LIPT : A Reversible Lossless Text Transform to Improve Compression Performance

Fauzia S. Awan, Nan Zhang, Nitin Motgi, Raja T. Iqbal, and Amar Mukherjee
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL. 31826
{fauzia, nzhang, nmotgi, riqbal, amar}@cs.ucf.edu

Abstract. Lossless compression researchers have developed highly sophisticated approaches, such as Huffman encoding, arithmetic encoding, the Lempel-Ziv family, Dynamic Markov Compression (DMC), Prediction by Partial Matching (PPM), and Burrows-Wheeler Transform (BWT) based algorithms. We propose an alternative approach in this paper to develop a reversible transformation that can be applied to a source text that improves existing algorithm's ability to compress. The basic idea behind our approach is to encode every word in the input text file, which is also found in the English text dictionary that we are using, as a word in our transformed static dictionary. These transformed words give shorter length for most of the input words and also retain some context and redundancy. Thus we achieve some compression at the preprocessing stage as well as retain enough context and redundancy for the compression algorithms to give better results. Bzip2 with our proposed text transform, LIPT, gives **5.24%** improvement in average BPC over Bzip2 without LIPT, and PPMD (a variant of PPM with order 5) with LIPT gives **4.46%** improvement in average BPC over PPMD (with order 5) without LIPT, for a set of text files extracted from Calgary and Canterbury corpuses, and also from Project Gutenberg. Bzip2 with LIPT, although 79.12% slower than the original Bzip in compression time, achieves average BPC almost equal to that of original PPMD and is also 1.2% faster than the original PPMD in compression time.

Keywords: Lossless text transform, dictionary based, LIPT, Bzip2, PPM

1. Related Work and Background

In the last decade, we have seen an unprecedented explosion of textual information through the use of Internet, digital library and information retrieval system. It is estimated that by the year 2002 the National Service Provider backbone will have an estimated traffic around 27,645 Gbps and that the growth will continue to be 100% every year [ ]. The text data competes for 45% of the total Internet traffic but no lossles compression standard for text has yet been proposed [ ]. A number of sophisticated algorithms have been proposed for lossless text compression of which Burrows Wheeler Transform (BWT) [1] and Prediction by Partial Matching (PPM) [2] outperform the classical algorithms like Huffman, arithmetic and LZ families of Gzip and Unix-compress.

BWT sorts a block of data and produces a reverse string on which a scheme called move to front (MTF) is applied. More is the frequency of common characters in the text, better is the performance of this algorithm. PPM compresses a sequence of symbols by predicting probabilities from preceding symbols. The preceding symbols considered in predicting these probabilities make up the *context* and the length of this context is called

the *order* of PPM. PPMD[] is a variant of PPM which compute escape probability as *(u/2)/n*, where *u* is the number of unique token seen so far, and *n* is the number of token seen so far. It is slightly better than PPMC and regarded to be suitable for bounded context model in PPM while PPMC is better for unbounded PPM model.
BWT has proved to be the most efficient and a number of efforts have been made to improve its efficiency [3,4,5,6]. PPM gives better compression ratio than that of BWT but is very slow. Efforts have also been made to improve PPM [7,8,9]. In this paper, we propose a new text transformation technique, called Length Index Preserving Transform (LIPT) that makes the text better compressible by most of the above methods. Our compression results on text files derived from the Canterbury, Calgary [14] and Gutenberg corpus [19] show consistent uniform compression ratio improvement in the range of around 3% to 6% over the best results obtained by various modifications in BWT [3,4,5,6] and PPM [7,8,9]. This comes at the expense of some storage overhead whose amortized cost is shown to be negligible. The overhead is due to sharing of a dictionary of approximately 0.5M bytes of data between the sender and receiver. Huffman compression method also needs sharing of the same static dictionary at both the sender and receiver end as does our method hence we compare the word-based Huffman with LIPT (we used Bzip2 as the compressor). We show that Bzip2 with LIPT outperforms word-based Huffman for text files. We present comparison of compression and decompression times for LIPT and other compression methods. We also measure typical transmission time improvements over the Internet and propose and implement a preliminary infrastructure for dictionary management.
The genesis of LIPT (Length Index Preserving Transform) can be traced to several other similar transforms developed by the M-5 Research Group at the Department of Computer Science, University of Central Florida to improve the compression performance for English text [10]. These transforms in chronological order are * -encoding (Star Encoding), LPT (Length Preserving Transform), RLPT (Reverse Length Preserving Transform), SCLPT (Shortened Context Length Preserving Transform). We will briefly describe only the Star transform to motivate our basic approach to the problem and then report LIPT, which is producing the best results.

