Dictionary-Based Fast Transform for Text Compression[*]

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

Abstract

In this paper we present StarNT, a dictionary-based fast lossless text transform algorithm. With a static generic dictionary, StarNT achieves a superior compression ratio than almost all the other recent efforts based on BWT and PPM. This algorithm utilizes ternary search tree to expedite transform encoding. Experimental results show that the average compression time has improved by orders of magnitude compared with our previous algorithm LIPT and the additional time overhead it introduced to the backend compressor is unnoticeable.

Based on StarNT, we propose StarZip, a domain-specific lossless text compression utility. Using domain-specific static dictionaries embedded in the system, 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.

1. Introduction

In the last decade, we have seen an unprecedented explosion of textual information flow over Internet through electronic mail, web browsing, digital library and information retrieval systems, etc. Given the continued increase in the amount of data that needs to be transferred or archived, the importance of data compression is likely to increase in the foreseeable future. Researchers in the field of lossless data compression have developed several sophisticated approaches, such as Huffman encoding, arithmetic encoding, the Ziv-Lempel family, Dynamic Markov Compression, Prediction by Partial Matching (PPM [9]), and Burrow-Wheeler Transform (BWT [5]) based algorithms, etc. PPM achieves better compression than almost all existing compression algorithms, but it is intolerably slow and also consumes large amount of memory to store context information. BWT rearranges 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 the efficiency of PPM and BWT.

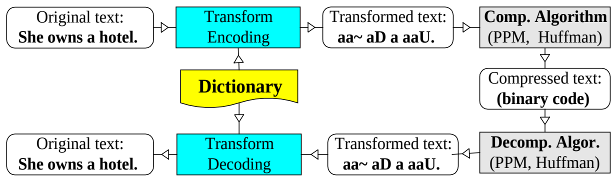

Figure 1. Text Transform Paradigm

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 [2, 8] which applied to a source text along with a backend compression algorithm, achieves better compression. 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 data compression module which compresses the transformed text. However, execution time performance and runtime memory expenditure of these compression systems have remained high compared with the backend compression algorithms such as bzip2 and gzip.

In this paper we introduce a fast transform algorithm, called Star New Transform (StarNT). Ternary Search Tree [4] is used in the transform to expedite the transform encoding. Compared with LIPT [2], the new transform achieves improvement not only in compression performance, but also in time complexity. Facilitated with StarNT, bzip2 and PPMD both achieve a better compression performance in comparison to most of the other recent efforts based on PPM and BWT. Experimental results show that, for our test corpus, StarNT achieves an average improve-

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

ment in compression ratio of 11.2% over bzip2 -9, 16.4% over gizp -9 and 10.2% over PPMD. Results show that, for our test corpus, the average compression time using the new transform with bzip2 -9, gzip -9 and PPMD is 28.1% slower, 50.4% slower and 21.2% faster compared with the original bzip2 -9, gzip -9 and PPMD, respectively. The average decompression time using StarNT with bzip2 -9, gzip -9 and PPMD is one and six times slower and 18.6% faster compared with the original bzip2 -9, gzip -9 and PPMD respectively. Besides, we draw a significant conclusion that bzip2 in conjunction with StarNT is better than PPMD both in time complexity and in compression performance.

Based on this transform, we developed StarZip, a domain specific lossless text compression utility for archival storage and retrieval. StarZip uses specific dictionaries for specific domains. We conducted experiments on five corpora. Five domain-specific dictionaries were constructed. Results show that, using StarNT, the average BPC (Bit per Character) improved 13% over bzip2 -9, 19% over Gzip -9, and 10% over PPMD.

2. StarNT: the Dictionary-Based Fast Transform

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

2.1. Why New Transform?

There are three considerations that lead us to the new transform algorithm.

First, we gathered data of word frequency and length of words information from our test corpus[1], as depicted in Figure 2. It is very clear that almost more than 82% of the words in English text have lengths greater than three. If we can recode each English word with a representation with 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 mapping mechanism, as described later.

The second consideration is that the transformed output should maintain some of the original context information as well as providing some kind of "*artificial*" but strong context, which can be explored by the backend compressor.

