Compression of Concatenated Web Pages Using XBW*

Radovan Šesták and Jan Lánský

Charles University, Faculty of Mathematics and Physics,
Department of Software Engineering
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
radofan@gmail.com, zizelevak@gmail.com

Abstract. XBW [10] is modular program for lossless compression that enables testing various combinations of algorithms. We obtained best results with XML parser creating dictionary of syllables or words combined with Burrows-Wheeler transform - hence the name XBW. The motivation for creating parser that handles non-valid XML and HTML files, has been system EGOTHOR [5] for full-text searching. On files of size approximately 20MB, formed by hundreds of web pages, we achieved twice the compression ratio of bzip2 while running only twice as long. For smaller files, XBW has very good results, compared with other programs, especially for languages with rich morphology such as Slovak or German. For any big textual files, our program has good balance of compression and run time.

Program XBW enables use of parser and coder with any implemented algorithm for compression. We have implemented Burrows-Wheeler transform which together with MTF and RLE forms block compression, dictionary methods LZC and LZSS, and finally statistical method PPM. Coder offers choice of Huffman and arithmetic coding.

1 Introduction

In this article we list results of compression of big XML files. Motivation for this work has been compression of data from web. Speed is very important for full-text searching and hence compression is not always the best choice. On the other hand archiving of old versions of web pages requires vast amount of space on disk. Also this data is not often used and hence compression could help with insufficient dist space. XML format is very redundant and therefore very good compression ratio can be achieved. Furthermore related web pages, with regard to its origin, contain long sequences of identical data. These properties of data enabled us to compress the data to tenth of original size.

We used for testing XML files from system *EGOTHOR* [5]. These files have size around 20MB and were formed by concatenation of hundreds of web pages and contain lots of text. Big files can be compressed more effectively due to

* This work was supported by Charles University Grant Agentur in the project "Text compression" (GAUK no. 1607, section A) and by the Program "Information Society" under project 1ET100300419.

V. Geffert et al. (Eds.): SOFSEM 2008, LNCS 4910, pp. 743–754, 2008.
© Springer-Verlag Berlin Heidelberg 2008

following reasons. With the use of dictionary there is better ratio of size of dictionary to size of file. And most importantly entropy of text files is decreasing which means that following characters can be better predicted. Problematic is the fact that these files do not have valid XML structure and often they are not well formed. This lead us to creating our own parser since the parsers we know of could not handle these files.

In the next section we describe parts of program XBW and their influence on compression. Then we list results of measurements and comparison with common compression programs.

1.1 Conflicting Name XBW

The XBW method was not named very appropriately, because it can be easily mistaken by the name xbw used by the authors of paper [4] for XML transformation into the format more suitable for searching. In another article these authors renamed the transformation from xbw to XBW. Moreover they used it in compression methods called XBzip and XBzipIndex.

Another confusion comes from the fact that originally XBW stood in our articles for combination of methods BWT+MTF+RLE using alphabet of syllables for valid XML files. In what we know present as XBW the main idea of compression XML using BWT is present, but otherwise significantly differs from original work.

2 Implemented Methods

In Figure 1 the connections of parts of program is shown. All parts are optional. From algorithms RLE, LZC, LZSS and PPM at most one can be used because all of these algorithms use coder. In XBW Huffman (HC) and arithmetic coder (AC) are implemented. We have obtained the best results with combination of Parser + BWT + MTF + RLE + AC, which is used as default settings of the program.

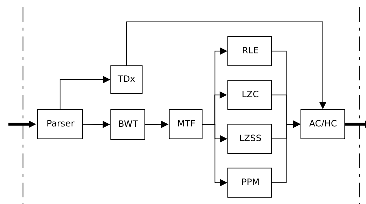

Fig. 1. XBW architecture

2.1 Parser and TDx

Implemented parser uses syntax of XML files for shortening the output. It omits closing characters and dynamically creates dictionary of tags and attributes. The rest of the data is split into characters, syllables or words and these are added to

trie. This choice is one of the parameters of parser. The use of words as alphabet is quite common for text compression, but its use for XML is not. The splitting into words or syllables and coding of dictionary is based on work of Lnsk [11].

