are still denoted by staring character '*'. The first word in the dictionary is encoded as "*0", the 1000th word is encoded as "*999", and so on. Special character handling is same as in LIPT. We compare the BPC results for Bzip2 –9 with the new transform NIT with Bzip2 –9 with LIPT. Bzip2 with NIT has an average BPC of 2.12 for all the files in all the corpuses combined. This means that Bzip2 –9 with NIT shows 6.83% improvement over the original Bzip2 –9. Bzip2 –9 with NIT shows 1.68 % improvement over Bzip2 with LIPT.

Combining the approach taken in NIT and the using letters to denote offset , we arrive at another transform which is same as NIT except that now we use letters of the alphabet [a-zA-Z] to denote the index or the linear address of the words in the dictionary, instead of numbers. This scheme is called *Letter Index Transform (LIT)*. Bzip2 with LIT has an average BPC of 2.11 for all the files in all corpuses combined. This means that Bzip2 –9 with LIT shows 7.47% improvement over the original Bzip2 –9. Bzip2 –9 with LIT shows 2.36 % improvement over Bzip2 with LIPT.

The transform dictionary sizes vary with the transform. The original dictionary takes 557537 bytes. The transformed dictionaries for LIPT, ILPT, NIT and LIT are 330636, 311927, 408547 and 296947 bytes, respectively. Note that LIT has the smallest size dictionary and shows uniform compression improvement over other transforms for almost all the files.

Because of its better performance compared to the performance of other transforms, we have compared LIT with PPMD, YBS, RK and PPMonstr. PPMD (order 5) with LIT has an average BPC of 1.99 for all the files in all corpuses combined. This means that PPMD (order 5) with LIT shows 6.88% improvement over the original PPMD (order 5). PPMD with LIT shows 2.53 % improvement over PPMD with LIPT. RK with LIT has average BPC lower than RK with LIPT. LIT performs well with YBS, RK, and PPMonstr giving 7.47%, 5.84%, and 7.0% improvement over the original methods, respectively. It is also important to note that LIT with YBS outperforms Bzip2 by 10.7% and LIT with PPPMonstr outperforms PPMD by 10%.

7. Theoretical Explanation

In this section we give theoretical explanation for the compression performance based on entropy calculations. We give mathematical equations to giving factors that affects compression performance. We provide experimental results in support of our proposed theoretical explanation..

7.1 Qualitative Explanation for Better Compression with LIPT

LIPT introduces frequent occurrences of common characters for BWT and good context for PPM as well as it compresses the original text. Cleary, Teahan, and Witten [CLTW95], and Larsson [Lar98] have discussed the similarity between PPM and Bzip2. PPM uses a probabilistic model based on the context depth and uses the context information explicitly. On the other hand the frequency of similar patterns and local context affect the performance of BWT implicitly. Fenwick [Fenw96] also explains how BWT exploits the structure in the input text. LIPT introduces added structure along with smaller file size leading to better compression after applying Bzip2 or PPMD.
The offset characters in LIPT represent a variable-length encoding similar to Huffman encoding and produce some initial compression of the text but the difference from Huffman encoding is significant. The address of the word in the dictionary is generated at the modeling rather than entropy encoding level and LIPT exploits the distribution of words in English language based on the length of the words as given in Figure 1. The

sequence of letters to denote the address also has some inherent context depending on how many words are in a single group, which also opens another opportunity to be exploited by the backend algorithm at the entropy level.

There are repeated occurrences of words with same length in a usual text file. This factor contributes in introducing good and frequent context and thus higher probability of occurrence of same characters (space, '*', and characters denoting length of words) that enhance the performance of Bzip2 (which uses BWT) and PPM as proved by results given earlier in the report. LIPT generates encoded file, which is smaller in size than the original text file. Because of the small input file along with a set of artificial but well defined deterministic context, both BWT and PPM can exploit the context information very effectively producing a compressed file that is smaller than the file without using LIPT. In order to verify our conjecture that LIPT may produce effective context information based on the frequent word length recurrence in the text, we made some measurement on order statistics of the file entropy with and without using LIPT and calculated the effective BPC over context length up to 5. The results are given in Table 9 with respect to a typical file alice29.txt for the purpose of illustration.

Context Length (Order)	Original (count)	Original BPC-Entropy	LIPT (count)	LIPT BPC-Entropy
5	114279	1.98	108186	2.09
4	18820	2.50	15266	3.12
3	12306	2.85	7550	**2.03**
2	5395	3.36	6762	**2.18**
1	1213	4.10	2489	**3.50**
0	75	6.20	79	6.35
Total	152088		**140332**	

Table 9: Comparison of bits per character (BPC) in each context length used by PPMD, and PPMD with LIPT for the file alice29.txt

The data in Table 9 shows that LIPT uses less number of bits for context order 3, 2, and 1 as compared to the original file. The reason for this can be derived from our discussion on frequency of recurrence of length throughout the input text. For instance in alice29.txt there are 6184 words of length 3 and 5357 words of length 4 (there are words of other lengths as well but here we are taking lengths 3 and 4 to illustrate our point). Out of these, 5445 words of length 3 and 4066 words of length 4 are found in our English dictionary. This means that the sequence, space followed by '*c', will be found 5445 times and the sequence, space followed by '*d', will be found 4066 times in the transformed text for alice29.txt using LIPT. There can be other sequences of offset letters after 'c' or 'd' but at least these three characters (including space) will be found in this sequences this many times. Here these three characters define a context of length three. Similarly there will be other repeated lengths in the text. The sequence space followed by

'*' with two characters defines context of length two that is found in the transformed text very frequently. We can see from Table 9 that the BPC for context length 2 and 3 is much lower than the rest. Apart from the space, '*', and the length character sequence the offset letters also provide added probability for finding similar contexts. LIPT also achieves a pre-compression at the transformation stage (before actual compression). This is because transformation for a word can use at most 6 characters and hence words of length 7 and above are encoded with fewer characters. Also the words with other lengths can be encoded with fewer characters depending on the frequency of their usage (their offset will have fewer characters in this case). Due to sorting of dictionary according to frequency and length blocking we have been able to achieve reduction in size of the original files in the range of 7% to 20%.

7.2 Entropy Based Explanation for Better Compression Performance of Our Transforms

Let n denote the total number of unique symbols (characters) in a message Q and P_i denote the probability of occurrence of each symbol i. Then entropy S of message Q [ShW98] is given by:

$$S = -\sum_{1}^{n} P_i \log_2 P_i \tag{2}$$

Entropy is highest when all the events are equally likely that is when all P_i are $1/n$. The difference between the entropy at this peak (maximum entropy) where probability P is $1/n$ and the actual entropy given by equation (2) is called the redundancy R of the source data.

$$R = -\log_2(1/P) - (-\sum_{1}^{n} P_i \log_2 P_i) = \log_2 n + \sum_{1}^{n} P_i \log_2 P_i \tag{3}$$

This implies that when entropy S is equal to maximum entropy $S_{max} = \log_2 n$ then there is no redundancy and the source text cannot be compressed further. The performance of compression method is measured by the average number of bits it takes to encode a message. Compression performance can also be measured by *compression factor*, which is the ratio of the number of characters in the source message to the number of characters in the compressed message.

7.3 Explanation Based on First Order (Context Level Zero) Entropy

First order entropy is sum of entropies for each symbol in a text. It is based on context level zero i.e. no prediction based on preceding symbols. Compressed file size can be given in terms of redundancy and original file size. We can rewrite Equation (3) as:

$$R = S_{\max} - S \tag{4}$$

R is the total redundancy found in the file. Now compressed file size F_c is given as:

$$F_c = F - \frac{R \times F}{S_{\max}} \tag{5}$$

where F is the size of uncompressed file. Notice that $\frac{R}{S_{\max}}$ gives the per symbol redundancy for the file.

