Post BWT Stages of the Burrows-Wheeler Compression Algorithm

Jürgen Abel

Ingenieurbüro Dr. Abel GmbH
Lechstrasse 1
41469 Neuss – GERMANY
Email: juergen.abel@data-compression.info

The lossless Burrows-Wheeler compression algorithm has received considerable attention over recent years for both its simplicity and effectiveness. It is based on a permutation of the input sequence – the Burrows-Wheeler transformation – which groups symbols with a similar context close together. In the original version, this permutation was followed by a Move-To-Front transformation and a final entropy coding stage. Later versions used different algorithms, placed after the Burrows-Wheeler transformation, since the following stages have a significant influence on the compression rate. This article describes different algorithms and improvements for these post BWT stages including a new context based approach. Results for compression rates are presented together with compression and decompression times on the Calgary corpus, the Canterbury corpus, the large Canterbury corpus and the Lukas 2D 16 bit medical image corpus.

Keywords: compression, Burrows-Wheeler transformation, block sorting

1. Introduction

First, a historical overview of the Burrows-Wheeler Compression Algorithm (BWCA) is presented; the basic concepts of the different parts of the algorithm are explained in the next sections.
The family of the block sorting algorithms based on the Burrows-Wheeler Transformation (BWT) has grown over the past few years starting with the first implementation described by Burrows and Wheeler in 1994 [1] and BWT based compression found acceptance in the LINUX field and as a new format in ZIP-files. Several authors have presented improvements to the original algorithm. Andersson and Nilsson published in 1994 and 1998 several papers about Radix Sort, which can be used as the first sorting step during the BWT [2, 3]. Fenwick described some BWT sort improvements including sorting long words instead of single bytes in 1995 [4]. Kurtz presented in 1998 and 1999 several papers about BWT sorting stages with suffix trees, which needed less space than other suffix tree implementations and are linear in time [5, 6].

Sadakane described a fast suffix array sorting scheme in 1997 and 2000 [7, 8]. In 1999, Larsson presented an extended sorting scheme for suffix arrays [9]. Based on already sorted suffices, Seward developed in 2000 two fast suffix sorting algorithms called "copy" and "cache" [10]. Itoh and Tanaka presented in 1999 a fast sorting algorithm called the two stage suffix sort [11]. Kao improved the two stage suffix sort in 2001 by a new technique which is very fast for sequences of repeated symbols [12]. Manzini and Ferragina published in 2002 some improved suffix array sorting techniques based on the results of Seward, Itoh and Tanaka [13]. Beside linear approaches based on suffix trees, Kärkkäinen and Sanders presented in 2003 an algorithm which sorts the array in linear time [14].

Several techniques for the post BWT stages have been published as well. Besides the Move-To-Front (MTF) improvements from Schindler in 1997 [15] and from Balkenhol and Shtarkov in 1999 [16], an MTF replacement, called Inversion Frequencies, was introduced by Arnavut and Magliveras in 1997 [17, 18, 19]. Switching between different post BWT stages was examined by Chapin in 2000 [20]. Deorowicz presented in 2000 another MTF replacement, named Weighted Frequency Count [21]. An efficient post BWT stage, Incremental Frequency Count, was presented by Abel in 2005 [22] and more deeply with an hierarchical coding model in 2007 [23].

Various modeling techniques for the entropy coding at the end of the compression process were presented by Fenwick [4, 24, 25], Balkenhol and Shtarkov [16], Deorowicz [21] and Maniscalco [26, 27, 28].

The purpose of this paper is to provide an overview of different BWT based compression algorithms from a practical point of view. The focus is set on different stages subsequent to the BWT, with no special preprocessing for different kind of data like text preprocessing [23, 29, 30, 31, 32, 33, 34, 35], or binary preprocessing before the BWT [36]. Several variants including a new context based algorithm will be presented and their basic properties, compression rates, compression times and decompression times compared.