It is also possible to use parser in text mode in which the structure of XML files is not taken into account and the input is only split into symbols of chosen alphabet (characters, syllables or words). Another parameter for parser is the choice of file coding; supported are dozens of codings for which we used library *iconv* [16].

Dictionary, that is stored in memory in form of trie during processing the file, is saved to output by coding its structure using methods TD1, TD2 or TD3 [12]. In method TD1 for each node we code the distance from left son, number of sons and boolean value which determines if the node represents a string. Methods TD2 and TD3 take advantage of the fact that we are coding syllables and words from natural languages which satisfy special conditions and hence better compression is achieved. Default method is TD3.

2.2 Block Compression

The class of methods for compression based on work of Burrows and Wheeler [1], uses reversible transformation which is called Burrows-Wheeler Transform (BWT). Often the combination of this transformation together with following effective coding is called block compression. The name comes from the fact that the input is split into block of fixed size and BWT is performed for each block. In default settings of our implementation we combine BWT with algorithms MTF and RLE. Decoding is significantly faster for BWT than coding.

Very important parameter of BWT is size of block to used. Larger blocks have better compression ratio at cost of longer run time and higher memory requirements. For example, in bzip2 algorithm, the maximum block size is 900 KB. In our program we use default setting of 50MB which can be changed.

Algorithm MTF used after BWT renumbers input and the result is string which contains relatively small numbers and sequences of zero. We implemented MTF using splay tree [7] which improves its speed especially for large alphabet. After MTF algorithm RLE is run which outputs the character and number of its occurrences. This is written in form of bits using coder.

We have RLE in three variants RLE1, RLE2 and RLE3 using RLE2 as default. In variant RLE1 sequence of repeating characters is replaced by three symbols. First one is escape sequence followed the character and number of occurrences. Alphabet used is increased by this escape symbol. In variant RLE2 each character there is also special escape symbol hence the resulting alphabet is twice the size of original. Repeating sequence of characters is replaced by escape symbol of the character and number of occurrences. In variant RLE3 we add to alphabet for each character special symbol for each possible number of repeatings of the character. We limit the length of sequence by fixed number. Hence in output we replace each sequence by special symbol. This method is suitable for small alphabets with small limit to length of sequence.

2.3 Dictionary Methods

Dictionary compression methods are usually adaptive; during coding they update dictionary of phrases. During compression we search for longest match of uncoded part of input with some phrase from dictionary. These methods work especially well for inputs where short sequences are (max 5-15 characters) repeating which is true for textual data. These algorithms are frequently used, because they are relatively fast and use little memory. When these algorithms are used, XBW has similar results as Gzip which also uses dictionary methods.

There are two main types of methods one based on LZ77 [17] and on LZ78 [18]. Method LZ77 has dictionary represented by compressed part of document in form of sliding window of fixed size which is moving right during compression. Method LZ78 builds the dictionary explicitly. In each step of compression a phrase from dictionary used for coding is lengthened by one character which follows after this phrase in uncompressed part of input.

LZC. LZC [6] is an improved version of LZ78 which uses trie data structure for dictionary and the phrases are numbered by integers in order of adding. During initialization, the dictionary is filled with all characters from the alphabet. In each step we search for maximal string S in dictionary being a prefix of non-coded part of input. The number of phrase S is then sent to the output. Actual input position is moved forward by the length of S. If the compression ratio starts to get worse the dictionary is cleaned. During decoding, if we get number of phrase that is in the dictionary we output the phrase. If the number does not stand for any phrase in the dictionary we can create that phrase by a concatenation of the last added phrase with its first character.

LZSS. LZSS [15] is improved version of LZ77 where the dictionary is represented by sliding window which we shift to the right during compression. We search for the longest prefix of non-coded input which matches string S from sliding window. LZ77 outputs always ordered triple $< D, L, N >$, where D is the distance of found string S in sliding window from its beginning, L is the length of S and N is the symbol following S in non-coded part of input. LZ77 has the disadvantage that if no match is found, hence S has zero length, output has been unnecessarily long. So in LZSS minimal threshold value is used when to code string S using D and L. If it is shorter, we code it as sequence of characters N. We use one bit to signal if the match for S has been long enough. So the output in each step is either $< 0, D, L >$ or several tuples $< 1, N >$.