Substituting value of R from Equation (3) into Equation (5) we get:

$$F_c = F \times \frac{S}{S_{\max}} \tag{6}$$

The quantity $S/S_{\max}$ is called the relative entropy and gives a measure of fraction of file that can be compressed. As we are using ASCII character set as our alphabet so S_{max} is 8 for our discussion. Hence equation (6) becomes:

$$F_c = F \times \frac{S}{8} \tag{7}$$

Compression Factor C is given by

$$C = \frac{F}{F_c} \tag{8}$$

Substituting value of F_c from Equation (6) into Equation (8) we get:

$$C = \frac{S_{\max}}{S} \tag{9}$$

This relationship is also verified in Table 10 in compression factor column. Table 10 gives the first-order entropy (Equation (2)) and redundancy (Equation (3)) data for Calgary Corpus. The maximum entropy is 8 bits/symbol as we are using 256 character set as our alphabet. Table 10 also shows theoretical compression factors for respective files based on first order entropy S and compression factors based on experimental results using first-order character based Huffman coder (context level zero). The last two columns of Table 10 show that the theoretical and experimental compression factors are very close and the compression factor is inversely proportional to redundancy based on first order entropy. Higher compression factor means more compression as compression factor is given by the ratio of uncompressed file size to compressed file size. Redundancy shows how much a file can be compressed. Lower entropy means less number of bits to encode the file resulting in better compression. Table 10 verifies all these observations.

File	first order entropy *S* (bits/character)	Redundancy *R* (bits/character)	Compression factor (based on first order entropy *S*)	Compression factor (based on experimental results using Huffman Coding)
Trans	5.53	2.47	1.45	1.44
Progc	5.20	2.80	1.54	1.53
Bib	5.20	2.80	1.54	1.53
News	5.19	2.81	1.54	1.53
Paper6	5.01	2.99	1.60	1.59
Paper1	4.98	3.02	1.61	1.59
Paper5	4.94	3.06	1.62	1.61
Progp	4.87	3.13	1.64	1.63
book2	4.79	3.21	1.67	1.66
Progl	4.77	3.23	1.68	1.67
Paper4	4.70	3.30	1.70	1.69
Paper3	4.67	3.33	1.71	1.71
Paper2	4.60	3.40	1.74	1.73
book1	4.53	3.47	1.77	1.75

Table 10: Entropy and redundancy in relation with compression factor (Calgary Corpus- original files)

Now let F_t represent the size of the file transformed with any one of our transforms, which has the size F before application of the transform. Remember that our transforms produce a reduced file size for the intermediate text by a factor s, $0<s<1$ (for *-transform for which s is slightly greater than 1 implying an expansion). Hence we can write:

$$F_t = F \times s \tag{10}$$

The compressed file size F_c is given by

$$F_c = \frac{F_t \times S_t}{S_{\max}} \tag{11}$$

where S_t is entropy of the transformed file. Intermediate compression factor C_{int} is given by

$$C_{\text{int}} = \frac{F_t}{F_c} \tag{12}$$

Substituting the value of F_c in Equation (8):

$$C = \frac{F}{F_t \times S_t \Big/ S_{\max}} \tag{13}$$

Finally, substituting the value of F_t, we get

$$C = \frac{S_{\max}}{s \times S_t} \tag{14}$$

Equation (14) shows that the compression given by *compression factor C is inversely proportional to the product of file size reduction factor s achieved by a transform, and entropy* S_t. Smaller s and proportionally smaller entropy of the transformed file means higher compression. Equation (14) is same as Equation (9) which is for uncompressed (original) files with s= 1 and $S_t = S$.

The results for the products $s \times S$ for original files and $s \times S_t$ of all our transforms for Calgary corpus are given in Table 11.

File	Original $s \times S$	C	LIPT $s \times S_t$	C	ILPT $s \times S_t$	C	NIT $s \times S_t$	C	LIT $s \times S_t$	C
Trans	5.53	1.44	5.10	1.56	4.39	1.62	4.61	1.61	**4.38**	**1.64**
Bib	5.20	1.53	4.83	1.65	4.63	1.72	4.74	1.68	**4.51**	**1.77**
Progc	5.20	1.53	5.12	1.55	2.20	1.61	2.36	1.52	**2.13**	**1.62**
News	5.19	1.53	4.96	1.60	4.71	1.69	4.93	1.61	**4.60**	**1.73**
Paper6	5.01	1.59	4.58	1.73	4.28	1.85	4.22	1.88	**4.11**	**1.93**
Paper1	4.98	1.59	4.42	1.80	4.11	1.93	4.13	1.92	**3.99**	**1.99**
Paper5	4.94	1.61	4.41	1.80	4.14	1.92	4.24	1.87	**3.98**	**1.99**
Progp	4.87	1.63	4.93	1.61	4.67	1.65	4.94	1.53	**4.64**	**1.66**
book2	4.79	1.66	4.20	1.89	3.86	2.05	3.91	2.03	**3.71**	**2.14**
Progl	4.77	1.67	4.61	1.72	4.71	1.82	5.09	1.67	**4.69**	**1.83**
Paper4	4.70	1.69	4.00	1.98	3.67	2.16	3.70	2.13	**3.52**	**2.25**
Paper3	4.67	1.71	3.98	1.99	3.62	2.19	3.74	2.12	**3.50**	**2.26**
Paper2	4.60	1.73	4.00	1.98	3.63	2.18	3.70	2.14	**3.48**	**2.28**
book1	4.53	1.75	4.05	1.95	3.66	2.17	3.72	2.13	**3.50**	**2.27**

Table 11: Product $s \times S$ for original and $s \times S_T$ for all transformed files in Calgary Corpus

We also give experimental compression factor C obtained using Huffman Coding (character based). From Table 11 we can see that LIT has the lowest $s \times S_t$ for most of the files and thus will give the best compression performance. This is also depicted in Figure 4 which shows that compression is inversely proportional to product of file size factor s and entropy of the file S_t (Only the graph for LPIT is shown).

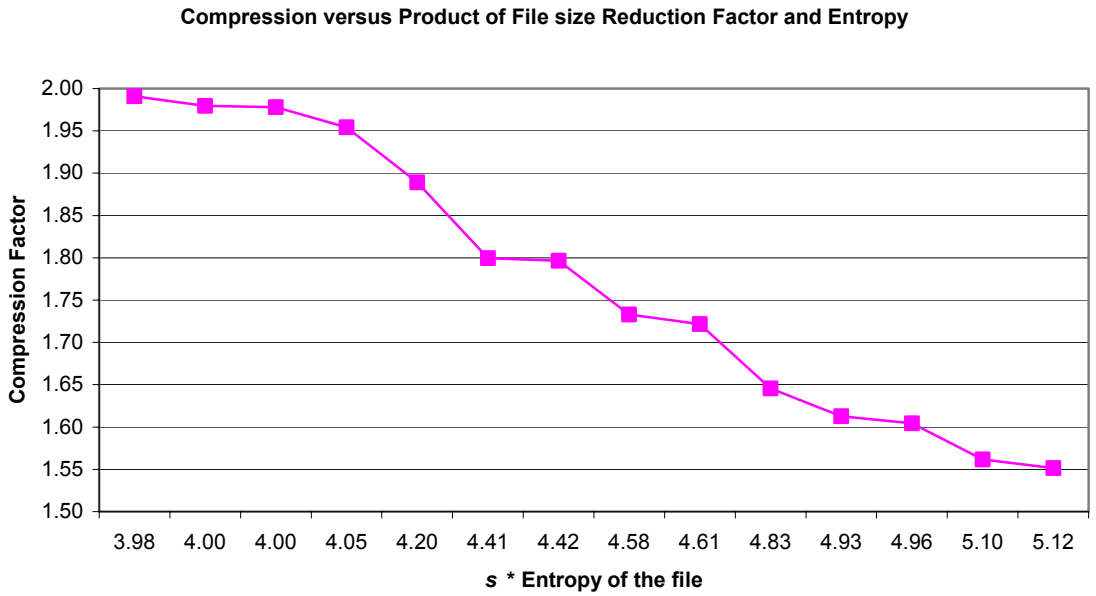


Figure 4: Graph showing Compression versus Product of File size reduction factor s and File Entropy S_t