Besides the compression efficiency, fast transform is also desirable. In the transform decoding phase, we treat the transformed codewords as offsets of words in the dictionary. Thus, searching for a word in the dictionary takes constant time ($O(1)$). In the transform encoding phase, the underlaying data structure to support the transform operation is the

[1]About the information on the test corpus, please refer to Section 3.

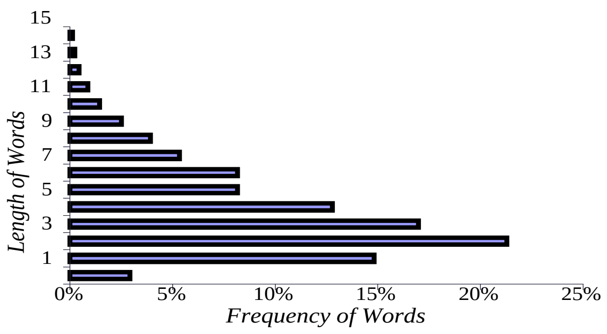

Figure 2. Word Frequency Distribution

key for transform efficiency. Following are several possible 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. Digital search tries are also very fast, however, they have exorbitant space requirements: suppose each node has 52-way branching, then one node will typically occupy 213 bytes. In our transform, a 21-level digital search trie is needed. It will consume nearly $52^{20} * 213 \approx 4.5 * 10^{36}$ bytes memory! Or, we can use binary search tree (LIPT take this approach), which is space efficient. Unfortunately, the search cost is quite high for binary search. In the new transform, we utilize ternary search tree to store the transform dictionary. This data structure provides a very fast transform encoding speed with a low storage overhead.

2.2. Ternary Search Tree

Ternary search trees [4] are similar to digital search tries in that strings are split in the tree with each character stored in a single node as *split char*. In ternary search tree each node contains three children: left child, middle child and right child. All elements less than the *split char* are stored in the left child, those greater than the *split char* are stored in the right child, while the middle child contains all elements with the same character.

Searching in Ternary search trees are quite straightforward: current character in the search string is compared with the *split char* at the node. If the character is less than the *split char*, then go to the left child; if the character is greater than the *split char*, then go to the right child; otherwise, if the character equals 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(nlogn)* time.

Furthermore, ternary search trees are quite space efficient. If several strings share same prefix, then corresponding nodes to these prefixes can be reused. As an example, Figure 3 illustrates a ternary search tree for seven words (a, air, all, an, and, as, at). In this ternary search tree, only nine nodes (instead of 16 nodes) are needed.

In the transform encoding module, words in the transform dictionary are stored in the ternary search tree with the address of corresponding codewords. The ternary search tree is split into 26 distinct ternary search sub-trees. The root addresses of these ternary search sub-trees are stored in an array. Each ternary search tree contains all words with same initial characters. For example, all words with initial character 'a' in the transform dictionary exist in the first ternary search sub-tree, while all words with initial character 'b' exist in the second sub-tree, and so on.

The order in which we insert nodes into the ternary search tree has a lot of performance impact. First, this order determines the time needed to construct the ternary search tree of the transform dictionary. Second, it also determines the performance of the search operation that is the key factor of the transform efficiency. There are several approaches to insert words. The first one is to insert nodes in the order of middle element first, thus the result is a balanced tree with a shortest construct time. The second approach is to insert nodes in the order of the words frequency, thus the result is a skinny tree that is very costly to build but efficient to search. In our transform we follow the natural order of words in the transform dictionary. Results show that this approach works very well.

2.3. Dictionary Mapping

The transform dictionary used in the 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 2.1, words in the transform dictionary ***D*** are sorted according to the following rules:

- Most frequently used words are listed in the beginning of the dictionary in the decreasing order of their frequency of occurrence. There are 312 words in this group.
- The remaining words are sorted in ***D*** according to their lengths. Words with longer lengths are stored after words with shorter lengths. Words with same length are sorted in the decreasing order of their frequency of occurrence.
- To achieve better compression performance for backend data compression algorithm, only letters [a..zA..Z] are used to represent the codeword.

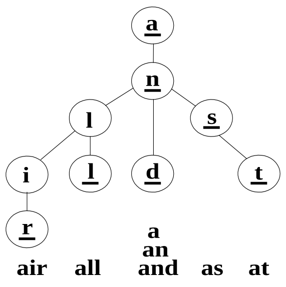

Figure 3. A Ternary Search Tree

With the ordering specified as above, all words in ***D*** are 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" as its codeword, 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, the transform dictionary ***D*** can contains totally 52+52*52 +52*52*52 = 143,364 entries.