2. Basic Concepts

2.1 Standard scheme

A typical scheme of the Burrows-Wheeler Compression Algorithm (BWCA) is presented in Figure 1 and consists of four stages. Each stage is a block transformation of the input buffer data and forwards the output buffer data to the next stage. The stages are processed sequentially from left to right for compression; for decompression they are processed from right to left with the respective backward transformations. For compression, the first stage is the BWT. The purpose of this stage is

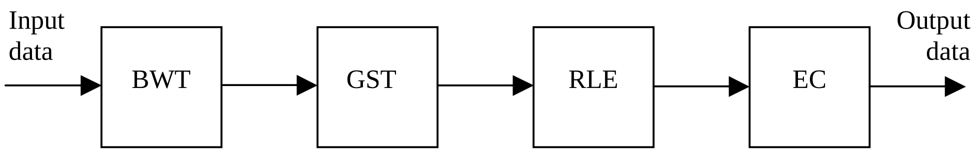


Figure 1: Typical scheme for the Burrows-Wheeler compression algorithm

to sort the data in a way that symbols with a similar context are grouped closely together. The BWT stage keeps the number of symbols during the transformation constant, except an additional index, which is created during the forward transformation and which has a value between 0 and buffer size - 1. The additional index is needed during the backward transformation in order to move the symbols back to their original order. The index as an integer can be encoded for example as a short byte sequence in the front of the sorted symbols. The second stage is called Global Structure Transformation (GST), which transforms the local context of the symbols to a global context [21, 37]. A typical representative of a GST stage is the Move-To-Front transformation (MTF), which was used by Burrows and Wheeler in their original publication [1] and which was the first algorithm used as a GST stage in a BWCA. The MTF stage is an algorithm for the list update problem, which replaces the input symbols by corresponding recency ranking values [38]. Just like the BWT stage – taking the additional index of the BWT not into account – a GST stage does not alter the number of symbols. The third stage typically shrinks the number of symbols by applying a Run Length Encoding scheme (RLE). Different algorithms have been presented for this purpose, with the Zero Run Transformation (RLE0) from Wheeler found to be an efficient one [24]. The last stage is an Entropy Coding stage (EC), which compresses the symbols by using an adapted model.

In order to elucidate the operation modes of the different stages, Figure 2(a) - 2(e) displays the transformed data of the input string "abracadabraabracadabra" in hexadecimal ASCII code. The input data of the BWT stage – except the additional index which is not shown for simplicity – is shown in Figure 2(a). As can be seen in Figure 2(b) the output data of the BWT stage contains many sequences of repeating symbols and has a local structure, i.e. symbols with a similar context form small fragments. The GST stage – in this example an MTF scheme – transforms the local structure of the BWT output to a global structure by using a ranking scheme according to the last recently used symbols and produces sequences of continuous zeros which are displayed in Figure 2(c). The RLE0 stage from Wheeler in Figure 2(d) removes the zero runs and the final EC stage produces a bit output in Figure 2(e) by using an arithmetic coding scheme.

```
(a) BWT input   : 61 62 72 61 63 61 64 61 62 72 61 61 62 72 61 63 61 64 61 62 72 61
(b) BWT output  : 61 72 72 64 64 61 72 72 63 63 61 61 61 61 61 61 61 61 62 62 62 62
(c) GST output  : 61 72 00 65 00 02 02 00 65 00 02 00 00 00 00 00 00 00 65 00 00 00
(d) RLE0 output : 62 73 00 66 00 03 03 00 66 00 03 00 00 00 66 00 00
(e) EC output   : 00 0D 01 8D B3 FF 81 00 72 A8 E8 2B
```

Figure 2: Transformed HEX data of "abracadabraabracadabra" by different stages

The BWT sorts the input array using the following context. By reversing the symbol order of the input array, it is possible to use the preceeding context instead of the following context similar to Prediction by Partial Matching (PPM) [39]. Balkenhol and Shtarkov reversed the symbol order for binary files before the BWT [16]. They defined binary files as files with an alphabet size of 256. Fenwick mentioned, that even for binary files, context reversing does not automatically lead to better compression ratios [40]. For some type of file, e.g. for image files with 16 bit pixels, which store values with the most significant byte first (big endian) as the radiographs from the Lukas 2D 16 bit medical image corpus [41], reversing the input order offers noticeable improvement as to be seen in the results section. It depends on the respective type of file, if context reversing achieves advantages; this implementation reverses binary files. Not all binary files have an alphabet size of 256, e.g. 8-bit image files with less than 256 colours. Therefore, all files which have an alphabet size higher than 230 are treated as binary files and are rearranged in reverse order before the BWT in the presented implementations if not otherwise stated.