The Star Transform

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. We have explained the basic approach of our compression method in the previous sentence but let us use the same sentence as an example to explain our point further. Lets rewrite it with a lot of spelling mistakes: ***Our philosopy of compresion is to transfom the txt into som intermedate form which can be compresed with bettr efficency and which xploits the natural redndancy of the laguage in making this tranformation.*** Most people will have no problem to read it. This is because our visual perception system recognizes each word with an approximate signature pattern for the word as opposed to an actual and exact sequence of letters and we have a dictionary in our brain, which associates each misspelled word with a corresponding, correct word. The signatures for the word for computing machinery could be arbitrary as long as they are unique. We claim that the following sequence of characters represents the first sentence of this paragraph (philosophy statement without spelling mistakes) in a unique fashion:

****a ********** ** ********* *d *e ******** *** ***a ***c ***b *********** ***d ***** **b *a **********a ***e ****** *********a **c ***** ******* *** ******a *********b ** *** ******** *b *****a **** ****************

There are exactly six two letter words in the sentence (of, be, in, on, to) which can be uniquely encoded as (**, *a, *b, *c, *d) where '*' is a special place holder character. We use both lower case and capital letters and if there were more than 53 two-letter words, the 54th word would have gotten the encoding a* etc. All other groups of words of same length can be similarly encoded. There are 53^l possible encoding for words of length l which is more than sufficient for English language. Note the transformed text heavily skews the letter frequency distribution to the '*' character and indeed in the compressed text '*' needs only one bit by most classical and recent compression algorithms. It is of interest to mention that we star encoded a 60,000 word English dictionary, which never needed more than two letters of the alphabet for its unique encoding.

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 character '*' is used as a placeholder in the Star Encoding but in the rest of the text transforms (LPT, RLPT, SCLPT, LIPT) '*' is used to denote the beginning of an encoded word. 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 alphabets 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 alphabets 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. From Table 1 given below it can be seen that Star encoding (*-encoding) gives a better average BPC (bits per character) performance for character-based Huffman, Gzip, and Bzip2 but gives worse average BPC performance for word-based arithmetic coding and PPMD. Note that the BPC figures are rounded off to two decimal places and %improvement factors are calculated on actual figures and not rounded off BPC. One of the main reason for the degraded performance is caused by the non-English words and non-alphabetical symbols in the text. Define the missing rate is the percentage of bytes in a file which is not in a word of our dictionary. In the current test corpus the average missing rate for files is 25.56%, namely 25.56% of the bytes are kept as it was or some special characters are added. For the files with better performance the missing rate is 23.42%, while the files with worse performance have an average missing rate of 28.58%. These missing words are transformed as they were and can be regarded as "noise" in the star converted file for further compression. Unlike LIPT, most of the bytes hit are converted to '*' character in star encoding. So the untransformed words have very different context to those generated by transformed words. 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. Also notice that, although the average BPC for star encoding is worse than original, for PPMD, there are 16 files has improved BPC, 12 files has degraded BPC. Therefore the amount of hit words is an important factor for the final compression ratio.