2.4. Transform Encoding

The proposed new transform differs from earlier Star-family transforms with respect to the meaning of the character '*'. Originally the character '*' denotes the beginning of a codeword. In our new transform, it means that the following word does not exist in the transform dictionary ***D***. The main reason for this change is to reduce the size of the transformed intermediate file, thus the encoding/decoding time of the backend compression algorithm can be minimized. Currently the transform dictionary ***D*** only contains lower-case words. Dedicated operations were designed to handle the initial letter capitalized words and all-letter capitalized words. The character '∼' appended to the transformed word denotes that the initial letter of the corresponding word in the original text file is capitalized. The appended character ' ` ' denotes that all letters of the corresponding word in the original text file are capitalized. The character '\' is used as escape character for encoding the occurrence of '\*', '∼', ' ` ' , and '\' in the input text file.

A *transformer* is initiated to read in the input text, which performs the transform operation when it recognizes that the next word(strictly speaking, in the current implementation it should be the lower case form of the string) exists in the transform dictionary ***D***. Then it replaces the word with the corresponding codeword (if necessary, appending it with a special symbol) and continues. If the word does not exist in the transform dictionary ***D***, it will be attached with a prefix character "*". Currently the *transformer* can only manipulate single word. It is most probably that the compression ratio should increase if its ability can be extended to manage sequence of all kinds of symbols (such as capital letters, lower case letters, blank symbols, punctuation, etc) in future implementation.

2.5. Transform Decoding

The transform decoding module performs the inverse operation of the transform encoding module. The escape character and special symbols ('*', '$\sim$', ' ` ', and '\') are recognized, and transformed codewords are replaced with their original forms.

There is a very important property of the transform dictionary mapping mechanism that can be used to accelerate the transform decoding. Because codewords in the transform dictionary are assigned sequentially, and only letters [a..zA..Z] are used to denote codewords, these codewords can be deemed as the address of the codeword in the transform dictionary. Based on this observation, a simple address calculating function can be devised to calculate the address, thus corresponding words in the original text file can be retrieved from the transform dictionary in time complexity of $O(1)$. In the transform decoding module the transform dictionary can be implemented using a compact array.

3. Performance Evaluation

We evaluated the compression performance as well as the compression time improvement using our test corpus which consists of 28 files from the Canterbury-Calgary[2]and Gutenburg[3] Corpus. All these test files are listed in Tabel 1. The generic transform dictionary used for the test corpus is derived independently from an additional corpus. The experiment was carried out on a 360MHz Ultra Sparc-IIi Sun Microsystems machine housing SunOS 5.7 Generic_106541-04. Because bzip2 and PPM outperform other compression algorithms, and gzip is one of the most widely used compression tool, we choose bzip2 -9, PPMD (order 5) and gzip -9 as the backend compression tool.

3.1. Timing Performance over LIPT

In the transform encoding module of StarNT, the time complexity of constructing the ternary search tree with n strings is *O(nlogn)*, and searching a word with length k will need at most *O(logn+k) string comparisons*. While in the transform decoding module, the time complexity for searching a word in the transform dictionary is only *O(1)*. In LIPT, the time complexity of constructing the binary search tree with n strings is *O(nlogn)*, and searching a word will need at most *O(logn) character comparisons*, both in the transform encoding module and the transform decoding module. Because character comparison is more efficient than string comparison, it is clear that StarNT is faster than LIPT both in transform encoding module and in transform decoding module. It should be pointed out that StarNT use the same mechanism to parse the text file as LIPT. The experimental data is illustrated in Table 2. All these data are average values of 10 runs. By average time we mean the un-weighted average (simply taking the average of the transform encoding/decoding time) over the entire test corpus. The results can be summarized as follows:

- The average transform encoding time using new transform is only about 23.7% of that using LIPT.
- The average transform decoding time using new transform is only about 15.1% of that using LIPT.
- Especially, the speed of transform encoding phase and decoding phase of the transform algorithm is asymmetric. The decoding module runs faster than encoding module by 39.3% averagely. The main reason is that the simple address calculating function used in the transform decoding module is more efficient than the ternary search tree used in the transform encoding module.

