The Block Lossless Data Compression Algorithm

Weiling Chang†, Binxing Fang†, Xiaochun Yun††, Shupeng Wang††

†Research Centre of Computer Network and Information Security Technology, Harbin Institute of Technology, Harbin 150001, China

††Institute of Computing Technology, Chinese Academy of Science, Beijing 100080, China

Summary

The mainstream lossless data compression algorithms have been extensively studied in recent years. However, rather less attention has been paid to the block algorithm of those algorithms. The aim of this study was therefore to investigate the block performance of those methods. The main idea of this paper is to break the input into different sized blocks, compress separately, and compare the results to determine the optimal block size. The select of optimal block size involves tradeoffs between the compression ratio and the processing time. We found that, for PPM, BWT and LZSS, a block size of greater than 32 KiB may be optimal. For Huffman coding and LZW, a moderate sized block (16KiB for Huffman and 32KiB for LZSS) is better. We also use the mean block standard deviation (MBSD) and locality of reference to explain the compression ratio. We found that good data locality implies a large skew in the data distribution, and the greater data distribution skew and the MBSD, the better the compression ratio. There is a positive correlation between MBSD and compression ratio.

Key words:

Block data compression, LZSS, LZW, PPM, BWT

1. Introduction

Lossless data compression techniques are often partitioned into statistical and dictionary techniques. Statistical compression assigns codes to symbols so as to match code lengths with the probabilities of the symbols. Dictionary method exploits repetitions in the data. We can also divide the lossless data compression into two major families: stream compression and block compression. Most compression methods operate in the streaming mode, where the codec inputs a byte or several bytes, processes them, and continues until an end-of-file is sensed. Block compression is applied to data chunks of varying sizes for many types of data streams, which is a sequence of bytes or bits, having a nominal length (a block size). In block compression algorithm, the input stream is read block by block and each block is compressed separately.

The block-based compression algorithms have been extensively used in many different fields.

While system processor and memory speeds have continued to increase rapidly, the gap between processor and memory and disk speed has still widened. Apart from advances in cache hierarchies, computer architects have addressed this speed gap mainly in a brute force manner by simply wasting memory resources. As a result, the size of caches and the amount of main memory, especially in server systems, has increased steadily over the last decades. Clearly, techniques that can use memory resources effectively are of increasing importance to bring down the cost, power dissipation, and space. Lossless data compression techniques have the potential to utilize in-memory resources more effectively. It is known from many independent studies that dictionary-based methods, such as LZ-variants, can free up more than 50% of in-memory resources.

In disaster backup system, a disk snapshot is an exact copy of the original file system at a certain point in time. The snapshot is a consistent view of the file system "snapped" at the point in time the snapshot is made. It preserves the disk file system by enabling you to revert to the snapshot in case something goes wrong. The bitmap contains one bit for every block which is multiples of disk sector on the snapped disk. Initially, all bitmap entries are zero. A set bit indicates that the appropriate block was copied from the snapped file system to the snapshot or changed. In order to achieve the best space utilization and support delta backup or recovery, we must use block compression algorithm to compress the disk data, therefore the block size should be carefully chosen. A long list of small blocks wastes space on pointers and harms compression efficiency; however, large blocks may contain substantial segments of unchanged data which wastes transmission bandwidth. The size of block depends on the granularity of delta backup and the compression efficiency of compression algorithm.

Delta encoding is a way of storing or transmitting data in the form of differences between sequential data

Supported by the National High-Tech Development 863 Program of China (Grant Nos. 2007AA01Z406, 2007AA010501, 2009AA01A403, 2009AA01Z437)

Manuscript received October 5, 2009
Manuscript revised October 20, 2009

rather than complete files. In addition to in-memory or on-the-fly compression and disaster backup system, block compression algorithm may be one of the delta compression solutions.

The basic compression algorithms, such as LZ, Huffman, PPM etc., have been extensively studied in recent years. However, few researchers attempts to focus on the block algorithm of basic compression algorithms. The purpose of this paper is to investigate the optimal block size for LZSS compression method and analyze factors which affect the optimal block size.

The rest of this paper is organized as follows. Section 2 reviews the traditional and recent related works for lossless compression. Section 3 introduces the factors which affect the compression efficiency of LZSS method. Experimental evidence and implementation considerations are presented in Section 4. Section 5 contains concluding remarks.

2. Previous work

In what follows we give a very brief account of some algorithms of this paper used.

2.1 Huffman Coding and Related Techniques

Huffman coding [1] is an entropy encoding algorithm used for lossless data compression. It uses a specific method for choosing the representation for each symbol, resulting in a prefix-free code that expresses the most common characters using shorter strings of bits than are used for less common source symbols. Huffman coding is optimal when the probability of each input symbol is a negative power of two. Prefix-free codes tend to have slight inefficiency on small alphabets, where probabilities often fall between these optimal points. "Blocking", or expanding the alphabet size by coalescing multiple symbols into "words" of fixed or variable-length before Huffman coding, usually helps, especially when adjacent symbols are correlated.