7.4 Explanation based on higher order context-based entropy

A context-based text compression method uses the context of a symbol to predict it. This context has certain length k called the context order and the method is said to be following order-k Markov Model [ShW98]. PPMD (order 5) used in experiments in this report uses order-5 Markov Model. Burrows Wheeler method is also a context-based method [Fenw96; Lar98] using Markov Model. Burrows Wheeler method is a context-based compression method but performs a transformation on the input data and then applies a statistical model.

If we define each context level c_i as a finite state of Markov model then the average entropy S_c of a context-based method is given by:

$$S_c = \sum_{i=0}^{k} P(c_i) S(c_i) \qquad (15)$$

where i is the context level, k is the length of maximum context (context order), $P(c_i)$ is the probability of the model to be in context level c_i , and $S(c_i)$ is the entropy of context level c_i . In practice, this average entropy S_c can also be calculated by totaling the bits taken in each context level and dividing it by the total count of characters encoded at that level. Now based on Equation (15), let us give the entropies for a few files in Calgary Corpus. Table 12 gives the comparison of the product of file size reduction s and average context-based entropy S_c for Bzip2, Bzip2 compression factors C_{bzip2} and PPMD compression factors C_{PPMD} for original, LIPT, and LIT files. The maximum context length is 5 (order 5). These calculations are derived from similar information as given in Table 11 for respective files. The files in Table 12 are listed in ascending order of

compression factors. From the data given in Table 12 it can be seen that compression factor for bzip2, C_{bzip2} and also for PPMD, C_{PPMD} are inversely proportional to $s \times S_c$ .One also notes that LIT has the lowest $s \times S_c$ but highest compression factors. PPMD with LIT has the highest compression factors. Hence lower $s \times S_c$ justifies the better compression performance for LIT. The same argument can be applied to other transforms.

Files	C_{bzip2} (original)	C_{PPMD} (original)	original $s \times S_c$	C_{bzip2} (LIPT)	C_{PPMD} (LIPT)	LIPT $s \times S_c$	C_{bzip2} (LIT)	C_{PPMD} (LIT)	LIT $s \times S_c$
paper6	3.10	3.32	2.41	3.33	3.50	2.28	3.42	**3.62**	**2.21**
progc	3.16	3.39	2.36	3.28	3.48	2.30	3.33	**3.55**	**2.25**
news	3.18	3.40	2.35	3.26	3.47	2.31	3.31	**3.52**	**2.27**
book1	3.31	3.47	2.30	3.46	3.58	2.23	3.49	**3.61**	**2.22**
book2	3.88	4.07	1.96	4.02	4.19	1.91	4.08	**4.26**	**1.88**
bib	4.05	4.30	1.86	4.14	4.37	1.83	4.24	**4.47**	**1.79**
trans	4.61	5.45	1.47	5.46	5.69	1.41	5.55	**5.84**	**1.37**

Table 12: Relationship between Compression Factors (using Bzip2) and file size and entropy $k \times S_c$ for a few files from Calgary Corpus

Now let us compare the context level entropies of alice29.txt (Canterbury Corpus) transformed with LIPT and LIT. Table 13 gives level-wise entropies (column 3 and 5). The first column of Table 13 gives the order of the context (context level). The second and the fourth columns give the total count of predicting next input character using the respective context order for respective methods. For instance the count 108186 for LIPT in the row for context level 5 denotes that 108186 characters were predicted using up to five preceding characters using PPMD method. The third and fifth columns give the entropies or the bits needed to encode each character in the respective context level.

Context Length (Order)	LIPT (count)	LIPT (Entropy-BPC)	LIT (count)	LIT (Entropy-BPC)
5	108186	2.09	84534	2.31
4	15266	3.12	18846	3.22
3	7550	2.03	9568	2.90
2	6762	2.18	6703	1.84
1	2489	3.50	2726	3.52
0	79	6.35	79	6.52
Total /Average	140332	2.32	122456	2.50

Table 13: Entropy comparison for LIPT and LIT for alice29.txt from Canterbury Corpus

The entropy in each context level is calculated by multiplying the probability of the context level, which is the count for that context level divided by the total count (size of file in bytes), by the total number of bits needed to encode all the characters in the respective context. Bits needed to encode each character in respective context is calculated dynamically based on the probability of occurrence of input character given a certain preceding string of characters with length equal to respective context order. The $\log_2$ of this probability gives the number of bits to encode that particular character. The sum off all number of bits needed to encode each character using the probability of

occurrence of that character after a certain string in the respective context gives the total bits needed to encode all the characters in that context level.

Note that in Table 13 by average (in last row of the table) we mean the weighted average which is the sum of BPC multiplied by respective count in each row, divided by the total count. Note also that the average entropy for PPMD with LIT is more than the entropy for PPMD with LIPT, whereas LIT outperforms LIPT in average BPC performance. Although the entropy is higher but note that the total size of the respective transform files are different. LIPT has larger file size compared to LIT. Multiplying entropy which is average bits/symbol or character with total count from Table 13, for LIPT we have 2.32× 140332= 325570.24 bits, which is equal to 40696.28 bytes. From Table 13 for LIT, we have 2.50×122456= 306140 bits, which is equal to 38267.5 bytes. We see that the total number of bits and hence bytes for LIT is less than LIPT. This argument can be applied to justify performance of our transforms with Bzip2 as it is BWT based method and BWT exploits context. This can be shown for other transforms and for other files as well. Here we have only given one file as an example to explain. We have obtained this data for all the files in Calgary corpus as well.

Our transforms keep the word level context of the original text file but adopt a new context structure on the character level. The frequency of the repeated words remains the same in both the original and transformed file. The frequency of characters is different. Both these factors along with the reduction in file size contribute towards the better compression with our transforms. The context structure affects the entropy S or S_t and reduction in file size affects s. We have already discussed that compression is inversely proportional to the product of these two variables. In all our transforms, to generate our transformed dictionary, we have sorted the words according to frequency of usage in our English Dictionary D. For LIPT, words in each length block in English Dictionary D are sorted in descending order according to frequency. For ILPT, there is a sorting based on descending order of frequency inside each initial letter block of English Dictionary D. For NIT, there is no blocking of words. The whole dictionary is one block. The words in the dictionary are sorted in descending order of frequency. LIT uses the same structure of dictionary as NIT. Sorting of words according to frequency plays a vital role in the size of transformed file and also its entropy. Arranging the words in descending order of usage frequency results in shorter codes for more frequently occurring words and larger for less frequently occurring words. This fact leads to smaller file sizes.