2.4 PPM Method

The newest implemented method is statistical method called Prediction by Partial Matching (PPM) [2], which codes characters based on their probability after some context. Probabilities of characters after contexts are counted dynamically. This method, designed for compression of texts in natural languages, is quite slow and requires lots of memory. We have implemented variants PPMA, PPMB, PPMC with optional exclusions and setting for maximal context.

2.5 Coder

Final bit output is provided by coder. We implemented Huffman and arithmetic coder. Both variants are implemented in static and adaptive version. Arithmetic coder uses Moffat data structure and Huffman coder is implemented in canonic version. The choice of either Huffman or arithmetic coder is given at compile time. Default is the arithmetic coder, because it yield slightly better compression ratio than Huffman, and it is significantly faster when adaptive versions are used.

Huffman coding is compression method which assigns symbols codes of fixed length. It generates optimal prefix code which means that no code is a prefix of another code. Codes of all symbols are in binary tree and the edges have values 0 and 1. The code for symbol is given by the path from root to node representing the symbol. In static version we know the frequencies of symbols before construction of the tree. We sort symbols by their frequencies. In each step we take two symbols A and B with the smallest frequencies and create new one C which has frequency the sum of A and B they are his sons. In adaptive version we start with tree with one node representing escape symbol which is used for insertion of unencountered symbols. When adding node for unencountered symbol we create node A for this symbol and node B for escape symbol. Both have frequency one C representing original escape sequence is their father. When we increase frequency for symbol we first move its node to the right of nodes with equal frequencies. Then we increase its frequency and recursively repeat this step on its father.

The idea of arithmetic coding is to represent input as number from interval $[0, 1)$. This interval is divided into parts which stand for probability of occurrence of symbols. Compression works by specifying the interval. In each step interval is replaced by subinterval representing symbols of alphabet. Since the arithmetic coding does not assign symbols codes of fixed length, arithmetic coding is more effective than Huffman coding.

3 BWT

We describe Burrows-Wheeler transform in more detail, because its use in optimized version with input modified by parser allowed to get results we present. BWT during coding requires lexicographical order of all suffixes. The resulting string has on i-th place last symbol of i-th suffix. We assume that we have linear order on set of symbols Σ, which we call alphabet. Symbol in this sense can be character, syllable or word.

$X \equiv x_0x_1...x_{n-1}$, $\forall i \in \{0, .., n-1\}$, $x_i \in \Sigma$ is string of length n. i-th *suffix* of string X is string $S_i = x_ix_{i+1}..x_{n-1} = X[i..n-1]$. i-th suffix is smaller than j-th suffix, if first symbol in which they differ, is smaller, or i-th suffix is shorter. $S_i < S_j \iff \exists k \in 0..n-1 : S_i[0..k-1] = S_j[0..k-1]$ & $(S_i[k] < S_j[k] \vee (i+k = n$ & $j+k < n))$. We store the order of suffixes in *suffix array* SA, for which the following holds: $\forall i, j \in \{0..n-1\}$, $i < j \rightarrow S_{SA[i]} \leq S_{SA[j]}$.

The result of BWT for string X is $\tilde{X}$. $\tilde{X} \equiv \tilde{x}_0..\tilde{x}_{n-1}$ where $\tilde{x}_i = x_{|SA[i]-1|_n}$. Absolute values stand for modulo n, which is necessary in case $SA[i] = 0$.

Repetitiveness of the file influences the compression ration and run time of coding phase of BWT. We denote longest common prefix or match length by $lcp(S_i, S_j) = max\{k; S_i[0..k-1] = S_j[0..k-1]\}$. *Average match length* $AML \equiv \frac{1}{n-1}\sum_{i=0}^{n-2} lcp(S_{SA[i]}, S_{SA[i+1]})$ is the value we use in text for measuring repetitiveness of files.

We have implemented a few algorithms for sorting suffixes with different asymptotic complexity. The fastest algorithm for not too much repetitive files is ($AML <$ 1000) is Kao's modification of Itoh algorithm [9], which has time complexity $O(AML{\cdot}n{\cdot}log\, n)$. For very repetitive files algorithm due to Krkkainen and Sanders [8] with complexity $O(n)$. Note that the choice of algorithm for BWT does not influence compression ratio, but only time and memory requirements.