Prediction by Partial Matching (PPM) [2, 3] is an adaptive statistical data compression technique based on context modeling and prediction. In general, PPM predicts the probability of a given character based on a given number of characters that immediately precede it. Predictions are usually reduced to symbol rankings. The number of previous symbols, n, determines the order of the PPM model which is denoted as PPM(n). Unbounded variants where the context has no length limitations also exist and are denoted as PPM*. If no prediction can be made based on all n context symbols a prediction is attempted with just n-1 symbols. This process is repeated until a match is found or no more symbols remain in context. At that point a fixed prediction is made. PPM is conceptually simple, but often computationally expensive. Much of the work in optimizing a PPM model is handling inputs that have not already occurred in the input stream. The obvious way to handle them is to create a "never-seen" symbol which triggers the escape sequence. But what probability should be assigned to a symbol that has never been seen? This is called the zero-frequency problem. PPM compression implementations vary greatly in other details. The actual symbol selection is usually recorded using arithmetic coding, though it is also possible to use Huffman encoding or even some type of dictionary coding technique. The underlying model used in most PPM algorithms can also be extended to predict multiple symbols. The symbol size is usually static, typically a single byte, which makes generic handling of any file format easy.

2.2 The LZ family of compressors

Lempel–Ziv compression is a dictionary method based on replacing text substrings by previous occurrences thereof. The dictionary of Lempel–Ziv compression starts in some predetermined state but the contents change during the encoding process, based on the data that has already been encoded. Ziv-Lempel methods are popular for their speed and economy of memory, the two most famous algorithms of this family are called LZ77 [4] and LZ78 [5]. One of the most popular versions of LZ77 is LZSS [6], while one of the most popular versions of LZ78 is LZW [7].

Lempel-Ziv-Storer-Szymanski (LZSS) is a derivative of LZ77, which was created in 1982 by James Storer and Thomas Szymanski. The LZ77 solves the case of no match in the window by outputting an explicit character after each pointer. This solution contains redundancy: either is the null-pointer redundant, or the extra character that could be included in the next match. And in LZ77 the dictionary reference could actually be longer than the string it was replacing. The LZSS algorithm solves this problem in a more efficient manner: the pointer is output only if it points to a match longer than the pointer itself; otherwise, explicit characters are sent. Since the output stream now contains assorted pointers and characters, each of them has to have an extra ID-bit which discriminates between them, LZSS uses one-bit flags to indicate whether the next chunk of data is a literal (byte) or a reference to string.

LZW compression replaces strings of characters with single codes. It does not do any analysis of the incoming text. Instead, LZW builds a string translation table from the text being compressed. The string translation table maps fixed-length codes to strings. The string table is initialized with all single-character strings. Whenever a previously-encountered string is read from the input, the

longest such previously-encountered string is determined, and then the code for this string concatenated with the extension character is stored in the table. The code for this longest previously-encountered string is output and the extension character is used as the beginning of the next word. Compression occurs when a single code is output instead of a string of characters. Although LZW is often explained in the context of compressing text files, it can be used on any type of file. However, it generally performs best on files with repeated substrings, such as text files.

2.3 The Burroughs-Wheeler Transform

Most compression methods operate in the streaming mode, where the codec inputs a byte or several bytes, processes them, and continues until an end-of-file is sensed. The BWT [8] works in a block mode, where the input stream is read block by block and each block is encoded separately as one string. BWT takes a block of data and rearranges it using a sorting algorithm. The resulting output block contains exactly the same data elements that it started with, differing only in their ordering. The transformation is reversible, meaning the original ordering of the data elements can be restored with no loss of information. The BWT is performed on an entire block of data at once. The method is thus also referred to as block sorting.

2.4 Block Compressors

In 2003, Mohammad [9] introduced the concept of block Huffman coding. His main idea is to break the input stream into blocks and compress each block separately. He chooses block size in such a way that he can store one full single block in main memory. He use a block size as moderate as 5 KiB, 10 KiB or 12 KiB. He observed that to obtain better efficiency from his block Huffman coding, a moderate sized block is better and the block size does not depend on file types.

bzip2 [8,10] is a block compression utility, which uses the Burrows-Wheeler transform to convert frequently recurring character sequences into strings of identical letters, and then applies a move-to-front transform and finally Huffman coding. In bzip2 the blocks are generally all the same size in plaintext, which can be selected by a command-line argument between 100KiB–900 KiB. bzip2 is generally considerably better than that achieved by more conventional LZ77/LZ78-based compressors, and approaches the performance of the PPM family of statistical compressors.

Arithmetic coding [11-13] is a form of variable-length entropy encoding that converts a stream of input symbols into another representation that represents frequently used symbols using fewer bits and infrequently used symbols using more bits, with the goal of using fewer bits in total. As opposed to other entropy encoding techniques that separate the input message into its component symbols and replace each symbol with a code word, arithmetic coding encodes the entire message (a single block) into a single number, a fraction n where ($0.0 \leqslant n < 1.0$).

Gzip [14,15] is a software application used for file compression. gzip is based on the DEFLATE algorithm, which is a combination of LZ77 and Huffman coding. In gzip, data which has been compressed by LZSS is broken up into blocks, whose size is a configurable parameter, and each block uses a different compression mode of Huffman coding (Literals or match lengths are compressed with one Huffman tree, and match distances are compressed with another tree.).

3. The block Lossless Data Compression

Our main idea is to break the input into blocks, compress each block separately, and compare the results to determine the optimal block size. One thing we have to consider in this proposed method is the block size. Since the sector size, which is the smallest accessible amount of data on hard disk, is 512 bytes. If we take block size to be less than 1 KiB, there would be a large number of blocks. This will cause penalty in compression ratio and considerably increase the running time. So we use a block size which is greater than 1KiB. We choose block size in such a way that the block size is multiples of 1KiB, but less than or equal to half of test file size.