2.2 Burrows-Wheeler Transformation

As the first stage of the compression algorithm, the Burrows-Wheeler Transformation (BWT) is the heart of the algorithm [1]. The purpose of this stage is the reordering of the input data depending on its context. The reordering produces many runs of equal symbols inside the output data of the BWT stage. In order to move the reordered symbols back into their original positions, a backward transformation exists, which reproduces exactly the input sequence. The backward transformation is a simple linear algorithm which is much faster than the forward transformation. Further information and variations of the BWT are explicated in 2008 by Adjeroh, Bell and Mukherjee [42].

The function of the forward and backward transformation will be explained on the example of the input string *ABRAKADABRA* with a length of 11 symbols. The counting of the index starts with 0. The preceeding context of a symbol *s* inside a string is defined as the context build by symbols left from *s*, which is the standard context used in compression algorithms. The following context of a

Table 1
Producing the n rotations of ABRAKADABRA.

Index	F									L
0	A	B	R	A	K	A	D	A	B	R	A
1	A	A	B	R	A	K	A	D	A	B	R
2	R	A	A	B	R	A	K	A	D	A	B
3	B	R	A	A	B	R	A	K	A	D	A
4	A	B	R	A	A	B	R	A	K	A	D
5	D	A	B	R	A	A	B	R	A	K	A
6	A	D	A	B	R	A	A	B	R	A	K
7	K	A	D	A	B	R	A	A	B	R	A
8	A	K	A	D	A	B	R	A	A	B	R
9	R	A	K	A	D	A	B	R	A	A	B
10	B	R	A	K	A	D	A	B	R	A	A

symbol *s* is defined as the context produced by symbols right from *s*, which is the context used in the BWT because of the alphabetical sorting.

For the forward transformation, the input string with the length *n* is written *n*-1 times under the original string, producing a list with *n* lines. Each line – starting with the second line – is then rotated *i* symbols to the right, with *i* being the index of that line. Symbols which leave the last column are moved back into the first column. Table 1 shows the result for the input string *ABRAKADABRA*. Next, all lines are sorted in alphabetical order. During this process, two columns have a special meaning: the first and the last column. The first column – called column *F* (first) – contains all symbols of the input string sorted in alphabetical order, which places symbols with a similar following context directly behind each other. The last column– called column *L* (last) – has some remarkable properties even without being completely sorted. Because of the cyclic sorting, *L* is the predecessor column of *F*. Therefore, many symbols with a similar following context are placed nearby which produces many runs of equal symbols in column *L*. Compared to the original string, *L* is much easier to compress for most entropy coders. The result for the input *ABRAKADABRA* can be seen in Table 2. The output of the BWT are column *L* and index *i* of the original input string inside the list after the sorting. In this example, the output consists of the string *RDAKRAAAABB* and the index 2.

The entire forward transformation can be described briefly as:

1. Create all possible rotations of the input string resulting in a list of *n* rotated strings.
2. Sort the rotated strings in alphabetical order.
3. Output the last column *L* of the list and the index *i* of the input string inside the list.

Table 2
Sorting of the n rotations of ABRAKADABRA.

Index	**F**									**L**
0	A	A	B	R	A	K	A	D	A	B	R
1	A	B	R	A	A	B	R	A	K	A	D
2	A	B	R	A	K	A	D	A	B	R	A
3	A	D	A	B	R	A	A	B	R	A	K
4	A	K	A	D	A	B	R	A	A	B	R
5	B	R	A	A	B	R	A	K	A	D	A
6	B	R	A	K	A	D	A	B	R	A	A
7	D	A	B	R	A	A	B	R	A	K	A
8	K	A	D	A	B	R	A	A	B	R	A
9	R	A	A	B	R	A	K	A	D	A	B
10	R	A	K	A	D	A	B	R	A	A	B

Common implementations of the BWT do not calculate *n* rotated strings of the input string, which would be impractical for larger input sizes, but use different data structures like a suffix array. Instead of *n* rotated strings the suffix array uses only the original string and the *n* positions of the start of the rotations [7, 8, 11, 12], leading to a much lower memory consumption.