In block compression the file is split into blocks of fixed size and BWT is run on each block. This method is used to decrease memory and time requirements, because BWT requires memory linear in size of input. Time complexity of most of algorithms for BWT is asymptotically super linear and BWT is the slowest part of block compression during compression. However, use of smaller blocks worsens the compression ratio.

The main reason why XBW has significantly better compression ration than bzip2 is the use of BWT on whole file at once. Our program runs in reasonable time thanks to preprocessing of the input by parser and the use of alphabet of words, which shortens the input for BWT. Very important consequence of the use of parser is the decrease of the value of AML. However use of parser requires use of algorithms, which do not work with byte alphabet of size 256 characters, but can work with 4 byte alphabet. When words are used as alphabet, for the tested files, the size of alphabet created by parser is approximately 50 thousand.

4 Corpora

Our corpus is formed by three files which come from search engine *EGOTHOR*. The first one is formed by web pages in Czech, the second in English and third in Slovenian. Their size in this order are: 24MB, 15MB, 21MB and the values of AML, describing their repetitiveness, are approximately 2000. Information about compression ratio of XBW on standard corpora Calgary, Cantebury and Silesia can be found in [10].

5 Results

First we list results of program XBW for various compression methods and the effect of parser on the results. Then we show influence of alphabet. At the end we compare results of XBW using optimal parameters with commonly used programs Gzip, Rar and Bzip2.

All results have been obtained using arithmetic coder. BWT has been run over whole input at once followed by MTF and RLE (parameter RLE=2). PPM has run with parameters PPM_ exlusions=off a PPM_order=5.

The size of compressed files includes coded dictionary which is created always when parser is used. Compression ratio is listed in bits per byte.

The run time has been measured under Linux and stands for sum of system and user time. This implies that we list time without waiting for disk. Measurements has been performed on PC with processor AMD Athlon X2 4200+ with 2GB of RAM. The data is in megabytes in second where the uncompressed size of file is used both for compression and decompression.

Table 1 shows the results of compression ratio for various methods for alphabet of characters. These results show effect of XML, which improves the compression ratio by approximately ten percent.

Next in Tables 2 and 3 we list the speed of program with and without parser using alphabet of characters. Results show that in almost all cases the parser degrades the speed. The reason is that we have to work with dictionary and the time saved by slightly shortening the input does not compensate for the work with dictionary. The exception is compression using BWT. Here the shortening the input and decreasing its repetitiveness significantly fastens BWT, which is the most demanding part of block compression.

Method commonly used for text compression is the use of words as symbols of alphabet. In Table 4 we show the influence of alphabet on compression ratio. For textual data in English best compression ratio is achieved with using words and method BWT. For Czech and Slovenian the syllables are better, because these languages have rich morphology. One word occurs in text in different forms and each form is added into dictionary. With the use of syllables core of the word is added, which can be formed by more syllables, and the end of word. But these last syllables of words are common for many words and hence there are more occurrences of then in the text. For dictionary methods LZx the words are by far the best choice.

The effect of large alphabet on speed varies and is shown in Tables 5 and 6. For all algorithms the decompression is faster for words than for characters. On the other hand decompression when parser with words has been used is still slower than decompression without parser see Table 1. The use of words increases the speed of compression only when BWT is used. Significant increase in speed for BWT is due to shortening the input and decreasing approximately three times AML. Results for PPM and words are not shown since the program did not finish within hour.

Previous results show that the best compression ratio has the algorithm BWT. Also it is evident that parser improves compression ratio for all algorithms. The fastest compression is achieved using LZC and fastest decompression using LZSS.

Our primary criterion is compression ratio and since method BWT has by far the best compression ratio, we focus mainly on BWT. In case the speed is priority choice of dictionary methods is advisable.