The select of optimal block size involves trade-offs among various factors, including the degree of compression, the specific application, the block size and the processing time.

3.1 Information entropy

The theoretical background of lossless data compression is provided by information theory. Information theory is based on probability theory and statistics. The most fundamental results of Information theory are Shannon's source coding theorem, which establishes that, on average, the number of bits needed to represent the result of an uncertain event is given by its entropy. Intuitively, entropy quantifies the uncertainty involved when encountering a random variable. From the point of view of data compression, entropy is the amount of information, in bits per byte or symbol, needed to encode it. Shannon's entropy measures the information contained in a message as opposed to the portion of the message that is determined or predictable. The entropy indicates how easily message data can be compressed.

3.2 Locality of reference

Locality of reference, also known as the principle of locality, is one of the cornerstones of computer science. Locality of reference is the phenomenon of the same value or related storage locations being frequently accessed. This phenomenon is widely observed and a rule of thumb, often called the 90/10 rule, states that a program spends 90% of its execution time in only 10% of the code. As far as data compression is concerned, principle of locality is the most recently used data objects is likely to be reused in the near future.

Locality is one of the predictable behaviors that occur to data stream. The data stream exhibits locality in that data distribution is non-uniform in terms of the data objects being used. A consequence of data stream locality is that some objects are used much more frequently than others making these highly used entities attractive compression targets.

Data stream which exhibit strong locality, are good candidates for data compression through the use of some techniques. For example, LZSS uses the built-in implicit assumption that patterns in the input data occur close together. Data streams that don't satisfy this assumption compress poorly. BWT permutes blocks of data to become more compressible by performing a sorting algorithm on the shifted blocks according to its previous context. MTF is a simple heuristic method which exploits the locality of reference.

One possible quantifiable definition for data locality in the spirit of the 90/10 rule is the smaller percentage of used data objects (not necessarily to unique data objects) that are responsible for 90% of compression ratio. Thus, good data locality implies a large skew in the data distribution.

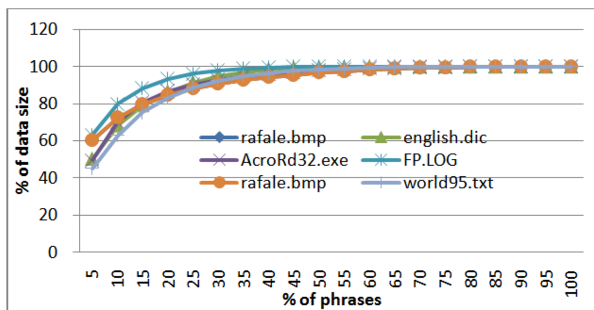

Fig. 1. Data distributions skew in 6 test files.

Figure 1 indicates the fraction of data size (greater than 80%) that is attributable to the most frequently referenced data phrase (less than 20% of phrases and characters). The graphs indicate that compressible data possess significant data distribution locality, as the 90/10 rule predicts. Interestingly, they also indicate that the greater data distribution skew or the more locality, the better the compression ratio.

3.3 Mean Block Standard Deviation (MBSD)

In probability and statistics, the standard deviation is a measure of the dispersion of a collection of values. The standard deviation measures how widely spread the values in a data set are. If many data points are close to the mean, the standard deviation is small; if many data points are far from the mean, then the standard deviation is large. If all data values are equal, then the standard deviation is zero.

We can use the mean block standard deviation (MBSD) to measure the dispersion of byte values. The MBSD is computed by the following formula:

$$\mathbf{MBS} = \frac{1}{b}\sum_{i=1}^{b}\sigma_i \quad (1)$$

$$\sigma_i = \sqrt{\frac{1}{n-1}\sum_{j=1}^{n}\left(y_j - \bar{y}\right)^2} \quad (2)$$

$$y_j = \frac{x_j}{\bar{x}} \quad (3)$$

where $\bar{y}$, $\bar{x}$ is the arithmetic mean of the values x_j and y_j, x_j is defined as the count of byte value j in block i. b is the block number, n is the number of byte value in block i.

Lossless data compression algorithms cannot guarantee compression for all input data sets. Any lossless compression algorithm that makes some files shorter must necessarily make some files longer, to choose an algorithm always means implicitly to select a subset of all files that will become usefully shorter.

There are two kinds of redundancy contained in the data stream: statistics redundancy and non-statistics redundancy. The non-statistics redundancy includes redundancy derived from syntax, semantics and pragmatics. The trick that allows lossless compression algorithms to consistently compress some kind of data to a shorter form is that the data the algorithm are designed to act on all have some form of easily-modeled redundancy that the algorithm is designed to remove. Order-1 statistics-based compressor compress the statistics redundancy, higher orders statistics-based and dictionary-based compression algorithms, which exploit the statistics redundancy and the non-statistics redundancy, are designed to respond to specific types of local redundancy occurring in certain applications.

Table 1. MBSD and Compression Ratio (CR, bpc) of 10 files on optimal blocking mode.