	Original (BPC)	***-encoded (BPC)**	**Improvement of *-encoded over Original %**
Huffman (character based)	4.87	**4.12**	15.34

Arithmetic (word based)	**2.71**	2.90	-6.90
Gzip-9	2.70	**2.52**	6.81
Bzip2	2.28	**2.24**	1.88
PPMD	**2.1384**	2.1387	-0.01

Table 1: BPC comparison among different compression methods

2. Length Index Preserving Transform (LIPT)

Earlier we proposed several improvements over the *-encoding called LPT, RLPT and SCLPT. These transformations (LPT and RLPT) follow the same basic principle as that of the star algorithm except that the sequence of stars are replaced by a fixed sequence of letters from the alphabet ending in the letter 'x' in the natural order they appear in the alphabet or in reverse natural order starting from 'x', which act as the place holders for '*' characters. This sequence is then appended at the end by an address encoded in three letters beginning with the letter 'z' which allows the correct decoded word to be read from the dictionary. The purpose of the fixed sequence was to create domination of some fixed but artificial context in the transformed text that can be exploited by the backend compression algorithm. In a later improvement (SCLPT), the fixed sequence of letters was simply replaced by the first letter of the sequence.

We propose a novel reversible lossless text transform called LIPT (Length Index Preserving Transform) which gives better compression and time performance than all the other transforms mentioned above except in one case where Star Encoding gives 0.04% better average BPC result for Gzip –9 than LIPT. While encoding, LIPT replaces the first character after '*' in LPT approach by an alphabet [a-z] denoting the length of the original input word. In our approach 'a' denotes length of 1, 'b' denotes length of 2, 'z' denotes length of 26, 'A' denotes length of 27, so on till 'Z' which denotes length of 52. All the transforms use an English language dictionary that has about 60,000 words and takes about 0.5 Mbytes in uncompressed form. This dictionary needs to be the same at the compression and decompression ends. This English dictionary D 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 English language. 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 dictionary D_i. Let $FR(D_i[j])$ denote the LIPT encoded word, for the j^{th} word in the dictionary D_i in the English dictionary D. The last three characters represent a variable-length encoding similar to Huffman encoding and produces 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. The number of words in English dictionary goes up like a bell curve until it hits maximum at length 8 and then falls beyond length 8 similarly until about length 21. 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. In LIPT, a word $D_i[j]$, in the original English dictionary D, where $j = 0$ is encoded as $FR(D_i[0]) = *c_l$ where c_l stands for a character in the alphabet set [a-z] each denoting corresponding length [1-26]. For $j>0$, $FR(D_i[j]) = * c_l c[c][c]$ where c cycles through [a-z, A-Z]. If $1 \le j \le 52$ then $FR(D_i[j]) = *c_l c$. If $53 \le j \le 2704$ then $FR(D_i[j]) = *c_l cc$. If $2705 \le j \le 140608$ then $FR(D_i[j]) = *c_l ccc$. For instance, the 1st word of length 10 in the

English dictionary *D* will be encoded as $FR(D_{10}[0])$ = "*j", $FR(D_{10}[1])$ = "*ja", $FR(D_{10}[27])$ = "*jA", $FR(D_{10}[53])$ ="*jaa", $FR(D_{10}[79])$ = "*jaA", $FR(D_{10}[105])$ = "*jba" , $FR(D_{10}[2757])$ = "*jaaa", $FR(D_{10}[2809])$ = "*jaba", and so on. This scheme allows for a total of 140608 words encoding for each length, which is more than enough. By the above method a transformed English dictionary is generated in the memory. All the input words are transformed according to above method. Each word that is not found in the English dictionary *D* is transferred as it is.

The transformed words have '*', then alphabet denoting the original word length, and then the respective word length block offset. This sequence is used in decoding at the receiver end. The length block indicator is picked up from the character after '*' which is then used to access the respective length block in the English dictionary *D*. The alphabets after the length alphabet in the transformed word give the offset from the start of the length block. Thus the pointer is directly moved to the respective position pointed by the offset in the respective length block in the English dictionary *D* and the word found at this position is the needed decoding for the transformed word. For instance the transformed word "*jaba" denotes an original word of length 10 as the alphabet 'j' after '*' denotes 10. The three alphabets after 'j' give the offset 1+52+52*52+52+1= 2809. Hence the 2810^{th} word in the block of length 10 in the original dictionary *D* is looked up and replaces the transformed word "*jaba". If there is a capitalization mask at the end of the transformed word then it is applied to the decoded word. For instance if there is '~' at the end like "*jaba~" then the initial letter of the decoded word is capitalized. The words without '*' in front of them are non-transformed words and hence are written to decoded file as it is without any decoding. The escape character '\' is stripped from the special characters on decoding.
Formally the LIPT algorithm can be stated as follow:

Encoding of the English dictionary *D*
wordlength ← 0
oldwordlength ← 0

Experimental Results and Discussion for LIPT
LIPT achieves a sort of pre-compression for all the text files. This was proved from our experiments. LIPT beats all the other methods listed in Table 1 uniformly. It also beats word-based arithmetic coding and PPMD as opposed to *-encoding. Arithmetic coding with LIPT shows 3.42% improvement in average BPC over the original word-based arithmetic coding and PPMD with LIPT shows 4.46% improvement over original PPMD for the text files in all the three corpuses combined. As Bzip2 and PPM are considered the most efficient compression algorithms and lot of efforts are being made to improve their performance so in this section we focus our attention to these two algorithms, and Gzip which is commonly used and commercially available compression method, and give results of using LIPT with these three compressors only. Tables below give a comparison for the average BPC performance. By average BPC we mean the unweighted average and in all the tables and results the BPC figures are rounded off to two decimal places and %improvement factors are calculated on actual figures and not rounded off BPC. Data in Table 2(a), 2(b), and 2(c) show that the overall average BPC for all the three corpuses.

Combining the three corpuses and taking the average BPC for all the text files , 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 3 gives the average BPC comparison of PPMD without LIPT and PPMD with LIPT for each Corpus individually. For all the textfiles in all three corpuses, PPMD (order 5) gives an overall average BPC of **2.14**, and using PPMD with LIPT gives average BPC of **2.04**, and overall improvement of **4.46%**. LIPT gives much better BPC results than *-encoding and these original compression methods. Table 4 shows the BPC results for the original Gzip –9, and Gzip –9 with LIPT. Gzip –9 with LIPT shows an improvement of 6.78% in average BPC over the original Gzip –9.

FileNames	File Size	Bzip2 (BPC)	Bzip2 with LIPT (BPC)
Calgary			
paper5	11954	3.24	**2.95**
paper4	13286	3.12	**2.74**
paper6	38105	2.58	**2.40**
Progc	39611	2.53	**2.44**
paper3	46526	2.72	**2.45**
progp	49379	1.74	**1.72**
paper1	53161	2.49	**2.33**
Progl	71646	1.74	**1.66**
paper2	82199	2.44	**2.26**
trans	93695	1.53	**1.47**
bib	111261	1.97	**1.93**
news	377109	2.52	**2.45**
book2	610856	2.06	**1.99**
book1	768771	2.42	**2.31**
Average BPC		2.36	**2.22**

(a)

FileNames	File Size	Bzip2 (BPC)	Bzip2 with LIPT (BPC)
Canterbury			
grammar.lsp	3721	2.76	**2.58**
xargs.1	4227	3.33	**3.10**
fields.c	11150	2.18	**2.14**
cp.html	24603	2.48	**2.44**
asyoulik.txt	125179	2.53	**2.42**
alice29.txt	152089	2.27	**2.13**
lcet10.txt	426754	2.02	**1.91**
plrabn12.txt	481861	2.42	**2.33**
world192.txt	2473400	1.58	**1.52**
bible.txt	4047392	1.67	**1.62**
kjv.gutenberg	4846137	1.66	**1.62**
Average BPC		2.26	**2.17**

(b)

FileNames	File Size	Bzip2 (BPC)	Bzip2 with LIPT (BPC)
Gutenberg			
anne11.txt	586960	2.22	**2.12**
1musk10.txt	1344739	2.08	**1.98**
world95.txt	2988578	1.54	**1.49**
Average BPC		1.95	**1.86**

(c)