[2]Canterbury-Calgary corpus: http://corpus.canterbury.ac.nz/
[3]Gutenburg corpus: http://www.promo.net/pg/

Table 1. Test Corpus

Corpus	File	Size (byte)
Calgary	paper5	11954
	paper4	13286
	paper6	38105
	progc	39611
	paper3	46526
	progp	49379
	paper1	53161
	progl	71646
	paper2	82199
	trans	93695
	bib	111261
	news	377109
	book2	610856
Canterbury	book1	768771
	grammar.lsp	3721
	xargs.1	4227
	fields.c	11150
	cp.html	24603
	asyoulik.txt	125179
	alice29.txt	152089
	lcet10.txt	426754
	plrabn12.txt	481861
	world192.txt	2473400
	bible.txt	4047392
Gutenberg	kjv.gutenberg	4846137
	anne11.txt	586960
	1musk10.txt	1344739
	world95.txt	2736128

Table 2. Comparison of Transform Time (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

3.2. Timing Performance with Backend Compression Algorithm

The encoding/decoding time when StarNT is combined with the backend data compression algorithms, i.e. bzip2 -9, gzip -9 and PPMD (order 5), is given in Table 3 and Table 4. In both tables we also present the corresponding data for LIPT. Following conclusions can be drawn from table 3 and Table 4:

- 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 with 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 1 and 6 times slower, and is 18.6% faster compared with the original bzip2 -9, gzip -9 and PPMD respectively.

It must be pointed out that when bzip2 is powered by StarNT, the time increase is imperceptible to human being, when the size of the input text file is small, viz. less than 100Kbytes. If the text size is large, viz. 400Kbytes or more, compressing using the transform will be faster than that without the transform. This is mainly due to the file size compaction introduced by the transform, which makes the backend compressor runs faster.

The data in Table 3 and Table 4 also shows that the new transform works better than LIPT when both are applied with backend compression algorithm, not only in the compression phase but also in the decompression phase. For example, the average compression time using the new transform algorithm with bzip2 -9, gzip -9 and PPMD is 39.7%, 54.5% and 19.5% faster compared with that when LIPT is applied. Similarly, the average decompression time using the new transform algorithm with bzip2 -9, gzip -9 and PPMD is 77.3%, 86.5% and 23.9% faster respectively.

3.3. Compression Performance of StarNT

We compared the compression performance (in terms of BPC) of our proposed transform with results of other recent

Table 3. Comparison of Encoding Speed (in seconds)

	Calgry	Cantrbry	Gutnbrg	AVRG
bzip2	0.36	2.73	4.09	1.69
bzip2+StarNT	0.76	3.04	4.40	2.05
bzip2+ LIPT	1.33	5.22	7.01	3.47
gzip	0.23	2.46	2.28	1.33
gzip+StarNT	0.86	3.36	3.78	2.06
gzip+LIPT	1.70	6.59	9.67	4.47
PPMD	9.58	68.3	95.4	41.9
PPMD+StarNT	7.94	55.7	75.2	33.9
PPMD+LIPT	9.98	69.2	90.9	41.9

Table 4. Comparison of Decoding Speed (in seconds)

	Calgry	Cantrbry	Gutnbrg	AVRG
bzip2	0.13	0.82	1.15	0.51
bzip2+StarNT	0.33	1.53	2.22	1.00
bzip2+ LIPT	1.66	6.77	8.46	4.40
gzip	0.04	0.22	0.29	0.14
gzip+StarNT	0.27	1.16	1.44	0.72
gzip+LIPT	1.64	9.15	7.99	5.27
PPMD	9.65	71.2	95.4	43.0
PPMD+StarNT	8.07	57.8	76.9	35.0
PPMD+LIPT	10.9	77.2	98.7	46.4

improvements on BWT and PPM, as listed in Table 5 and Table 6. The data in Table 5 and Table 6 have been taken from the references given in the respective columns.