File	Huffman		LZSS		LZW		PPMVC		bzip2	
	MBSD	CR	MBSD	CR	MBSD	CR	MBSD	CR	MBSD	CR
A10.jpg	0.21	8.00	0.21	8.90	0.22	11.08	0.21	7.92	0.21	7.94
AcroRd32.exe	4.32	5.96	3.41	4.51	4.37	5.38	3.69	3.12	3.58	3.51
english.dic	4.33	4.13	3.74	2.85	4.19	3.03	4.19	1.88	4.25	2.11
FlashMX.pdf	0.96	7.58	0.68	7.68	1.01	9.85	0.68	6.59	0.68	6.73
FP.LOG	2.77	5.44	2.76	1.52	2.77	2.02	2.76	0.22	2.76	0.28
MSO97.DLL	2.96	6.43	2.47	5.59	3.04	6.52	2.47	3.98	2.67	4.46
ohs.doc	2.92	6.14	1.84	2.94	2.92	4.06	1.84	1.53	1.87	1.74
rafale.bmp	3.34	4.91	2.52	3.21	3.33	3.25	2.92	1.53	2.92	1.72
vcfiu.hlp	5.50	4.32	4.13	2.63	5.27	3.34	4.45	1.20	4.13	1.38
world95.txt	3.38	5.13	3.34	4.37	3.38	4.40	3.34	1.23	3.34	1.54
enwik8	3.48	5.09	3.42	4.03	3.48	4.56	3.42	1.98	3.42	2.32

It can be seen from table 1, as the values of MBSD increase, the values of the compression ratio also increase. Likewise, as the value of MBSD decreases, the value of the other variable also decreases. There is a positive correlation between MBSD and compression ratio. Thus, we can use the MBSD to measure the compression performance, the greater the MBSD is, and the better the compression ratio will be.

4. Experiment Result

Table 2: Test files used in experiments: raw size in KiB (kibibyte: kilo binary byte)

File	Size	Category	Origin
A10.jpg	823	1152x864 pixels JPEG image file	Maximum Compression Corpus[16]
AcroRd32.exe	3,781	Acrobat Reader 5.0 executable file	
english.dic	3,973	Alphabetically sorted English word-list	
FlashMX.pdf	4,421	Adobe Acrobat document	
FP.LOG	20,134	Fighter-planes.com traffic log file	
MSO97.DLL	3,694	Microsoft Office 97 Dynamic Link Library file	
ohs.doc	4,071	MS Word file	
rafale.bmp	4,053	1356x1020 pixels Bitmap file	
vcfiu.hlp	4,025	Delphi First Impression OCX Windows help file	
world95.txt	2,919	English text file	
enwik8	97,657	Data extracted from the Wikipedia	Hutter Prize[17]

We show in this section our empirical results on four mainstream compression schemes. Our experiments carried out on a small set of test files. Table 2 lists the files used for our experiments.

The Huffman coding、 LZSS and LZW algorithm adapted from the related codes of the data compression book [18]. The PPMVC algorithm adapted from the related codes of Dmitry Shkarin and Przemyslaw Skibinski [19]. The bzip2 coding uses the code of literature [10]. All associated codes were written in the C or C++ language and compiled by Microsoft Visual C++ 6.0. All experiments are performed on a Lenovo computer with an Intel Core2 CPU 4300 @ 1.80GHz & 1.79GHz, 1024 MB memory. The operating system in use is the Microsoft Windows XP.

Experimental results of running the block compression algorithms on our test files are shown in table 3 – 6. The compression ratios for each file shown in the tables are given in bits per character (bpc) or the reduction in size relative to the uncompressed size:

$$\textbf{Compression Ratio} = 1 - \frac{\textbf{Compressed Size}}{\textbf{Uncompressed Size}}$$

Compression Ratio Gain = Blocking Compression Ratio –No Blocking Compression Ratio

The best ratio for each file is printed in boldface.

4.1 Block PPM

PPMVC [15] is a file-to-file compressor. It uses Variable-length Contexts technique, which combines traditional character based PPM with string matching. The PPMVC algorithm, inspired by its predecessors, PPM* and PPMZ, searches for matching sequences in arbitrarily long, variable-length, deterministic contexts. The algorithm significantly improves the compression performance of the character oriented PPM, especially in lower orders (up to 8).

Table 3 and table 4 show results of experiments with block PPMVC algorithm. The experiments were carried out using PPMVC ver. 1.2 (default mode). The results presented in table 3 and 4 indicate that a reduced block size may result in lower compression ratio, and the no blocking mode of PPMVC gives the best compression. From the point of practical use, the optimal block size seems to be greater than 32 KiB.

Table 3. Compression ratio (bpc) in PPMVC for different block sizes (KiB).

File Name	1	2	4	8	16	32	64	128	256	512	1024	Half of file size	No Blocking
A10.jpg	8.65	8.41	8.24	8.14	8.07	8.03	8.02	8.00	7.98	-	-	7.96	**7.92**
AcroRd32.exe	4.31	3.94	3.68	3.49	3.34	3.24	3.18	3.15	3.13	**3.12**	3.12	3.15	3.16
english.dic	2.24	2.03	1.93	1.89	**1.88**	1.89	1.92	1.95	1.99	2.02	2.07	2.11	2.11
FlashMX.pdf	7.50	7.21	7.01	6.89	6.81	6.75	6.68	6.67	6.67	6.65	6.63	6.59	**6.59**
FP.LOG	2.35	1.56	1.10	0.82	0.64	0.51	0.42	0.35	0.31	0.27	0.25	0.22	**0.22**
MSO97.DLL	5.27	4.93	4.69	4.51	4.37	4.28	4.20	4.13	4.09	4.04	4.00	3.99	**3.98**
ohs.doc	3.21	2.75	2.42	2.20	2.05	1.94	1.84	1.75	1.67	1.62	1.60	1.59	**1.53**
rafale.bmp	3.04	2.83	2.66	2.33	2.09	1.92	1.80	1.71	1.64	1.59	1.56	**1.53**	1.53
vcfiu.hlp	2.47	2.12	1.88	1.73	1.64	1.51	1.40	1.32	1.27	1.22	1.20	**1.20**	1.22
world95.txt	4.27	3.83	3.49	3.20	2.78	2.35	2.03	1.79	1.61	1.47	1.37	1.32	1.23
Average	3.47	2.96	2.65	2.42	2.26	2.14	2.05	1.98	1.93	1.90	1.88	1.86	**1.85**

Table 4. Compression Ratio Gain (%) in PPMVC for different block sizes (KiB).