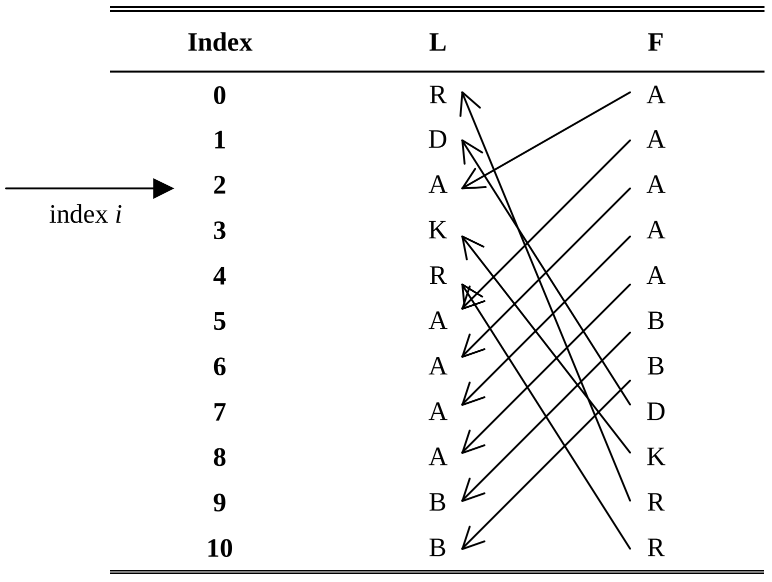
Since no backward algorithm exists for a standard alphabetical sort of symbols, the question arises how the backward transformation of the BWT is able to reconstruct the original order of the symbols. One of the most remarkable properties of the BWT is the fact that not only the original order of the symbols can be reconstructed by column *L* and index *i*, but that the backward transformation is even much faster and simpler than the forward transformation.

Since the forward transformation is a special sort of the input symbols, column *L*, which is the output of the forward transformation and the input of the backward transformation, contains exactly the same symbols than column *F*; they differ only in their respective positions. Therefore, column *F* can be reproduced by a simple alphabetical sort of the symbols of column *L*. Columns *L* and *F* are then placed behind each other and build a simple table as shown in Table 3. The index *i* from the output of the forward transformation is the starting point of the backward transformation. All symbols of column *F* are output using a special order of the lines. The first symbol to output is the symbol at column *F* in line *i*, in this example the symbol *A*, which is the third *A* from the top in column *F*. The next line is the line which contains the third *A* from the top in column *L*. Here, line 6 contains the third *A* and the symbol of column *F* in line 6, a *B*, is output as the second symbol. This symbol is the second *B* in column *L*, leading to line 10, which contains the second *B* in column *F*. The third output is the symbol of column *F* in line 10, an *R*. The process continues until it reaches the starting line *i*, where it stops.

Table 3
Reproducing the F column from RDAKRAAAABB.

	Index	**L**	**F**
	0	R	A
	1	D	A
index *i* →	**2**	A	A
	3	K	A
	4	R	A
	5	A	B
	6	A	B
	7	A	D
	8	A	K
	9	B	R
	10	B	R

Table 4
Calculation of the output symbols from RDAKRAAAABB.



	Index	**L**	**F**
	0	R	A
	1	D	A
index *i* →	**2**	A	A
	3	K	A
	4	R	A
	5	A	B
	6	A	B
	7	A	D
	8	A	K
	9	B	R
	10	B	R

The entire backward algorithm can be described by 5 steps:

1. Create column *F* by sorting the symbols of column *L*.
2. Start with line *i*.
3. Output the symbol at column *F* of the current line.
4. Move to the line which has the same position at the *L* column within the set of the last symbol than the position of the last symbol inside the set of the last symbol at the *F* column, e.g. if the last symbol was an *C*, and the *C* was the fifth *C* in column *F*, move to the line with the fifth *C* in column *L*.