8. Conclusions

In this chapter, we have given an overview of classical and recent lossless text compression algorithms and then presented our research on text compression based on transformation of text as a preprocessing step for use by the available compression algorithms. For comparison purposes, we were primarily concerned with gzip, Bzip2, a version of PPM called PPMD and word based Huffman. We give theoretical explanation of why our transforms improved the compression performance of the algorithms. We have developed an web site (http://vlsi.cs.ucf.edu) as a test bed for all compression

algorithms. To use this, one has to click the "online compression utility" and the client could then submit any text file for compression using all the classical compression algorithms, some of the most recent algorithms including Bzip2, PPMD, YBS, RK and PPMonstr and, of course, all the transformed based algorithms that we developed and reported in this chapter. The site is still under construction and is evolving. One nice feature is that the client can submit a text file and obtain statistics of all compression algorithms presented in the form of tables and bar charts. The site is being integrated with the Canterbury website.

Acknowledgement

The research reported in this chapter is based on a research grant supported by NSF Award No. IIS-9977336. Several members of the M5 Research Group at the School of Electrical Engineering and Computer Science of University of Central Florida participated in this research. The contributions of Dr. Robert Franceschini and Mr. Holger Kruse with respect to star transform work are acknowledged. The collaboration of Mr. Nitin Motgi and Mr. Nan Zhang in obtaining some of the experimental results is also gratefully acknowledged. Mr. Nitin Motgi's help in developing the compression utility website is also acknowledged.

References

[Arna00] Z. Arnavut. Move-to-Front and Inversion Coding. *Proceedings of Data Compression Conference*, Snowbird Utah, pp. 193-202, 2000.

[AwMu01] F. Awan, A. Mukherjee. LIPT: A Lossless Text Transform to Improve Compression. *International Conference on Information Theory: Coding and Computing*, Las Vegas Nevada, IEEE Computer Society, April 2001, pp 452- 460.

[Awan01] F. Awan, Lossless Reversible Text Transforms, *MS thesis*, University of Central Florida, July, 2001.

[BaKS99] B. Balkenhol, S. Kurtz , and Y. M. Shtarkov. Modifications of the Burrows Wheeler Data Compression Algorithm . *Proceedings of Data Compression Conference*, Snowbird Utah, pp. 188-197, 1999.

[BeM89] T. C. Bell and A. Moffat. *A note on the DMC data compression scheme*. The British Computer Journal, 32(1):16--20, 1989.

[BuWh94] M. Burrows and D.J. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. *SRC Research Report* 124, Digital Systems Research Center, Palo Alto, CA., 1994.

[Bzip2] Bzip2 memory usage http://krypton.mnsu.edu/krypton/software/bzip2.html

[Cant00] Calgary and Canterbury Corpi http://corpus.canterbury.ac.nz

[Chap00] B. Chapin. Switching Between Two On-line List Update Algorithms for Higher Compression of Burrows-Wheeler Transformed Data. *Proceedings of Data Compression Conference*, Snowbird Utah, pp. 183-191, 2000.

[ClW84] J.G. Cleary, and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. Comm. COM*-32, 4 (1984), pp. 396-402.

[ClTW95] J.G. Cleary, W.J. Teahan, and Ian H. Witten. Unbounded Length Contexts for PPM, *Proceedings of Data Compression Conference*, March 1995, pp. 52-61.

[CoH87] G.V. Cormack and R.N. Horspool, Data Compression Using Dynamic Markov Modeling, *Computer Journal*, Vol. 30, No. 6, 1987,pp. 541-550.

[Effo00] M. Effros. PPM Performance with BWT Complexity: A New Method for Lossless Data Compression. *Proceedings of Data Compression Conference*, Snowbird Utah, pp. 203-212, 2000.

[Fenw96] P. Fenwick. Block Sorting Text Compression. *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australia, January 31 – February 2, 1996.

[FrMu96] R Franceschini and A. Mukherjee. Data Compression Using Encrypted Text. Proceedings of the third Forum on Research and Technology, Advances on Digital Libraries, ADL 96, pp. 130-138.

[GBLL98] J. D. Gibson, T. Berger, T. Lookabaugh, D. Lindbergh and R.L Baker. Digital Compression for Multimedia: Principles and Standards. Morgan Kaufmann, 1998.

[Gute71] http://www.promo.net/pg/

[Howa93] P.G.Howard. The Design and Analysis of Efficient Lossless Data Compression Systems (Ph.D. thesis). Providence, RI:Brown University, 1993.

[Huff52] D. A. Huffman, A Method for the Construction of Minimum Redundancy Codes, *Proceedings of the Institute of Radio Engineers* 40, 1952, pp.1098-1101.

[KrMu97-1] H. Kruse and A. Mukherjee. Data Compression Using Text Encryption. *Proceedings of Data Compression Conference*, 1997, IEEE Computer Society Press, pp. 447.

[KrMu97-2] H. Kruse and A. Mukherjee. Preprocessing Text to Improve Compression Ratios. *Proceedings of Data Compression Conference*, 1998, IEEE Computer Society Press 1997, pp. 556.

[Lar98] N.J. Larsson. The Context Trees of Block Sorting Compression. N. Jesper Larsson: The Context Trees of Block Sorting Compression. *Proceedings of Data Compression Conference*, 1998, pp 189- 198.

[Moff90] A. Moffat. Implementing the PPM data Compression Scheme, IEEE Transaction on Communications, 38(11), pp.1917-1921, 1990

[PPMDH] PPMDH ftp://ftp.elf.stuba.sk/pub/pc/pack/

[RK-1] RK archiver http://rksoft.virtualave.net/

[RK-2] RK archiver http://www.geocities.com/SiliconValley/Lakes/1401/compress.html

[SaOH00] K. Sadakane, T. Okazaki, and H. Imai. Implementing the Context Tree Weighting Method for Text Compression. *Proceedings of Data Compression Conference*, Snowbird Utah, pp. 123-132, 2000.

[Sal00] D. Salomon. Data Compression: The Complete Reference. 2nd Edition, Springer Verlag, 2000.

[Say00] K. Sayood. Introduction to Data Compression. Morgan Kaufman Publishers, 1996.

[Sewa00] J. Seward. On the Performance of BWT Sorting Algorithms. *Proceedings of Data Compression Conference*, Snowbird Utah, pp. 173-182, 2000.

[ShW98] C. E. Shannon, W. Weaver, The Mathematical Theory of Communication, University of Illinois Press, 1998.

[Sh48] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, Vol. 27, pp.379-423,623-656, 1948.

[TREC00] http://trec.nist.gov/data.html

[WiST95] F. Willems, Y. M. Shtarkov, and T .J.Tjalkens. The Context-Tree Weighting Method: Basic Properties. *IEEE Transaction on Information Theory*, IT-41(3), pp. 653-664, 1995.

[WiMB99] I. H. Witten, A. Moffat, T. Bell. Managing Gigabyte, Compressing and Indexing Documents and Images. 2nd Edition, Morgan Kaufmann Publishers, 1999.

[YBS] YBS compression algorithm http://artest1.tripod.com/texts18.html

[ZiL77] A. Lempel and J. Ziv. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, pages 337--343, May 1977.