File Name	1	2	4	8	16	32	64	128	256	512	1024	Half of file size	No Blocking
A10.jpg	-9.17	-6.20	-4.09	-2.73	-1.93	-1.46	-1.23	-1.09	-0.79	-	-	-0.52	**1.03**
AcroRd32.exe	-14.31	-9.76	-6.47	-4.04	-2.28	-0.98	-0.24	0.22	0.40	**0.49**	0.48	0.13	60.47
english.dic	-1.53	1.05	2.31	2.79	**2.90**	2.77	2.46	2.03	1.52	1.13	0.51	0.05	73.58
FlashMX.pdf	-11.34	-7.74	-5.28	-3.76	-2.75	-1.93	-1.17	-1.01	-0.99	-0.78	-0.46	-0.02	**17.62**
FP.LOG	-26.58	-16.72	-11.03	-7.53	-5.25	-3.66	-2.51	-1.65	-1.05	-0.64	-0.36	0.00	**97.23**
MSO97.DLL	-16.08	-11.84	-8.84	-6.64	-4.94	-3.73	-2.82	-1.95	-1.43	-0.81	-0.26	-0.09	**50.26**
ohs.doc	-21.07	-15.23	-11.12	-8.36	-6.49	-5.11	-3.90	-2.76	-1.82	-1.12	-0.86	-0.72	**80.89**
rafale.bmp	-18.83	-16.19	-14.04	-9.91	-6.99	-4.88	-3.36	-2.20	-1.37	-0.72	-0.32	**0.01**	80.84
vcfiu.hlp	-15.61	-11.14	-8.20	-6.35	-5.14	-3.59	-2.20	-1.25	-0.53	0.08	0.26	**0.34**	84.70
world95.txt	-38.03	-32.52	-28.22	-24.61	-19.39	-14.04	-10.02	-7.02	-4.74	-3.02	-1.74	-1.05	**84.61**
Average	-20.21	-13.87	-9.92	-7.15	-5.15	-3.59	-2.43	-1.60	-1.02	-0.57	-0.30	-0.10	**76.85**

We believe that the Maximum Corpus files are too small to get a full potential from the block PPMVC

algorithm, so we decided to perform similar experiments on the Hutter Prize corpus. Figure 2 shows the experiment results which were carried out on the test file enwik8. It can be seen from the figure that the compression generally increases as the block size increases, it is similar to the results achieved in the previous experiments.

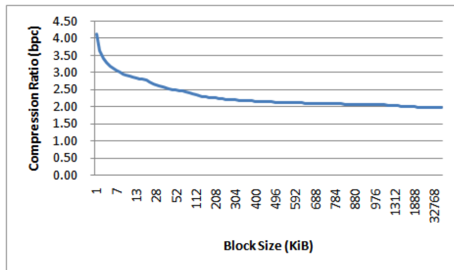

Fig. 2. Compression Ratio of file “enwik8” for different block sizes.

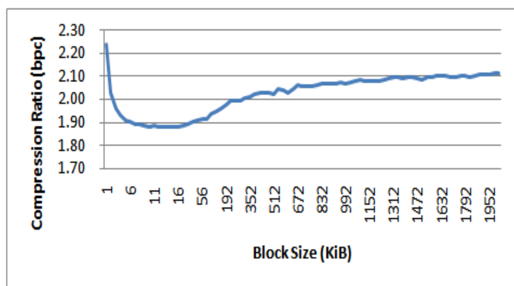

Fig. 3. Compression Ratio of file “english.dic” for different block sizes.

One of the drawbacks with PPM is that it performs relatively poorly at the start. This is because it has not yet built up the counts for the higher order context models, so must resort to lower order models. Figure 3 shows the compression ratio of file English.dic under different block sizes. We can see in the figure that PPM obtains the best performance when the block size is 13 KiB. The poor performance PPM in no blocking mode can be explained by the fact that the english.dic is alphabetically sorted English word-list (354,951 words), PPM must reinitialize for every about 13 KiB-sized block.

4.2 Block Huffman

Table 5 displays the compression ratio gain for block Huffman to original Huffman coding. From the results of Table 5, we have found that, in most cases, the block Huffman coding has a better compression ratio than no blocking Huffman coding, and with the increasing block size, the compression ratio deteriorates. The optimal block size in which it obtains the best compression ratio is more or less 16KiB. The reason for the better efficiency may be attributed to the principle of locality of data. Given this, it can be concluded that the block Huffman coding is advantageous over original no blocking Huffman coding.

Table 5. Compression Ratio Gain (%) in Huffman for different block sizes (KiB).