In our implementation the generic transform dictionary is a static dictionary shared both by transform encoder and by transform decoder. This generic transform dictionary is derived independently from an additional corpus. This dictionary contains nearly 55K entries. The size of the dictionary is nearly 0.5MB. Because StarNT is used in a domain-specific fashion, where those domain-specific transform dictionaries are embedded into the compression system statically and it is not necessary to contain an extra copy of the transform dictionary in each compressed file (as is explained in Section 4), we can justify that our transform algorithm outperforms almost all the other improvements, as can be concluded from Table 5 and Table 6. Figure 4 illustrates the comparison of average compression performance for our test corpus. The result is very clear:

- In our test corpus, facilitated with StarNT, bzip2 -9, gzip -9 and PPMD achieve an average improvement in compression ratio of 11.2% over bzip2 -9, 16.4% over gzip -9, and 10.2% over PPMD.

Table 5. Comparison of Efforts Based on BWT

File	Bzip2	MTF [1]	MBW [3]	Chap [6]	bzip2+ LIPT	bzip2+ StarNT
bib	1.97	2.05	1.94	1.94	1.93	**1.71**
book1	2.42	2.29	2.33	2.29	2.31	**2.28**
book2	2.06	2.02	2.00	2.00	1.99	**1.92**
news	2.52	2.55	2.47	2.48	2.45	**2.29**
paper1	2.49	2.59	2.44	2.45	2.33	**2.21**
paper2	2.44	2.49	2.39	2.39	2.26	**2.14**
progc	2.53	2.68	2.47	2.51	2.44	**2.32**
progl	1.74	1.86	1.70	1.71	1.66	**1.58**
progp	1.74	1.85	**1.69**	1.71	1.72	**1.69**
trans	1.53	1.63	1.47	1.48	1.47	**1.22**
Average	2.14	2.20	2.09	2.10	2.06	**1.94**

Table 6. Comparison of Efforts Based on PPM

File	PPMD	CTW order 16 [10]	NEW [7]	PPMD + LIPT	PPMD + StarNT
bib	1.86	1.86	1.84	1.83	**1.62**
book1	2.30	**2.22**	2.39	2.23	2.24
book2	1.96	1.92	1.97	1.91	**1.85**
news	2.35	2.36	2.37	2.31	**2.16**
paper1	2.33	2.33	2.32	2.21	**2.10**
paper2	2.32	2.27	2.33	2.17	**2.07**
progc	2.36	2.38	2.34	2.30	**2.17**
progl	1.68	1.66	1.59	1.61	**1.51**
progp	1.70	1.64	**1.56**	1.68	1.64
trans	1.47	1.43	1.38	1.41	**1.14**
Average	2.03	2.01	2.01	1.97	**1.85**

- The new transform works better than LIPT when applied to backend compression algorithms.
- The compression performance of bzip2 powered by StarNT is superior to the original PPMD. Combined with the consideration of timing performance, it is clear that bzip2 combined with StarNT is better than PPMD both in time complexity and compression performance.

The size of the generic transform dictionary is nearly 0.5MB. The memory requirement of the dictionary, mainly it is composed of the overhead of the ternary search tree, in the transform encoding phase is nearly 1M bytes. Compared with bzip2 and PPM in memory occupation, the new transform algorithm takes insignificant overhead. It should be pointed out that our programs have not yet been well optimized. There is potential for less time and smaller memory usage.

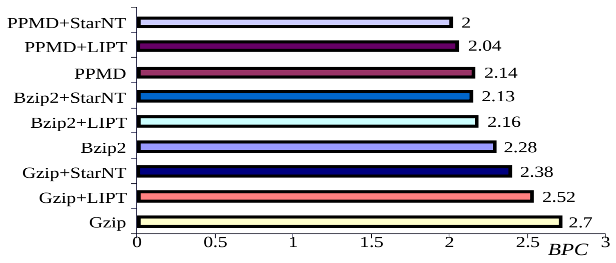

Figure 4. Compression Performance with/without Transform

4. StarZip: A Domain-specific Text Compression Tool

StarNT is the transform engine of our domain-specific text compression system StarZip. One of the key features of StarZip is that all text files belonging to a specific domain share one dedicated static transform dictionary which is embedded in the compression system. For different domains, there are different transform dictionaries. Since these dictionaries are embedded in the StarZip compression system, it is not necessary to provide an extra copy of the transform dictionary for every compressed file. We developed a set of tools to construct the transfrom dictionary for each specific domain. Each transform dictionary conforms to the mapping mechanism explained in section 2.3. Extending experiments were conducted on five corpora, and five domain-specific dictionaries were constructed, as is listed in Table 7. Encouraging results were achieved.