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

FileNames	File Size	PPMD (BPC)	PPMD with LIPT
Calgary			
paper5	11954	2.98	**2.74**
paper4	13286	2.89	**2.57**
paper6	38105	2.41	**2.29**
Progc	39611	2.36	**2.30**
paper3	46526	2.58	**2.37**
Progp	49379	1.70	**1.68**
paper1	53161	2.33	**2.21**
Progl	71646	1.68	**1.61**

FileNames	File Size	PPMD (BPC)	PPMD with LIPT
Canterbury			
grammar.lsp	3721	2.36	**2.21**
xargs.1	4227	2.94	**2.73**
fields.c	11150	2.04	**1.97**
cp.html	24603	2.26	**2.22**
asyoulik.txt	125179	2.47	**2.35**
alice29.txt	152089	2.18	**2.06**
lcet10.txt	426754	1.93	**1.86**
plrabn12.txt	481861	2.32	**2.27**

FileNames		PPMD (BPC)	PPMD with LIPT
Gutenberg			
anne11.txt	586960	2.13	**2.04**
1musk10.txt	1344739	1.91	**1.85**
world95.txt	2988578	1.48	**1.45**
Average BPC		1.84	**1.78**

paper2	82199	2.32	**2.17**
trans	93695	1.47	**1.41**
bib	111261	1.86	**1.83**
news	377109	2.35	**2.31**
book2	610856	1.96	**1.91**
book1	768771	2.30	**2.23**
Average BPC		2.23	**2.12**

(a)

world192.txt	2473400	1.49	**1.45**
bible.txt	4047392	1.60	**1.57**
kjv.gutenberg	4846137	1.57	**1.55**
Average BPC		2.11	**2.02**

(b)

(c)

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.

FileNames	File Size	GZIP (BPC)	GZIP with LIPT (BPC)
Calgary			
paper5	11954	3.34	**3.05**
paper4	13286	3.33	**2.95**
paper6	38105	2.77	**2.61**
Progc	39611	2.68	**2.61**
paper3	46526	3.11	**2.76**
Progp	49379	1.81	**1.81**
paper1	53161	2.79	**2.57**
Progl	71646	1.80	**1.74**
paper2	82199	2.89	**2.62**
trans	93695	1.61	**1.56**
bib	111261	2.51	**2.41**
news	377109	3.06	**2.93**
book2	610856	2.70	**2.48**
book1	768771	3.25	**2.96**
Average BPC		2.69	**2.51**

(a)

FileNames	File Size	GZIP (BPC)	GZIP with LIPT (BPC)
Canterbury			
grammar.lsp	3721	2.68	**2.61**
xargs.1	4227	3.32	**3.13**
fields.c	11150	2.25	**2.21**
cp.html	24603	2.60	**2.55**
asyoulik.txt	125179	3.12	**2.89**
alice29.txt	152089	2.85	**2.60**
lcet10.txt	426754	2.71	**2.42**
plrabn12.txt	481861	3.23	**2.96**
world192.txt	2473400	2.33	**2.18**
bible.txt	4047392	2.33	**2.18**
kjv.gutenberg	4846137	2.34	**2.19**
Average BPC		2.70	**2.54**

(b)

FileNames	File Size	GZIP (BPC)	GZIP with LIPT (BPC)
Gutenberg			
anne11.txt	586960	3.02	**2.75**
1musk10.txt	1344739	2.91	**2.62**
world95.txt	2988578	2.31	**2.15**
Average BPC		2.75	**2.51**

(c)

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

If we combine all the three corpuses and compute average BPC then 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 in section 5 on time performance analysis that Bzip2 with LIPT is much faster than original PPMD. (Note that although Bzip2 with LIPT gives lower BPC than the

original Bzip2, former is much slower than the later as discussed in section 5 of this paper).

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 which has three text only files from Project Gutenberg.