A Dictionary-Based Multi-Corpora Text Compression System

Weifeng Sun Amar Mukherjee Nan Zhang
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL. 32816
{wsun, amar, nzhang}@cs.ucf.edu

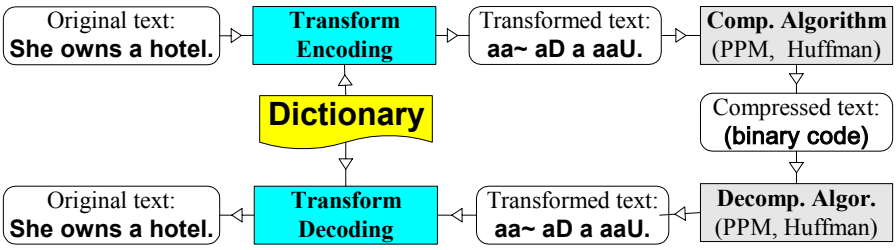
Abstract *In this paper we introduce StarZip, a multi-corpora lossless text compression utility which incorporates StarNT, our newly proposed transform algorithm. StarNT is a dictionary-based fast lossless text transform algorithm which utilizes ternary search tree to expedite transform encoding. For large files, viz. 400 Kbytes or more, our experiments show that the compression time is no worse than those obtained by bzip2 and gzip, and much faster than PPMD. However, if the file size is small, our algorithm is 28.1% and 50.4% slower than bzip2 -9, gzip -9 respectively and 21.1% faster compared to PPMD. We also achieve a superior compression ratio than almost all the other recent efforts based on BWT and PPM. StarNT is especially suitable for domain-specific lossless text compression used for archival storage and retrieval. Using domain-specific dictionaries, StarZip achieve an average improvement (in terms of BPC) of 13% over bzip2 –9, 19% over Gzip -9, and 10% over PPMD.*

7. Introduction

It is well known that there are theoretical predictions on how far a source file can be losslessly compressed, but no existing compression methods consistently attain these bounds over wide classes of text files. Researchers in the field of lossless text compression have developed several sophisticated approaches, such as Huffman encoding, arithmetic encoding, the Ziv-Lempel family, Dynamic Markov Compression, Prediction by Partial Matching (PPM [Moff90]), and Burrow-Wheeler Transform (BWT [BuWh94]) based algorithms, etc. LZ-family methods are dictionary based compression algorithm. Variants of LZ algorithm form the basis of UNIX compress, gzip and pkzip tools. PPM encodes each symbol by considering the preceding k symbols (an "order k" context). PPM achieves better compression than any existing compression algorithms, but it is intolerably slow and also consumes large amount of memory to store context information. Several efforts have been made to improve PPM [Howa93, Effr00]. BWT rearranges the symbols of a data sequence that share the same unbounded context by cyclic rotation followed by lexicographic sort operations. BWT utilize move-to-front and an entropy coder as the backend compressor. A number of efforts have been made to improve its efficiency [BKSh99, Chap00, Arna00].

In the recent past, the M5 Data compression group, University of Central Florida (http://vlsi.cs.ucf.edu/) has developed a family of reversible Star-transformations[ArMu01, FrMu96] which applied to a source text along with a backend compression algorithm, achieves better compression. The transformation is designed to make it easier to compress the source file. *Figure 1* illustrates the paradigm. The basic idea of the transform module is to transform the text into some intermediate form, which can be compressed with better efficiency. The transformed text is provided to a backend compressor which compresses the transformed text. The decompress process is just the reverse of the compression process. The drawback of these compression algorithms is

Figure 1. Text transform paradigm



that execution time performance and runtime memory expenditure is relatively high compared with the backend compression algorithm such as bzip2 and gzip.

In this paper we first introduce a fast algorithm for transform encoding and transform decoding, called *StarNT*. Facilitated with our proposed transform algorithm, bzip2 -9, gzip –9 and PPMD all achieve a better compression performance in comparison to most of the recent efforts based on PPM and BWT. Results shows that, for Calgary corpus, Canterbury corpus and Gutenberg corpus, StarNT achieves an average improvement in compression ratio of 11.2% over bzip2 –9, 16.4% over gzip –9, and 10.2% over PPMD. This algorithm utilizes *Ternary Search Tree* [BeSe97] in the encoding module. With a finely tuned dictionary mapping mechanism, we can find a word in the dictionary at time complexity *O(1)* in the transform decoding module. Results shows that for all corpora, the average compression time using the transform algorithm with bzip2 –9, gzip –9 and PPMD is 28.1% slower, 50.4% slower and 21.2% faster compared to the original bzip2 –9, gzip –9 and PPMD respectively. The average decompression time using the new transform algorithm with bzip2 –9, gzip –9 and PPMD is 100% slower, 600% slower and 18.6% faster compared to the original bzip2 –9, gzip –9 and PPMD respectively. However, since the decoding process is fairly fast, this increase is negligible. We draw a significant conclusion that bzip2 in conjunction with StarNT is better than both gzip and PPMD both in time complexity and compression performance.

Based on this transform, we developed StarZip[1], a domain-specific lossless text compression utility for archival storage and retrieval. StarZip uses specific dictionaries for specific domains. In our experiment, we created five corpora from publicly available website, and derived five domain-specific dictionaries. Results show that the average

[1] The name StarZip was suggested by Dr. Mark R. Nelson of Dr. Dobb's Journal. A featured article by Dr. Nelson appears in Dr. Dobb's Journal, August 2002, pp. 94-96.

BPC improved 13% over bzip2 –9, 19% over Gzip -9, and 10% over PPMD for these five corpora.

8. StarNT: the Dictionary-Based Fast Transform

In this section, we will first discuss three considerations from which this transformation is derived. After that a brief discussion about ternary search tree is presented, followed by detailed description about how the transform works.

8.1. Why Another New Transform?

There are three considerations that lead us to this transform algorithm.

First, we gathered data of word frequency and length of words information from our collected corpora (All these corpora are publicly available), as depicted in *Figure 2*. It is clear that almost more than 82% of the words in English text have the lengths greater than three. If we can recode each English word with a representation of no more than three symbols, then we can achieve a certain kind of "*pre-compression*". This consideration can be implemented with a fine-tuned transform encoding algorithm, as is described later.

The second consideration is that the transformed output should be compressible to the backend compression algorithm. In other words, the transformed immediate output should maintain some of the original context information as well as provide some kind of "artificial" but strong context. The reason behind this is that we choose BWT and PPM algorithms as our backend compression tools. Both of them predict symbols based on context information.

Finally, the transformed codewords can be treated as the offset of words in the transform dictionary. Thus, in the transform decoding phrase we can use a hash function to achieve *O(1)* time complexity for searching a word in the dictionary. Based on this consideration, we use a continuously addressed dictionary in our algorithm. In contrast, the dictionary is split into 22 sub-blocks in LIPT [AwMu01]. Results show that the new transform is better than LIPT not only in time complexity but also in compression performance.

The performance of search operation in the dictionary is the key for fast transform encoding. Here there are several approaches: the first one is hash table, which is really fast, but designing a fine-skewed hash function is very difficult. And unsuccessful searches using hash table are disproportionately slow. Another option is digital search tries, which are also very fast, however, they have exorbitant space requirements: suppose each node has 52-way branching, then one node will typically occupy at least 213 bytes. A three-level digital search tries with this kind of nodes will consume (1+52+52*52)*213=587,241 bytes! Or, we can use binary search tree (LIPT take this approach), which is really space efficient. Unfortunately, the search cost is quite high for binary search. In this transform, we utilize ternary search trees, which provide a very fast transform encoding speed at a low

Figure 2: Frequency of words versus length of words in English text

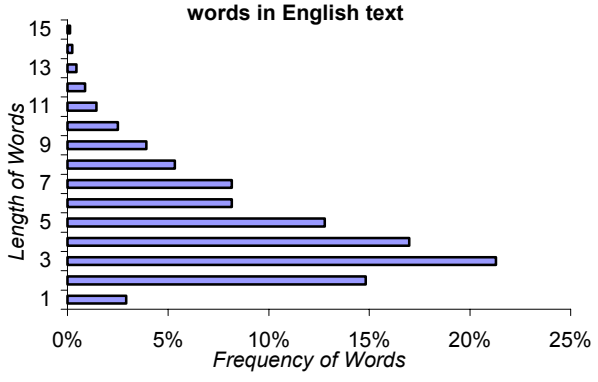
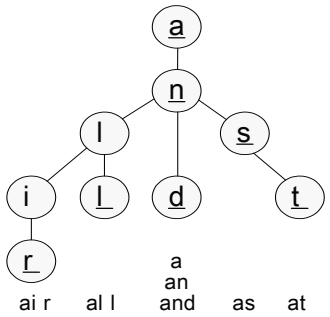


Figure 3: a Ternary Search Tree for seven words



storage overhead.