File Name	1	2	4	8	16	32	64	128	256	512	1024	Half of file size	No Blocking
A10.jpg	-22.98	-11.5	-5.71	-2.81	-1.36	-0.62	-0.25	-0.08	-0.02	-	-	0.00	**0.02**
AcroRd32.exe	-1.40	5.02	8.24	9.51	**9.62**	9.11	7.82	6.63	4.93	3.97	2.83	2.06	19.24
english.dic	1.43	2.52	**2.84**	2.76	2.52	2.16	1.77	1.42	1.10	0.63	0.35	0.10	45.53
FlashMX.pdf	-13.18	-3.75	0.82	2.71	3.39	**3.41**	3.18	2.47	1.64	1.04	0.86	0.14	1.89
FP.LOG	-7.60	-3.48	-1.47	-0.49	-0.04	0.17	0.25	**0.28**	0.27	0.25	0.23	0.09	31.78
MSO97.DLL	-8.77	-1.10	2.73	4.29	**4.69**	4.40	3.62	2.92	2.31	1.87	1.43	0.19	14.96
ohs.doc	-3.13	3.16	6.48	7.88	**8.29**	8.17	7.82	7.39	6.62	5.18	2.49	0.53	14.93
rafale.bmp	8.25	6.26	4.53	5.81	6.43	**6.68**	6.68	6.47	6.01	5.22	4.02	2.88	31.94
vcfiu.hlp	10.18	12.2	**12.97**	12.72	11.81	10.98	9.99	8.65	7.69	6.55	4.36	2.36	33.05
world95.txt	-5.80	-2.64	-1.10	-0.39	0.00	0.21	0.25	**0.26**	0.26	0.19	0.11	0.07	35.59
Average	-4.19	0.16	2.22	3.24	**3.54**	3.51	3.25	2.88	2.45	1.98	1.34	0.68	26.68

4.3 Block LZSS

Table 6 lists the average gain in compression ratio for LZSS of the various block size comparing with the no blocking mode. The index bit count of the LZSS implementation shown here is set to 12 bits and the length bit count macro is set to 4 bits.

Table 6. Compression Ratio Gain (%) in LZSS for different block sizes (KiB).

File Name	1	2	4	8	16	32	64	128	256	512	1024	Half of file size	No Blocking
A10.jpg	-1.24	-0.97	-0.61	-0.30	-0.15	-0.07	-0.03	-0.02	-0.01	-	-	0.00	**-11.20**
AcroRd32.exe	-12.73	-8.58	-4.89	-2.43	-1.24	-0.65	-0.33	-0.16	-0.08	-0.04	-0.02	0.00	**43.59**
english.dic	-4.94	-3.31	-1.90	-0.96	-0.48	-0.24	-0.12	-0.06	-0.03	-0.01	0.00	0.00	**64.38**
FlashMX.pdf	-4.96	-3.32	-1.91	-0.96	-0.53	-0.24	-0.10	-0.04	-0.03	-0.02	-0.02	0.00	**4.00**
FP.LOG	-21.39	-11.89	-6.31	-3.15	-1.57	-0.79	-0.40	-0.20	-0.10	-0.05	-0.03	0.00	**80.98**
MSO97.DLL	-11.81	-8.06	-4.69	-2.34	-1.14	-0.58	-0.30	-0.13	-0.07	-0.04	-0.01	-0.01	**30.19**
ohs.doc	-12.71	-7.98	-4.37	-2.18	-1.11	-0.56	-0.28	-0.13	-0.07	-0.03	-0.01	-0.01	**63.31**
rafale.bmp	-15.59	-13.34	-10.7	-5.34	-2.66	-1.32	-0.68	-0.33	-0.16	-0.07	-0.03	-0.02	**59.90**
vcfiu.hlp	-12.12	-7.70	-4.27	-2.12	-1.06	-0.53	-0.26	-0.13	-0.07	-0.03	-0.01	-0.01	**67.11**
world95.txt	-21.06	-14.87	-8.73	-4.37	-2.19	-1.11	-0.52	-0.26	-0.14	-0.06	-0.03	-0.01	**45.42**
Average	-15.23	-9.46	-5.46	-2.73	-1.36	-0.68	-0.34	-0.17	-0.08	-0.04	-0.02	0.00	**59.24**

We can see from table 6 that no blocking LZSS achieves best results on our test files. The average compression ratio gain of block LZSS is worse by about 15.23% - 0.02% than that of no blocking LZSS. It can be clearly seen in table 6 that the compression ratio gain increases with the increasing size of the block. We can use a moderate block size, so the optimal block size could be 32KiB or so, it is only worse by about 1% compared with the no blocking LZSS.

4.4 Block LZW

The LZW’s advantage over the LZ77-based algorithms is in the speed because there are not that many string comparisons to perform. Due to management of the dictionary, implementation of LZW is somewhat complicated. The code used here is a simple version, which uses twelve-bit codes. Further refinements add variable code word size (depending on the current dictionary size), deleting of the old strings in the dictionary etc.

Table 7. Compression Ratio Gain (%) in LZW for different block sizes (KiB).