LIPT introduces frequent occurrence of common characters for BWT and good context for PPM as well as it compresses the original text. Cleary, Teahan, and Witten [15], and Larsson [17] 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 [16] 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. 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 length alphabet) that enhances the performance of Bzip2 (which uses BWT) and PPM as proved by results given above. LIPT generates encoded file, which is smaller in size than the original text file. This smaller file is fed to the compression algorithm. Because of the small input file fed to the compressor along with proportional amount of context, the output file size from the compressor exploiting such characteristics (Bzip2 and PPM) is also smaller. Thus the net compression gained when LIPT is applied before compression algorithm is greater.

Context Length (Order)	Original PPMD (order 5) (bytes)	Original PPMD (order 5) (BPC)	PPMD (order 5) with LIPT (bytes)	PPMD (order 5) with LIPT (BPC)
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	6.30
Average BPC	152088	3.50	**140332**	**3.20**

Table 5: Comparison of bits per character in each context length used by PPMD, and PPMD with LIPT for the file alice29.txt

The data in Table 5 shows that LIPT uses less number of bits for context order 3, 2, and 1 as compared to the original PPMD. The average BPC is also lower for PPMD with LIPT. When we calculate the average BPC of PPMD with LIPT dividing the total bits used by the total count in the original file (140332*8/152088), we get net BPC of 2.05 for alice29.txt.

In the PPM view to the LIPT, more or longer deterministic context help to improve the performance. This is because of the "reuse" of the code and the accumulation of the

frequency of the certain context. The implicit gain in LIPT is of the following reason. The current models are not sematics model for compression. Therefore, we can transform the words into strings without semantic meaning. The '*' in LIPT keep the context of the delimiter space character and '*', the counter is necessary to distinguish the words in the same block. The letter in between is not only the indicator for the word length, but also determines the context in the original words as in LPT. It determines a context of length equal to the *word-length -1 – length-of-counter*. Thus, the longer the word we encode, the longer the deterministic context it represents. Noticing that most of the words in the dictionary are longer than 4, we preencode the words to avoid more occurrence in the context table for PPM algorithms. Furthermore, the code generated by LIPT is also in a fashion of keeping the longest initial context for words with same length. For the words with different lengths or words in the different block, except for the length indicator, the rest of the code, the '*' and counter, are generated under the same principle. This feature will give a skewed distribution of context in the PPM context table or BWT. Therefore yields better compression ratio.

3. Comparison of LIPT with Word-based Huffman

A typical word-based Huffman model is a zero-order word-based semi-static model [18]. 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 numbers. To avoid the infinite number of such strings, one way of encoding is to break them in to smaller pieces e.g. every 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 totally 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 [18] is selected to encode the words although the code length is the same as other word-based Huffman encoding. The main reason in [18] 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.

Comparing with word-based Huffman coding, LIPT is a preprocessor to transform the original words, which are predefined in a fixed English dictionary, to an artificial "language" which has no linguistic semantic meaning. However, every word can distinguish from one another and has the similar context patterns among the words with same length or have similar offset in the different word blocks. The transformation does not generate any direct statistics for the word frequencies. But it extracts deterministic strings within the word, which are encoded by a shorter code in an orderly manner. If new text files are added, the whole frequency distribution table should be recomputed as well as the Huffman codes for them. LIPT uses same alphabets as ASCII code and the words not in the dictionary are either kept in the original form or just appended at the end with a single special character. So when further compression, such as Gzip, BWT, or PPM is performed, the words in the dictionary and not in the dictionary may still have

chance to share the local context. Table 6 shows the BPC comparison. The word-based Huffman model is adopted from the system in [18]. For LIPT, we extract the alphabetic strings in the text as the dictionary and build the LIPT dictionary accordingly for each file. In contrast to the approach given in [18], we do not include the words composed of digits and mixture of alphabets and digits as well as other special characters. Namely, we try to make a fair comparison, however, word-based Huffman still makes use of a larger scope from the definition of "words". Comparing the average BPC, the Managing Gigabyte word-based Huffman model [18] has a **2.506** BPC for our test corpus. LIPT with Bzip2 has a BPC of **2.169**. The gain is **13.439%**. LIPT does not give improvement over word based Huffman for files with mixed text such as source files for programming languages. We are still investigating this case and are looking for improvements in LIPT in order to achieve a uniform improvement over word-based Huffman for all types of files. For files with more English word, LIPT has a gain from **1.194%** to **34.750%**.