8.2. Ternary Search Tree

Ternary search trees [BeSe97] are similar to digital search tries in that strings are split in the trees with each character stored in a single node as *split char*. Besides, three pointers are included in one node: left, middle and right. All elements less than the split character are stored in the left child, those greater than the split character are stored in the right child, while the middle child contains all elements with the same character.

Search operations in ternary search trees are quite straightforward: current character in the search string is compared with the *split char* at the node. If the search character is less than the *split char*, then go to the left child; if the search character is greater than the *split char*, go to the right child; otherwise, if the search character is equal to the *split char*, just go to the middle child, and proceed to the next character in the search string. Searching for a string of length k in a ternary search tree with n strings will require at most $O(\log n+k)$ comparisons. The construction time for the ternary tree takes $O(n\log n)$ time.

Furthermore, ternary search trees are quite space-efficient. In *Figure 3*, seven strings are stored in this ternary search tree. Only nine nodes are needed. If multiple strings have same prefix, then the corresponding nodes to these prefixes can be reused, thus memory requirements is reduced in scenarios with large amounts of data.

In the transform encoding module, words in the dictionary are stored in the ternary search trees with the address of corresponding codewords. The ternary search tree is split into 26 distinct ternary search trees. An array is used to store the addresses of these ternary search trees corresponding to the letters [a..z] of the alphabet in the main root node. Words have the same starting character are stored in same sub-tree, viz. all words starting with 'a' in the dictionary exist in the first sub-tree, while all words start with 'b' in second sub-tree, and so on.

In each leaf node of the ternary search tree, there is a pointer which points to the corresponding codeword. All codewords are stored in a global memory that is prepared in advance. Using this technique we can avoid storing the codeword in the node, which enables a lot of flexibility as well as space-efficiency. To expedite the tree-build operation, we allocate a big pool of nodes to avoid overhead time for allocating storage for nodes in sequence.

Ternary search tree is sensitive to insertion order: if we insert nodes in a good order (middle element first), we end up with a balanced tree for which the construction time is the small; if we insert nodes in the order of the frequency of words in the dictionary, then the result would be a skinny tree that is very costly to build but efficient to search. In our experiment, we confirmed that insertion order has a lot of performance impact in the transform encoding phase. Our approach is just to follow the *natural* order of words in the dictionary. Result shows that this approach works very well.

8.3. Dictionary Mapping

The dictionary used in this experiment is prepared in advance, and shared by both the transform encoding module and the transform decoding module. In view of the three considerations mentioned in section 2.2, words in the dictionary $\boldsymbol{D}$ are sorted using the following rules:

- Most frequently used words are listed at the beginning of the dictionary. There are 312 words in this group.

- The remaining words are stored in ***D*** according to their lengths. Words with longer lengths are stored after words with shorter lengths. Words with same length are sorted according to their frequency of occurrence.
- To achieve better compression performance for the backend data compression algorithm, only letters [a..zA..Z] are used to represent the codeword.

With the ordering specified above, each word in ***D*** is assigned a corresponding codeword. The first 26 words in ***D*** are assigned "a", "b", … , "z" as their codewords. The next 26 words are assigned "A", "B", … , "Z". The 53rd word is assigned "aa", 54th "ab". Following this order, "ZZ" is assigned to the 2756th word in ***D***. The 2757th word in ***D*** is assigned "aaa", the following 2758th word is assigned "aab", and so on. Hence, the most frequently occurred words are assigned codewords form "a" to "eZ". Using this mapping mechanism, totally 52+52*52+52*52*52 = 143,364 words can be included in ***D***.

8.4. Transform Encoding

In the transform encoding module, the shared static base dictionary is read into main memory and the corresponding ternary search tree is constructed.

A *replacer* is initiated to read in the input text character by character, which performs the replace operation when it recognizes that the string of a certain length of input symbols (or the lower case form of this string sequence) exists in the dictionary. Then it outputs the corresponding codeword (appended with a special symbol if needed) and continues. If the input symbol sequence does not exist in the dictionary, it will be output with a prefix escape character "*". Currently only single words are stored in the dictionary. For future implementation, the transform module will contain phrases or sequence of words with all kinds of symbols (such as upper case letter, smaller case letters, blank symbols, punctuations, etc) in the dictionary.

The proposed new transform differs from our earlier Star-family transforms with respect to the meaning of the character '*'. Originally it was used to indicate the beginning of a codeword. In our new transform, it denotes that the following word does not exist in the dictionary ***D***. The main reason for this change is to minimize the size of the transformed intermediate text file, because smaller size can expedite the backend encoding/decoding.

Currently only lower-case words are stored in the dictionary ***D***. Special operations were designed to handle the first-letter capitalized words and all-letter capitalized words. The character '~' appended at the end of an encoded word denotes that the first letter of the input text word is capitalized. The appended character '`' denotes that all letters of the word are capitalized. The character '\' is used as escape character for encoding the occurrences of '*', '~', '`', and '\' in the input text.

8.5. Transform Decoding

The transform decoding module performs the inverse operation of the transform-encoding module. The escape character and special symbols ('*', '~', '`', and '\') are recognized and processed, and the transformed words are replaced with their original forms, which are stored continuously in a pre-allocated memory to minimize the memory usage. And their addresses are stored in an array sequentially. There is a very important property of the dictionary mapping mechanism that can be used to achieve O(1) time complexity to search a word in the dictionary. Because codewords in the dictionary are assigned sequentially, and only letters [a..zA..Z] are used, these codewords can be

deemed as the address in the dictionary. In our implementation, we use a very simple one-to-one mapping to calculate the index of the corresponding original words in the array that stores the address of the dictionary words.

9. Performance Evaluation

Our experiments were carried out on a 360MHz Ultra Sparc-IIi Sun Microsystems machine housing SunOS 5.7 Generic_106541-04. We choose bzip2 (-9), PPMD (order 5) and gzip (-9) as the backend compression tool. In this section, all compression results are derived from the Canterbury-Calgary and Gutenburg Corpus. The reason why we choose bzip2, gzip and PPMD as our backend compressor is that bzip2 and PPM outperform other compression algorithms, and gzip is widely used.

9.1. Timing Performance

9.1.1. Timing Performance with Backend Compression Algorithm

When backend data compression algorithms, i.e. bzip2 -9, gzip -9 and PPMD (k=5) are used along with the new transform, the compression system also provides a favorable timing performance. Following conclusions can be drawn form *Table 1* and *Table 2*:

1. The average compression time using the new transform algorithm with bzip2 –9, gzip –9 and PPMD is 28.1% slower, 50.4% slower and 21.2% faster compared to the original bzip2 –9, gzip –9 and PPMD respectively.
2. The average decompression time using the new transform algorithm with bzip2 –9, gzip –9 and PPMD is 1 and 6 times slower, and is 18.6% faster compared to the original bzip2 –9, gzip –9 and PPMD respectively. However, since the decoding process is fairly fast for bzip2 and gzip, this increase is negligible.