File Name	1	2	4	8	16	32	64	128	256	512	1024	Half of file size	No Blocking
A10.jpg	-9.32	-7.68	-4.92	-2.46	-1.14	-0.56	-0.30	-0.21	-0.13	-	-	**0.05**	-38.54
AcroRd32.exe	9.07	15.77	21.94	**26.81**	25.46	21.00	14.93	7.79	1.83	-0.90	-6.06	-6.85	5.98
english.dic	4.64	9.87	14.10	17.52	**19.87**	15.56	11.43	8.15	5.32	4.15	2.61	1.36	42.29
FlashMX.pdf	-2.31	1.25	5.20	**6.46**	4.96	2.36	-0.27	-6.73	-7.19	-7.41	-4.45	-7.13	-29.57
FP.LOG	-52.05	-35.76	-22.61	-12.20	-4.05	0.54	1.65	**1.76**	1.44	1.05	0.58	-0.27	72.99
MSO97.DLL	3.44	9.80	15.77	20.07	18.10	13.79	8.41	5.17	3.46	4.08	3.19	-1.21	-1.60
ohs.doc	54.46	63.47	71.50	78.09	**82.08**	81.80	78.40	76.06	71.82	74.84	62.21	38.13	-32.86
rafale.bmp	22.98	27.1	31.49	39.33	45.06	47.60	**47.70**	45.75	41.72	31.55	10.66	-4.70	11.66
vcfiu.hlp	51.50	58.84	64.81	69.48	**71.35**	65.61	58.91	52.28	46.76	45.12	36.65	5.41	-13.09
world95.txt	-22.67	-13.60	-5.48	1.79	6.82	9.22	9.85	**9.95**	9.28	7.52	6.30	4.97	35.07
Average	-10.49	-0.39	8.18	15.19	19.37	**19.85**	18.17	15.82	13.77	12.58	8.74	2.13	28.04

It can be seen from the table 7 that the block LZW has a better compression ratio than original LZW method. The better efficiency in the compression ratio is the outcome of locality characteristics of the block LZW algorithm as it compresses locally rather than globally. Therefore, it can be concluded that the optimal block size for block LZW is about 32 KiB. The block LZW is advantageous over no blocking LZW coding in compression ratio.

4.5 Block bzip2

Table 8 shows the compression ratio gain in bzip2 for different block size. The bzip2 use the default mode. As can be seen from table 8, in most cases, the no blocking mode of bzip2 can obtain the best compression ratio. One of the exceptions is the test file english.dic which reaches the best efficiency at 10 KiB sized block. The reason for it is the specific distribution characteristic of file english.dic. Thus, it can be concluded that we need to have different compression algorithms for different kinds of files: there cannot be any algorithm that is good for all kinds of data.

Table 8. Compression Ratio Gain (%) in bzip2 for different block sizes (KiB).

File Name	1	2	4	8	16	32	64	128	256	512	1024	Half of file size	No Blocking
A10.jpg	-28.92	-20.19	-12.1	-6.25	-3.06	-1.58	-0.87	-0.50	-0.27	-0.15	-0.15	-0.15	**0.71**
AcroRd32.exe	-22.12	-15.81	-10.4	-5.93	-3.04	-1.23	-0.28	0.18	0.29	0.29	0.01	**0.03**	56.09
english.dic	-1.11	1.76	3.16	**3.68**	3.68	3.28	2.69	1.99	1.28	0.52	0.19	0.12	69.96
FlashMX.pdf	-27.01	-18.88	-11.80	-6.65	-3.66	-2.00	-0.87	-0.51	-0.38	-0.20	-0.15	-0.02	**15.82**
FP.LOG	-36.53	-23.24	-15.37	-10.24	-6.94	-4.60	-2.95	-1.74	-0.9	-0.34	-0.22	-0.02	**96.49**
MSO97.DLL	-25.30	-19.05	-13.33	-8.39	-5.16	-3.23	-2.04	-1.05	-0.54	0.01	**0.11**	-0.20	44.19
ohs.doc	-32.34	-23.39	-16.58	-11.44	-7.69	-5.11	-3.29	-1.85	-0.84	-0.10	-0.17	**0.06**	78.24
rafale.bmp	-18.81	-15.32	-12.62	-8.43	-5.67	-3.73	-2.48	-1.47	-0.78	-0.27	-0.16	**0.01**	78.55
vcfiu.hlp	-19.65	-14.16	-10.60	-8.14	-6.41	-4.52	-2.83	-1.70	-0.84	-0.13	-0.21	-0.14	**82.71**
world95.txt	-41.03	-34.49	-29.37	-25.01	-19.74	-14.33	-9.93	-6.43	-3.67	-1.51	-0.71	-0.47	**80.69**
Average	-28.27	-19.39	-13.39	-8.98	-6.01	-3.89	-2.42	-1.39	-0.70	-0.21	-0.16	-0.05	**74.61**

4.6 Time Efficiency of Block Methods

Figure 4 presents the effect of varying block size on compression time of five lossless compression algorithms. The times include both I/O and compression time, as well as the overhead blocking, open, close, etc.

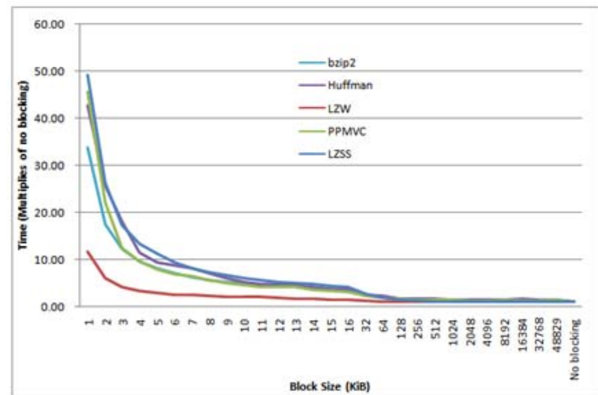

Fig. 4. Compression time and block size for file “enwik8”

As can be seen, for the Huffman coding, LZSS, bzip2 and PPMVC, the compression time decrease steeply when the block size increases from 1 KiB to 4 KiB, then decline steadily from 5 KiB until 64 KiB, and level off as the block size is greater than 64KiB. For LZW algorithm, the compression time declines steadily as the block size increases from 1 to 3 KiB, and then fluctuated slightly. The figures indicate that as the volume of block being compressed grows, compression become increasingly effective in reducing the overall compression time. One reason is due to disk activity. From figure 4, it also can be seen that LZW is the fastest method, because LZW has not that many string comparisons to perform or to build up counts. Therefore, in order to select the optimal block size, we must tradeoff between data compression speed and the amount of compression achieved, a moderate sized block (for example, greater than 32 KiB) may be appropriate.