5. Repeat 3 and 4 until line *i* is reached.

2.3 Global Structure Transformation

The Global Structure Transformation (GST) is the second stage within the basic Burrows-Wheeler Compression Algorithm (BWCA). The output of the BWT contains many symbol fragments, which have the same right context and a local probability distribution. It is typical for the output of the BWT that the symbol fragments change suddenly, switching to a new set of symbols with a different probability distribution. This local structure is transformed into a global structure by the GST stage, which is better to compress for the EC stage. A global structure, which is a structure with a stable probability distribution for the whole file, can be achieved for example by a recency ranking scheme like the Move-To-Front stage (MTF) [1]. The MTF stage produces an index sequence with many 0s. Another possibility is a distance measurement scheme like Inversion Frequencies (IF) [17, 18, 19]. Theses schemes will be explained in detail in the sections about the post BWT stages.

2.4 Run Length Encoding

Run Length Encoding (RLE) is a simple and popular data compression scheme. The sequence of length *l* of a repeated symbol *s* is replaced by a shorter sequence, usually containing one or more symbols of *s*, a length information and sometimes an escape symbol *c*. RLE stages differ from each other mainly in three points: the threshold *t*, the marking of the start of a run, and the coding of the length information *l*. The threshold *t* defines the minimum run length *l* for a run to be encoded. If *l* is smaller than *t*, the run keeps unchanged, and if *l* is greater or equal to *t*, the run is replaced. The start of a run can be indicated by a threshold run or an escape symbol *c*. If a threshold run is used, the start is characterized by a small sequence of s, which has a length of *t*. If an escape symbol *c* indicates the start of a run, *s* is normally put behind *c* in order to characterize the run symbol. The escape symbol *c* must not be an element of the alphabet or occurrences of *c* have to be encoded unambiguously. The length information *l* can be coded in different ways. Usually, *l* is put directly behind the threshold run or behind s. Note, that *c* and *l* disturb the context of *s* inside the output sequence.

In the past, different RLE schemes for BWCAs have been presented by Wheeler, Fenwick and Maniscalo [24, 43, 44]. The main function of the RLE is to support the probability estimation of the next stage. Long runs of a symbol *s* tend to overestimate the global symbol probability of *s* for fragments, where *s* occurs only occasionally. The result is that within these disjoined fragments, the probability value for *s* is too high which leads to lower compression. Balkenhol and Shtarkov name this phenomenon “the pressure of runs” [16]. The RLE stage helps to decrease this pressure. In order to improve the probability estimation of the EC stage, most BWCA schemes place the RLE stage directly in front of the EC stage.

Maniscalco describes in 2000 and 2001 algorithms which use a variable length code and divide the length information into two parts: an exponent part and a binary representation part [43, 44]. The exponent part, called the size of the variable length code in Maniscalco's paper, reflects the logarithm of *l*. The binary representation part, called the value of the variable length code by Maniscalco, contains the bits of the binary representation of *l* and can be transmitted independently from the exponent part.

Another example of an RLE stage for BWT based compressors is the Zero Run Transformation (RLE0) from Wheeler [24, 25]. Wheeler suggested coding only the runs of the 0 symbols and no runs of other symbols, since 0 is the symbol with the most runs. In this case, an offset of 1 is added to all symbols except 0 [24]. All occurrences of 0 in the input array of the RLE0 stage are encoded. The run length of each zero run is incremented by one and all bits of its binary representation except the most significant bit – which is always 1 – are stored by the symbols 0 and 1.

Some authors suggested an RLE stage before the BWT stage for speed optimization, but such a stage deteriorates the compression rate in general [21]. Since there are sorting algorithms which sort runs of symbols practically in linear time [11, 12, 10, 13] or even sort the whole array in linear time like the algorithm of Kärkkäinen and Sanders [14], there is no reason to use such a stage before the BWT stage for speed reasons. Besides speed, Chapin mentioned that an RLE stage before the BWT can help to achieve better compression rates in some cases by processing a larger amount of bytes at one go with the fixed block size of the BWT [45].