Table 7 contains comparison of compression ratios for different choices of parser, which shows that words are best for English and syllables for Czech and Slovenian. The choice of either words or syllables depends on the size of file and on morphology of the language. For languages with rich morphology,

Table 1. Influence of parser on compression ratio for alphabet of symbols

bpB	No Parser				Text Parser				XML Parser			
	BWT	LZC	LZSS	PPM	BWT	LZC	LZSS	PPM	BWT	LZC	LZSS	PPM
xml_cz	0,907	2,217	2,322	1,399	0,906	2,206	2,296	1,395	**0,894**	2,073	2,098	1,320
xml_en	0,886	2,044	2,321	1,292	0,887	2,044	2,321	1,292	**0,874**	1,915	2,115	1,239
xml_sl	0,710	1,982	2,010	1,205	0,710	1,979	2,003	1,204	**0,700**	1,850	1,797	1,129
TOTAL	0,834	2,093	2,213	1,305	0,833	2,087	2,200	1,303	**0,822**	1,957	1,998	1,234

Table 2. Influence of parser on compression speed

MB/s	No Parser				XML Parser - Symbols			
	BWT	LZC	LZSS	PPM	BWT	LZC	LZSS	PPM
xml_cz	0,368	3,587	1,498	0,106	0,457	2,668	1,418	0,091
xml_en	0,419	4,028	1,297	0,125	0,544	2,915	1,249	0,104
xml_sl	0,386	4,258	1,638	0,119	0,500	2,915	1,497	0,091
TOTAL	0,386	3,906	1,485	0,115	0,491	2,810	1,397	0,094

Table 3. Influence of parser on decompression speed

MB/s	No Parser				XML Parser - Symbols			
	BWT	LZC	LZSS	PPM	BWT	LZC	LZSS	PPM
xml_cz	4,260	4,724	5,257	0,117	2,577	3,156	3,415	0,096
xml_en	4,417	4,999	5,417	0,142	2,705	3,397	3,606	0,110
xml_sl	4,918	5,236	5,946	0,134	3,012	3,299	3,672	0,097
TOTAL	4,509	4,960	5,519	0,128	2,747	3,263	3,548	0,099

Table 4. Influence of alphabet on compression ratio

bpB	XML Parser											
	Symbols				Syllables				Words			
	BWT	LZC	LZSS	PPM	BWT	LZC	LZSS	PPM	BWT	LZC	LZSS	PPM
xml_cz	0,894	2,073	2,098	1,320	0,854	1,796	1,841	N/A	0,857	1,683	1,654	N/A
xml_en	0,874	1,915	2,115	1,239	0,836	1,626	1,785	N/A	0,830	1,514	1,558	N/A
xml_sl	0,700	1,850	1,797	1,129	0,664	1,559	1,541	N/A	0,668	1,457	1,390	N/A
TOTAL	0,822	1,957	1,998	1,234	0,783	1,672	1,723	N/A	0,785	1,563	1,539	N/A

Table 5. Influence of Alphabet on compression speed

MB/s	XML Parser - Symbols				XML Parser - Words			
	BWT	LZC	LZSS	PPM	BWT	LZC	LZSS	PPM
xml_cz	0,457	2,668	1,418	0,091	1,587	0,279	1,477	N/A
xml_en	0,544	2,915	1,249	0,104	2,009	0,920	1,093	N/A
xml_sl	0,500	2,915	1,497	0,091	1,566	0,443	1,349	N/A
TOTAL	0,491	2,810	1,397	0,094	1,666	0,399	1,319	N/A

Table 6. Influence of Alphabet on decompression speed

MB/s	XML Parser - Symbols				XML Parser - Words			
	BWT	LZC	LZSS	PPM	BWT	LZC	LZSS	PPM
xml_cz	2,577	3,156	3,415	0,096	3,986	3,951	3,923	N/A
xml_en	2,705	3,397	3,606	0,110	4,006	4,443	4,523	N/A
xml_sl	3,012	3,299	3,672	0,097	4,241	4,157	4,237	N/A
TOTAL	2,747	3,263	3,548	0,099	4,076	4,135	4,167	N/A

Table 7. Compression ratio for BWT

bpB	No Parser	Text Parser			XML Parser		
		Symbols	Syllables	Words	Symbols	Syllables	Words
xml_cz	0,907	0,906	0,859	0,862	0,894	**0,854**	0,857
xml_en	0,886	0,887	0,842	0,836	0,874	0,836	**0,830**
xml_sl	0,710	0,710	0,669	0,672	0,700	**0,664**	0,668
TOTAL	0,834	0,833	0,789	0,790	0,822	**0,783**	0,785