5. Conclusion

We can conclude from the discussion above, for PPM and LZSS algorithm, a bigger sized block may yield better compression ratios. However, for Huffman coding, BWT and LZW, a moderate sized block is better. The time performance of those block methods and potential improvements to block techniques are also investigated.

In our another paper, "The Block LZSS Compression Algorithm [20]", we have studied the block LZSS algorithm and investigated the relationship between the compression ratio of block LZSS and the value of index or length. We found that as the block size increases, the compression ratio becomes better. We also found that the bit of length has little effect on the compression performance, and the bit of index has a significant effect on the compression ratio. We showed that the more the bit of index is set, the bigger optimal block size is obtained.

We can conclude from table 6 that to obtain better efficiency from block LZSS, a moderate sized block

which is greater than 32KiB, may be optimal, and the optimal block size is not depend on file types.

From the results of table 4 and table 7, we have found that, in most cases, the block coding methods of Huffman and LZW have better compression ratio than original Huffman coding and LZW, and with the increasing block size, the compression ratio deteriorates. The optimal block size of Huffman coding is about 16KiB, and LZW obtain the best compression ratio in about 32KiB. The reason may be attributed to the principle of locality of data.

We also studied the blocking algorithm of bzip2. We found that block bzip2 is similar to the block PPM in that the compression efficiency is increased with the increasing block size.

References

[1] D.A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of the I.R.E., September 1952, pp 1098-1102.
[2] T. Bell, J. Cleary, and I. Witten, "Data compression using adaptive coding and partial string matching," IEEE Transactions on Communications, Vol. 32 (4), p. 396-402, 1984.
[3] A. Moffat, Implementing the PPM data compression scheme , IEEE Transactions on Communications, Vol. 38 (11), pp. 1917-1921, November 1990.
[4] Ziv, J., & Lempel, A. "A Universal Algorithm for Sequential Data Compression," IEEE Transactions on Information Theory, 23(3), pp.337-343, May 1977.
[5] Ziv, J., & Lempel, A. "Compression of individual sequences via variable-rate coding," IEEE Trans. Inform. Theory, 24(5), 530-536, September 1978.
[6] Storer, J.A., & Szymanski, T.G. "Data Compression via Textual Substitution," Journal of ACM, 29(4), 928-951, 1982.
[7] Welch, T.A. "A technique for high performance data compression", IEEE Computer, 17(6), 819, 1984.
[8] M. Burrows and D. J. Wheeler, "A Block-sorting Lossless Data Compression Algorithm", Digital Systems Research Canter Research Report 124, May 1994.
[9] M. Mannan, M. Kaykobad, "Block Huffman Coding", International Journal of Computers and Mathematics with Applications, vol 46, issue 10-11, pp 1581-1587, November - December 2003.
[10] bzip2: http://www.bzip.org/
[11] Rissanen, Jorma. "Generalized Kraft Inequality and Arithmetic Coding". IBM Journal of Research and Development 20 (3): 198–203, May 1976.
[12] Rissanen, J.J.; Langdon, G.G., Jr. "Arithmetic coding". IBM Journal of Research and Development 23 (2): 149–162, March 1979.
[13] Witten, Ian H.; Neal, Radford M.; Cleary, John G. "Arithmetic Coding for Data Compression". CACM 30 (6): 520–540, June 1987.
[14] gzip: http://www.gzip.org/, 18, Jun. 2008.
[15] P. Deutsch, "RFC1951: DEFLATE Compressed Data Format Specification version 1.3", May 1996.
[16] http://www.maximumcompression.com/index.html
[17] http://prize.hutter1.net/index.htm
[18] Nelson M., Gailly J., "The Data Compression Book, 2nd edition", M&T Books, New York, NY, 1995.
[19] Homepage of Przemysław Skibiński: http://www.ii.uni.wroc.pl/~inikep/.
[20] Wei-ling Chang, Xiao-chun Yun, Bin-xing Fang, Shu-peng Wang. The Block LZSS Compression Algorithm. Data Compression Conference (DCC2009): 439-439, March 2009

Weiling Chang was born in Shanxi province, China. He graduated with a BA Econ from University of International Business and Economics (UIBE) in 1993 and earned his master's degree in computer science from China Agricultural University (CAU) in 2006. He is currently in PhD program in Computer Science at Harbin Institute of Technology (HIT), Harbin, China. His major research interests include data compression, computer network and information security.

Binxing Fang, born in 1960, is a professor and supervisor of Ph.D. candidates, an academician of Chinese Academy of Engineering and the president of Beijing University of Post and Telecommunications. His research interests include computer network and information security.

Xiaochun Yun, born in 1971, is a professor and Ph.D. supervisor at Institute of Computing Technology of the Chinese Academy of Sciences. His research interests include computer network and information security.

Shupeng Wang, born in 1980, Ph.D. His research interests include computer network and information security.