In 2005, Abel presented a scheme, which uses an RLE stage directly behind the BWT stage – going back to an idea of Gringeler [46] – and mentioned two reasons for the new position [22]. First, encoding of the runs lowers the pressure of runs inside the GST stage, which leads to a more compressible GST output stream. Second, the compression process achieves a higher speed, because the length of the RLE output is usually smaller than the RLE input length and an RLE stage is faster than a GST stage. The scheme is shown in Figure 3 and used for all following BWCA

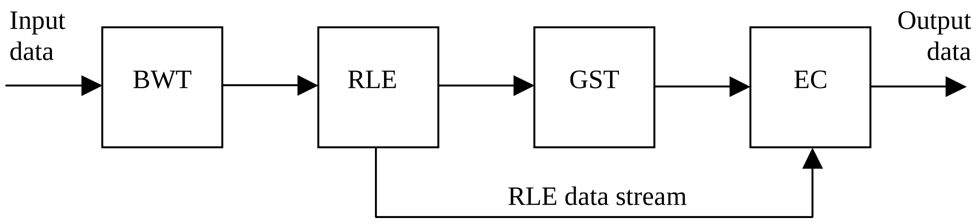


Figure 3: Improved BWCA with RLE stage in front of GST

versions if not otherwise stated. The output of the RLE stage consists of two streams, the main stream – including the input symbols without the runs – going to the GST stage as well as the RLE data stream going directly to the EC stage. The RLE data stream contains the run length information, which bypasses the GST stage in order not to disturb the symbol context of the following GST stage. At the EC stage, the RLE data stream is encoded separately from the GST output into the same output file [47, 23].

2.5 Entropy Coding

The type of encoding of the GST output inside the Entropy Coding (EC) stage has a strong influence on the compression rate. Different types of entropy coders can be used to compress the GST output and RLE data stream. The most important ones are Huffman and arithmetic coders. Huffman coders offer better compression speed and arithmetic coders offer better compression rates. The implementation presented here is based on arithmetic coding. Arithmetic coding assigns to each symbol of the alphabet a probability based on the former occurrences of that symbol. At the start of the encoding, the interval [0, 1) is divided into sub-intervals which equal the probabilities of the alphabet symbols. The larger the probability, the larger the sub-interval. For each symbol to encode, the sub-interval of the respective symbol is taken and again divided into sub-intervals proportional to the probabilities of the alphabet symbols. The output is a rational number, which is inside the sub-interval of the last symbol encoded. A good introduction can be found in the book of Nelson and Gailly [48].

Table 5
Examples of index codings.

Index	1. Level	2. Level	3. Level
0	‘0’	-	-
1	‘1’ ’0’ ’0’	-	-
2	‘1’ ’0’ ’1’	-	-
3	‘1’ ’1’	0	0
4	‘1’ ’1’	0	1
5	‘1’ ’1’	1	0
6	‘1’ ’1’	1	1
7	‘1’ ’1’	1	2
8	‘1’ ’1’	1	3
9	‘1’ ’1’	2	0
100	‘1’ ’1’	5	35
128	‘1’ ’1’	5	63
129	‘1’ ’1’	6	0
255	‘1’ ’1’	6	126