Table 8. Compression speed for BWT

MB/s	No Parser	Text Parser			XML Parser		
		Symbols	Syllables	Words	Symbols	Syllables	Words
xml_cz	0,368	0,324	1,056	**1,767**	0,457	1,073	1,587
xml_en	0,419	0,364	1,225	**2,128**	0,544	1,330	2,009
xml_sl	0,386	0,331	1,102	**1,790**	0,500	1,135	1,566
TOTAL	0,386	0,336	1,110	**1,853**	0,491	1,150	1,666

Table 9. Decompression speed for BWT

MB/s	No Parser	Text Parser			XML Parser		
		Symbols	Syllables	Words	Symbols	Syllables	Words
xml_cz	4,260	2,628	4,277	**4,817**	2,577	3,710	3,986
xml_en	4,417	2,494	4,612	**4,764**	2,705	3,981	4,006
xml_sl	4,918	2,639	4,686	**5,442**	3,012	3,685	4,241
TOTAL	4,509	2,598	4,494	**5,002**	2,747	3,765	4,076

Table 10. Running time for different parts of XBW

Seconds	Compression				Decompression			
	Parser	BWT	MTF	RLE	Parser	BWT	MTF	RLE
xml_cz	4,668	7,788	0,748	0,72	1,98	0,764	0,868	1,328
xml_en	2,364	3,800	0,388	0,448	1,112	0,716	0,440	0,796
xml_sl	3,352	7,404	0,496	0,504	1,592	0,676	0,556	0,916
TOTAL	10,384	18,992	1,632	1,672	4,684	2,156	1,864	3,04

Parser in text mode using words; BWT using Itoh; RLE - version 3

Table 11. Comparison of compression ratio

bpB	XBW	Gzip	Bzip2	Rar
xml_cz	**0,857**	1,697	1,406	1,161
xml_en	**0,830**	1,664	1,299	0,851
xml_sl	**0,668**	1,373	1,126	0,912
TOTAL	**0,785**	1,584	1,275	0,998

XBW: parser in text mode using words, Kao's algoritm for BWT
Gzip: gzip -9; Rar: rar -m5; Bzip2: bzip2 -9

Table 12. Comparison of compression speed

MB/s	Compression				Decompression			
	XBW	Gzip	Bzip2	Rar	XBW	Gzip	Bzip2	Rar
xml_cz	1,732	**10,320**	3,170	2,708	4,087	**25,004**	9,430	3,955
xml_en	2,058	**11,587**	3,454	2,689	4,309	**46,926**	11,722	6,137
xml_sl	1,758	**13,713**	3,245	3,190	4,614	**46,986**	13,132	4,775
TOTAL	1,812	**11,634**	3,262	2,853	4,313	**34,629**	11,045	4,640

XBW: parser in text mode using words, Kao's algoritm for BWT
Gzip: gzip -9; Rar: rar -m5; Bzip2: bzip2 -9

and for smaller files the syllables are better. Choice of either words or syllables effects the number of occurrences of symbols from dictionary for the input text. In program XBW we have implemented a few methods for splitting words into syllables. Results have been obtained using the choice *Left*. More details can be found in [10]. Interesting is the fact that the XML mode of parser has small influence on compression ratio. This is not due to incorrect implementation of parser, but due to properties of BWT for large blocks. For example for LZx methods the effect is significant. Again more detailed results are in [10].

Table 8 show the influence of parser on the speed of program. The fastest by far is the choice of words as symbols of alphabet for compression. For decompression (see table 9) the differences are small. In order to improve the speed it is better to use parser in text mode instead of XML mode for words.

There are many algorithms for sorting suffixes in BWT. The choice of this algorithm has big impact of overall performance of compression. Without the use of parser, sorting suffixes for big blocks amount up 90% of run time of whole program. More details are in [14]. For all files the fastest is Kao's modification of Itoh's algorithm [9] and it has been used in all measurements when BWT has been used.

Run time of separate parts of program are in Table 10. These times show in which parts there is the most room for improvement.