It must be pointed out that when the new transform is concatenated with backend compressor, the increase in compression time is imperceptible to human being, if the size of the input text file is small, viz. less than 100Kbytes. As the size of the input text file increases, the relative impact the new transform introduced to backend compressor decreases proportionally. Especially, if the text is very large, compression using the transform will be faster than that without using the transform. An example in case is file bible.txt (size 4,047,392), the runtime decreases when the transform is applied on bzip2. The reason behind this is the file size decrease introduced by the transform, which makes backend compressor runs faster. It also should be pointed out that our programs have not yet been well optimized, there is potential for less time and smaller memory usage.

Table 1: Comparison of Encoding Speed of Various Compressor with/without Transform (in seconds)

Corpus	bzip2	bzip2 + StarNT	Bzip2 + LIPT	Gzip	gzip + StarNT	gzip + LIPT	PPMD	PPMD + StarNT	PPMD + LIPT
Calgary	0.36	0.76	1.33	0.23	0.86	1.7	9.58	7.94	9.98
Canterbury	2.73	3.04	5.22	2.46	3.36	6.59	68.3	55.7	69.2
Gutenburg	4.09	4.4	7.01	2.28	3.78	9.67	95.4	75.2	90.9
AVERAGE	1.69	2.05	3.47	1.33	2.06	4.47	41.9	33.9	41.9

Table 2: Comparison of Decoding Speed of Various Compressor with/without Transform (in seconds)

Corpus	bzip2	bzip2 + StarNT	bzip2 + LIPT	gzip	Gzip + StarNT	gzip + LIPT	PPMD	PPMD + StarNT	PPMD + LIPT
Calgary	0.13	0.33	1.66	0.04	0.27	1.64	9.65	8.07	10.9
Canterbury	0.82	1.53	6.77	0.22	1.16	9.15	71.2	57.8	77.2
Gutenburg	1.15	2.22	8.46	0.29	1.44	7.99	95.4	76.9	98.7
AVERAGE	0.51	1	4.4	0.14	0.72	5.27	43	35	46.4

The data in *Table 2* and *Table 3* also shows that the new transform works better than LIPT when they are applied with backend compression algorithm both in the compression phase and decompression phase.

9.1.2. Improvement upon LIPT

Table 1 illustrates the transform encoding/decoding time of our transform algorithm against LIPT without any backend compression algorithm involved. All data are un-weighted average value of 10 runs. The results can be summarized as follows.

1. For all corpora, the average transform encoding and decoding times using the new transform decrease about 76.3% and 84.9%, respectively, in comparison to times taken by LIPT.
2. The decoding module runs faster than encoding module by 39.3% on average. The main reason is that the hash function used in the decoding phase is more efficient than the ternary search tree in the encoding module.

Table 3: Comparison of Transform Encoding and Decoding Speed (in seconds)

Corpora	StarNT		LIPT	
	Transform Encoding	Transform Decoding	Transform Encoding	Transform Decoding
Calgary	0.42	0.18	1.66	1.45
Canterbury	1.26	0.85	5.7	5.56
Gutenburg	1.68	1.12	6.89	6.22
AVERAGE	0.89	0.54	3.75	3.58

9.2. Compression Performance of StarNT

We compared the compression performance (in terms of BPC, bits per character) of our proposed transform with the results of the recent improvements on BWT and PPM as listed on *Table 4* and *Table 5,* taken from the references given in the respective columns. From *Table 4* and *Table 5*, it can be seen that our transform algorithm outperforms almost all the other improvements. The detailed compression results in terms of BPC for our test corpus are listed in the Appendix. Following is the summary:

1. Facilitated with StarNT, bzip2 -9, gzip –9 and PPMD an average improvement in compression ratio of 11.2% over bzip2 –9, 16.4% over gzip –9, and 10.2% over PPMD.
2. The StarNT works better than LIPT when is applied with backend compressor.
3. In conjunction with bzip2, our transform algorithm achieves a better compression performance than the original PPMD. Combined with the timing performance, we conclude that bzip2+StarNT is better than PPMD both in time complexity and compression performance.

9.3. Space Complexity

In our implementation the transform dictionary is a static dictionary shared by both transform encoder and transform decoder. The typical size of the off-line dictionary is about 0.5MB. The run-time overhead is as follows: the memory requirement of the ternary search tree for our dictionary in the transform encoding phase is about 1M bytes. In the transform decoding phase, less memory is needed. Bzip2 –9 needs about 7600K. PPM is programmed to use about 5100K + file size. So our transform algorithm takes insignificant overhead compared to bzip2 and PPM in memory usage.

Table 1: BPC comparison of new approaches based on BWT

File	Size (byte)	Mbswic [Arna00]	bks98 [BKSh99]	best x of 2x-1 [Chap00]	bzip2+ LIPT	bzip2+ StarNT
bib	111261	2.05	1.94	1.94	1.93	**1.71**
book1	768771	2.29	2.33	2.29	2.31	**2.28**
book2	610856	2.02	2.00	2.00	1.99	**1.92**
news	377109	2.55	2.47	2.48	2.45	**2.29**
paper1	53161	2.59	2.44	2.45	2.33	**2.21**
paper2	82199	2.49	2.39	2.39	2.26	**2.14**
progc	39611	2.68	2.47	2.51	2.44	**2.32**
progl	71646	1.86	1.70	1.71	1.66	**1.58**
progp	49379	1.85	**1.69**	1.71	1.72	**1.69**
trans	93695	1.63	1.47	1.48	1.47	**1.22**
Average		2.20	2.09	2.10	2.06	**1.94**

Table 2: BPC comparison of new approaches based on PPM

File	Size (byte)	Multi-alphabet CTW order 16 [SOIm00]	NEW [Effr00]	PPMD+ LIPT	PPMD+ StarNT
Bib	111261	1.86	1.84	1.83	**1.62**
book1	768771	**2.22**	2.39	2.23	2.24
book2	610856	1.92	1.97	1.91	**1.85**
news	377109	2.36	2.37	2.31	**2.16**
paper1	53161	2.33	2.32	2.21	**2.10**
paper2	82199	2.27	2.33	2.17	**2.07**
progc	39611	2.38	2.34	2.30	**2.17**
progl	71646	1.66	1.59	1.61	**1.51**
progp	49379	1.64	**1.56**	1.68	1.64
trans	93695	1.43	1.38	1.41	**1.14**
Average		2.01	2.01	1.97	**1.85**

10. StarZip: A Multi-corpora Text Compression Tool

We have developed a multi-corpora text compression system StarZip, which incorporates StarNT as the basic transform. One of the key features of our StarZip compression system is to develop domain specific dictionaries and provide tools to develop such dictionaries. We made preliminary studies on specific dictionaries and obtained some encouraging results. We experimented with five corpora derived form ibiblio.com [literature (3064 files, 1.2 G), History (233 files, 9.11M), Political Science (969 files, 33.4M), Psychology (55 files, 13.3M) and Computer Network (RFC, 3237 files, 145M)] and created separate domain-specific dictionaries for each corpus with entries of 60533, 39740, 38464, 45165 and 13987, respectively. Each dictionary is

created using the algorithm mentioned in section 2.3. For Bzip2, the average BPC using domain-specific dictionary shows an improvement of 13% compared with the original Bzip2. Using domain-specific dictionary gives a 6% improvement in BPC compared with the case when a common English dictionary is used. For Gzip, the average BPC using domain-specific dictionary gives an improvement of 19% and 7% over using a common dictionary. For PPMD, the average BPC using domain-specific dictionary has an improvement of 10% compared with the original PPMD and gives 5% gain in BPC when a common dictionary is used. We propose to conduct similar experiments on a large number of corpora to evaluate the effectiveness of our approach.