Table 7. Five Corpora in StarZip

Corpus	# of files	Size	Entries in the dictionary
Literature	3064	1.2 G	60533
History	233	9.11M	39740
Political	969	33.4M	38464
Psychology	55	13.3M	45165
Computer Network	3237	145M	13987

For bzip2, the average BPC using domain-specific transform dictionaries shows an improvement of 13% compared with the original bzip2 -9. Using domain-specific dictionaries gives a 6% improvement in BPC compared with the case when the generic transform dictionary[4] is used. For gzip, the average BPC using domain-specific transform dictionaries gives an improvement of 19% compared with the original gzip -9, and 7% improvement over the generic transform

[4]In fact, this is the generic transform dictionary presented in section 3.

dictionary. For PPMD, the average BPC using domain-specific transform dictionaries has an improvement of 10% compared with the original PPMD and gives 5% gain in BPC over the generic transform dictionary. We propose to conduct similar experiments on a large number of corpora to evaluate the effectiveness of our approach.

5. Conclusions

In this paper, we 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. In the transform decoding module searching can be performed in time complexity $O(1)$. We also achieve a superior compression ratio than almost all the other recent efforts based on BWT and PPM. Experimental results show that the average compression time has improved by orders of magnitude compared with our previous algorithm LIPT and the additional time overhead the proposed transform introduced to the backend compressor is unnoticeable.

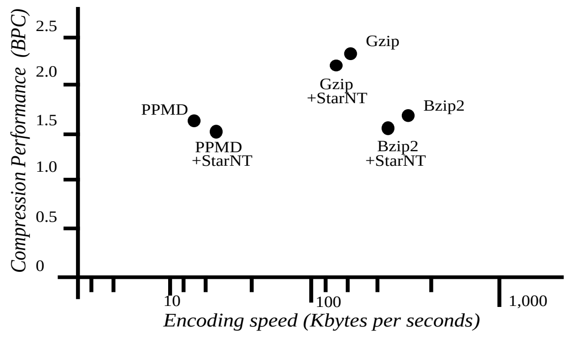

Figure 5. Compression Effectiveness versus Compression Speed

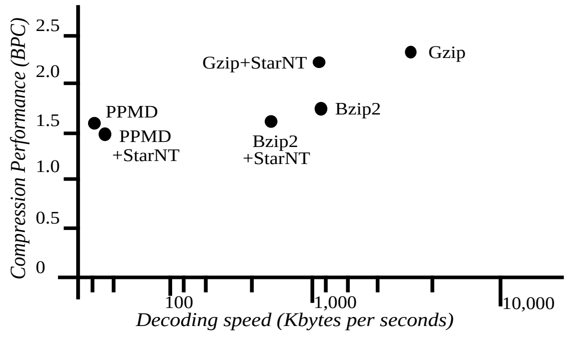

Figure 6. Compression Effectiveness versus Decompression Speed

We compared the compression effectiveness versus compression/decompression speed when bzip2 -9, gzip -9 and PPMD are used as backend compressor with our transform algorithm, as illustrated in Figure 5 and Figure 6. It is very clear that bzip2 combined with StarNT could provide a better compression performance that maintains an appealing compression and decompression speed.

Based on the new transform, we developed StarZip, a domain specific lossless text compression utility for archival storage and retrieval. Domain specific transform dictionaries are embedded in StarZip. Initial experimental results show that the average BPC improved 13% over bzip2 -9, 19% over gzip -9, and 10% over PPMD for these five corpora. One promising application of StarNT is to integrate it with some enterprise software, such as email system.

References

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

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

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

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

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

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

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

[8] R. Franceschini and A. Mukherjee. Data Compression Using Encrypted Text. In *Proceedings of the third Forum on Research and Technology, Advances on Digital Libraries*, pages 130–138. ADL, 1996.

[9] A. Moffat. Implementing the PPM data Compression Scheme. *IEEE Transaction on Communications*, 38(11):1917–1921, 1990.

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