6 Comparison with Other Programs

For comparison we show the results of programs Gzip, Rar and Bzip2. Programs for compression of XML data such as XMLPPM [3] and Xmill [13] can not cope

with non-valid XML files. Hence we could not get their results on our data. For programs Gzip, Rar and Bzip2 we used parameters for the best available compression. In Table 11 we list compression ratios. Our program compresses all files the best and is significantly better for files which are not in English.

In Table 12 are the results for speed of compression and decompression. The fastest is Gzip, but it also has the worst compression ratio and hence we compare speed of XBW only with Rar and Bzip2. Compression for XBW takes less twice the minimum of Rar and Bzip2. Decompression if comparably fast as for Rar and Bzip2 is approximately three times faster.

The performance of XBW is sufficient for common use, however it is slower than the speed of hard disks, and hence where speed is priority, it is better to use program based on dictionary methods such as Gzip. XBW has the best compression ratio and therefore it is suitable especially for long term archiving.

7 Future Work

In future work on XBW we aim to focus on two directions. The first is creation of parser which could be used also on binary data. The later is improving the run time of program where again we see the biggest potential in parser.

References

1. Burrows, M., Wheeler, D.J.: A Block Sorting Loseless Data Compression Algorithm. Technical report, Digital Equipment Corporation, Palo Alto, CA, U.S.A (2003)
2. Cleary, J.G., Witten, I.H.: Data compression using adaptive coding and partial string matching. IEEE Transactions on Communications COM-32(4), 396–402 (1984)
3. Cheney, J.: Compressing XML with Multiplexed Hierarchical PPM Models. In: Storer, J.A., Cohn, M. (eds.) Proceedings of 2001 IEEE Data Compression Conference, p. 163. IEEE Computer Society Press, Los Alamitos, California, USA (2001)
4. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Structuring labeled trees for optimal succinctness, and beyond. In: FOCS 2005. Proc. 46th Annual IEEE Symposium on Foundations of Computer Science, pp. 184–193 (2005)
5. Galamboš, L.: EGOTHOR, `http://www.egothor.org/`
6. Horspool, R.N.: Improving LZW. In: Storer, J.A., Reif, J.H. (eds.) Proceedings of 1991 IEEE Data Compression Conference, pp. 332–341. IEEE Computer Society Press, Los Alamitos, California, USA (1991)
7. Jones, D.W.: Application of splay trees to data compression. Communications of the ACM 31(8), 996–1007 (1988)
8. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 943–955. Springer, Heidelberg (2003)
9. Kao, T.H.: Improving suffix-array construction algorithms with applications. Master Thesis. Gunma University, Japan (2001)
10. Lánský, J., Šesták, R., Uzel, P., Kovalčin, S., Kumičák, P., Urban, T., Szabó, M.: XBW - Word-based compression of non-valid XML documents, `http://xbw.sourceforge.net/`

11. Lánský, J., Žemlička, M.: Compression of a Dictionary. In: Snášel, V., Richta, K., Pokorný, J. (eds.) Proceedings of the Dateso 2006 Annual International Workshop on DAtabases, TExts, Specifications and Objects. CEUR-WS, vol. 176, pp. 11–20 (2006)
12. Lánský, J., Žemlička, M.: Compression of a Set of Strings. In: Storer, J.A., Marcellin, M.W. (eds.) Proceedings of 2007 IEEE Data Compression Conference, p. 390. IEEE Computer Society Press, Los Alamitos, California, USA (2007)
13. Liefke, H., Suciu, D.: XMill: an Efficient Compressor for XML Data. In: Proceedings of ACM SIGMOD Conference, pp. 153–164 (2000)
14. Šesták, R.: Suffix Arrays for Large Alphabet. Master Thesis, Charles University in Prag (2007)
15. Storer, J., Szymanski, T.G.: Data compression via textual substitution. Journal of the ACM 29, 928–951 (1982)
16. The Open Group Base: iconv. Specifications Issue 6. IEEE Std 1003.1 (2004), `http://www.gnu.org/software/libiconv/`
17. Ziv, J., Lempel, A.: A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory 23(3), 337–342 (1977)
18. Ziv, J., Lempel, A.: Compression of Individual Sequences via Variable-Rate Coding. IEEE Transactions on Information Theory 24(5), 530–536 (1978)