11. Conclusion

In this paper, we have proposed a new transform algorithm which utilizes ternary search tree to expedite transform encoding operation. This transform algorithm also includes an efficient dictionary mapping mechanism based on which searching operation can be performed with time complexity O(1) in the transform decoding module.

We compared the compression effectiveness versus compression/decompression speed and compression ratio when bzip2 –9, gzip –9 and PPMD are used as backend compressor with our transform algorithm, as illustrated in *Figure 4* and *Figure 5*. It is very clear that bzip2 + StarNT could provide a better compression performance that maintains an appealing compression and decompression speed.

Figure 4: Compression effectiveness versus compression speed

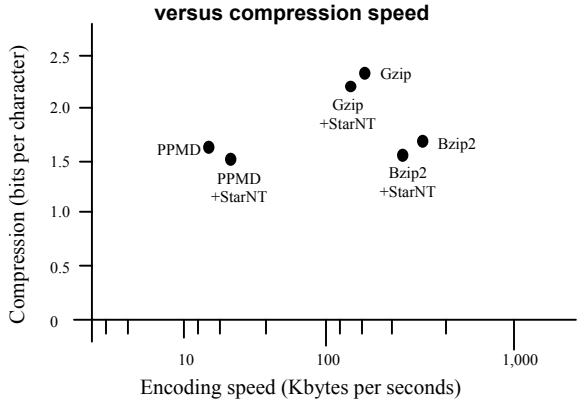
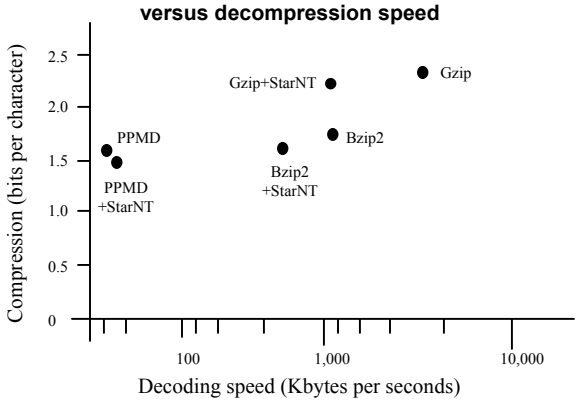


Figure 5: Compression effectiveness versus decompression speed



Acknowledgement

This research is partially supported by NSF grant number: IIS-9977336 and IIS-0207819.

References

[AwMu01] F. Awan and A. Mukherjee, "LIPT: A Lossless Text Transform to improve compression", Proceedings of International Conference on Information and Theory : Coding and Computing, IEEE Computer Society, Las Vegas Nevada, 2001.

[Arna00] Z. Arnavut, "Move-to-Front and Inversion Coding", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird, Utah, March 2000, pp. 193-202.

[BKSh99] B. Balkenhol, S. Kurtz , and Y. M. Shtarkov, "Modifications of the Burrows Wheeler Data Compression Algorithm", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, March 1999,pp. 188-197.

[BeSe97] J. L. Bentley and Robert Sedgewick, "Fast Algorithms for Sorting and Searching Strings", Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, January, 1997

[BuWh94] M. Burrows and D.J. Wheeler, "A Block-Sorting Lossless Data Compression Algorithm", *SRC Research Report* 124, Digital Systems Research Center, Palo Alto, CA, 1994.

[Chap00] B. Chapin, "Switching Between Two On-line List Update Algorithms for Higher Compression of Burrows-Wheeler Transformed Data", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, March 2000, pp. 183-191.

[Effr00] M. Effros, "PPM Performance with BWT Complexity: A New Method for Lossless Data Compression", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, March 2000, pp. 203-212.

[FrMu96] R. Franceschini and A. Mukherjee, "Data Compression Using Encrypted Text", *Proceedings of the third Forum on Research and Technology*, Advances on Digital Libraries, ADL 96, pp. 130-138.

[Howa93] P.G.Howard, "The Design and Analysis of Efficient Lossless Data Compression Systems", Ph.D. thesis. Providence, RI:Brown University, 1993.

[KrMu98] H. Kruse and A. Mukherjee, "Preprocessing Text to Improve Compression Ratios", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, 1998, pp. 556.

[Moff90] A. Moffat, "Implementing the PPM data Compression Scheme", *IEEE Transaction on Communications*, 38(11), pp.1917-1921, 1990

[SOIm00] K. Sadakane, T. Okazaki, and H. Imai, "Implementing the Context Tree Weighting Method for Text Compression", *Proceedings of Data Compression Conference,* IEEE Computer Society, Snowbird Utah, March 2000, pp. 123-132.

[Sewa00] J. Seward, "On the Performance of BWT Sorting Algorithms", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird, March 2000, pp. 173-182.

Appendix: Compression Results (BPC) Using StarNT

File	Size (byte)	bzip2 –9	bzip2 –9 +StarNT	gzip –9	gzip –9 +StarNT	PPMD	PPMD +StarNT
paper5	11954	3.24	2.76	3.34	2.78	2.98	2.56
paper4	13286	3.12	2.46	3.33	2.55	2.89	2.34
paper6	38105	2.58	2.29	2.77	2.40	2.41	2.17
progc	39611	2.53	2.32	2.68	2.45	2.36	2.17
paper3	46526	2.72	2.28	3.11	2.47	2.58	2.24
progp	49379	1.74	1.69	1.81	1.76	1.70	1.64
paper1	53161	2.49	2.21	2.79	2.35	2.33	2.10
progl	71646	1.74	1.58	1.80	1.65	1.68	1.51
paper2	82199	2.44	2.14	2.89	2.35	2.32	2.07
trans	93695	1.53	1.22	1.61	1.25	1.47	1.14
bib	111261	1.97	1.71	2.51	2.12	1.86	1.62
news	377109	2.52	2.29	3.06	2.57	2.35	2.16
book2	610856	2.06	1.92	2.70	2.24	1.96	1.85
book1	768771	2.42	2.28	3.25	2.66	2.30	2.24
grammar.lsp	3721	2.76	2.42	2.68	2.38	2.36	2.06
xargs.1	4227	3.33	2.90	3.32	2.87	2.94	2.57
fields.c	11150	2.18	1.98	2.25	2.03	2.04	1.81
cp.html	24603	2.48	2.01	2.60	2.13	2.26	1.85
asyoulik.txt	125179	2.53	2.27	3.12	2.58	2.47	2.24
alice29.txt	152089	2.27	2.06	2.85	2.38	2.18	2.00
lcet10.txt	426754	2.02	1.81	2.71	2.14	1.93	1.78
plrabn12.txt	481861	2.42	2.23	3.23	2.60	2.32	2.22
world192.txt	2473400	1.58	1.36	2.33	1.87	1.49	1.30
bible.txt	4047392	1.67	1.53	2.33	1.87	1.60	1.47
kjv.gutenberg	4846137	1.66	1.55	2.34	1.94	1.57	1.47
anne11.txt	586960	2.22	2.05	3.02	2.47	2.13	2.01
1musk10.txt	1344739	2.08	1.88	2.91	2.34	1.91	1.82
world95.txt	2736128	1.57	1.34	2.37	1.89	1.49	1.29
average		2.28	2.02	2.70	2.25	2.14	1.92