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

4. Dictionary Organization

To expedite searching, we pre-sort the dictionary lexicographically and use binary search which takes O(n log n + M * log n) number of comparisons, where M is number of words tokenized from the input file and n is the dictionary size. So, as M gets larger the performance degrades. We propose to organize dictionary into two levels. In level 1 we classify the words in dictionary based on the length of the word, and in level 2 we

classify it based on first character of the word. By this we confine our search domain to only small blocks of similar words that can expedite the process.
The words in the block are lexicographically sorted. The details of implementation are left out of the paper to conserve space. Experimental results show there is 61.808% improvement of search time by using this data structure. Memory overhead of 1.5KB is incurred to maintain this structure for faster memory operations. When new words are added they are placed at the end of respective blocks in the dictionary. This preserves the prior dictionary word-transform mapping, and scalability is provided without distortion in the dictionary, and yields a faster decoding time.

Dictionary Overhead
The size of dictionary is 0.5MB uncompressed and 197KB when compressed with Bzip2. In order to quantify the overhead, assume that the uncompressed size of the data to be transmitted is F and the uncompressed dictionary size is D. Then for Bzip2 with LIPT (data taken from Section 2), we can derive: $F \times 2.16 + D \times 2.28 \leq F \times 2.28$, which gives $F \geq 9.5$ MB. This means that to break even the overhead associated with dictionary, transmission of 9.5MB data has to be achieved. So if the normal file size for a transmission is say 1 MB then the dictionary overhead will break even after about 9.5 transmissions. All the transmission above this number contributes towards gain achieved by LIPT. Similarly for PPMD with LIPT, $F \geq 10.87$ MB. With increasing dictionary size, this threshold will go up, but in a scenario where thousands of files are transmitted, the amortized cost will be negligible.

5. Time Performance Analysis

The compression times are generated over the new Canterbury-Calgary and Gutenburg Corpus. These experiments where carried out on 360MHz Ultra Sparc-IIi Sun Microsystems machine housing SunOS 5.7 Generic_106541-04. Each time recorded for compression is average of two readings. In our experiments we compare compression times of Bzip2, Gzip(option-9) and PPMD against Bzip2 with LIPT, Gzip with LIPT and PPMD with LIPT. Average compression time using LIPT is 79.12% slower, 223% slower and 1.2% faster compared to original Bzip2, Gzip and PPMD respectively. The corresponding results for decompression times are 93.3% slower, 566% slower and 5.9% faster compared to original Bzip2, Gzip and PPMD respectively.
In an ideal channel, the reduction of transmission time is directly proportional to the amount of compression. But in a typical Internet scenario with fluctuating bandwidth, congestion and protocols of packet switching this does not hold true. Since PPMD is very slow, we excluded this from our measurement and conducted experiments on a typical day over the Internet. Our results are: transmission times without any compression for the corpus is 171.4415 seconds. There is 1.97% improvement in transmission by Gzip with LIPT over Gzip, and 5.91% improvement in transmission by Bzip2 with LIPT over Bzip2. Bzip2 is 36.18% better than Gzip without LIPT and 42.90% better than Gzip with LIPT. These improvements come with additional cost of processing required on the servers/nodes. The final paper will contain extensive test data supporting the above figures.

Acknowledgement
This research is supported by NSF Award No. IIS-9977336.

References

[1] 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.

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

[3] 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.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] R. Franceschini, H. Kruse, N. Zhang, R. T. Iqbal, and A. Mukherjee. Lossless, Reversible Transformations that Improve Text Compression Ratios (submitted for publication).

[11] 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.

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

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

[14] http://corpus.canterbury.ac.nz

[15] 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.

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

[17] 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.
[18] I.H.Witten, A. Moffat, T. Bell, "Managing Gigabyte, Compressing and Indexing Documents and Images", 2nd Edition, Morgan Kaufmann Publishers, 1999.
[19] http://www.promo.net/pg/