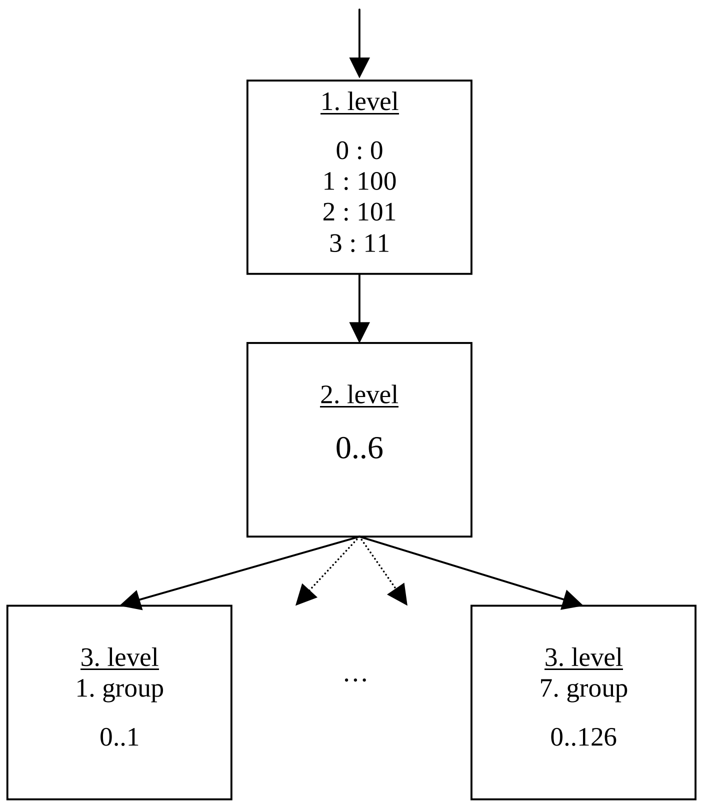


Figure 4: Coding model for recency ranking schemes

It is not sufficient to compress the GST output just by a simple arithmetic coder with a common unstructured order-n context. Even after the use of an RLE stage before or after the GST stage, the symbols 0 and 1 are still by far the most common symbols on average. As discussed by Fenwick [24, 25] and later by Abel [23], a hierarchical coding model offers good compression results for such skew distributions.

Similar to the model of Fenwick, this paper uses a hierarchical coding model for GST stages based on a ranking scheme, consisting of three hierarchical levels as shown in Figure 4 and [23]. The first and second level as well as each group in the third level use independent arithmetic coders with their own contexts of symbol frequencies. The first level handles the symbols 0, 1 and 2, which are the most frequent symbols. All symbols greater than 2 are handled by the second and third level of the model, with an escape symbol at the first level. Level two and three build a structured coding model. Level two acts as a selector and divides the symbols in seven disjoint subsets: {3, 4}, {5, ..., 8}, {9, ..., 16}, {17, ..., 32}, {33, ..., 64}, {65, ..., 128} and {129, ..., 255}. Level three handles the offset of the current symbol in each subset [24, 23]. For example, the symbol 8 would be encoded in the first level as $11_{binary,}$ in the second level as 1 and in the third level as 3 as shown in Table 5.

For GST stages based on a distance measurement, as well as for the RLE data stream, a different

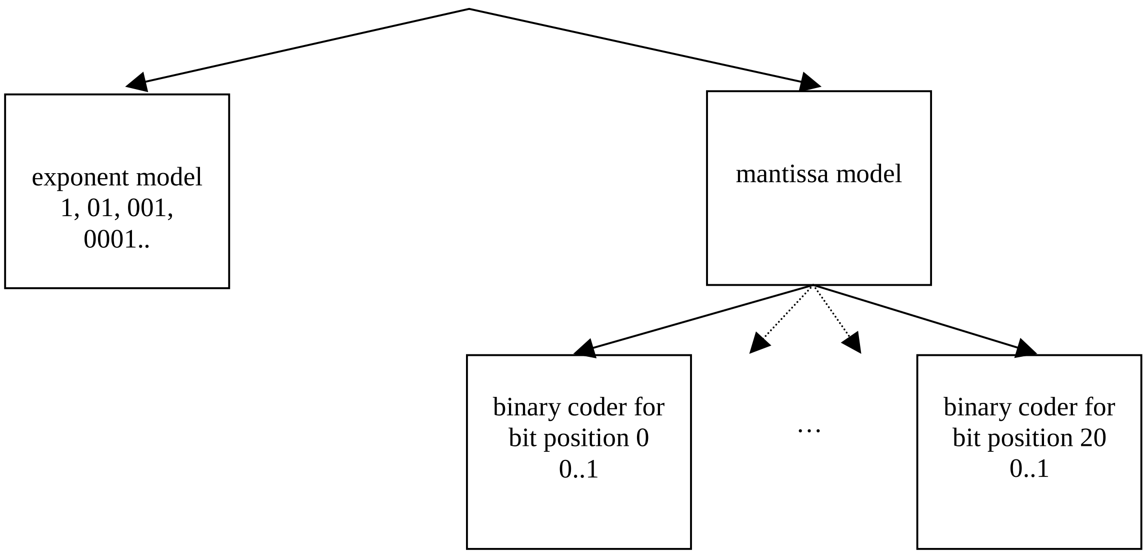


Figure 5: Coding model for distance schemes

coding model is used because the length of the alphabet is greater than 255 and because the context is different from the context of indices. The coding model consists of two main parts as pictured in Figure 5 and is similar to the Elias gamma code [49]. All parts are encoded by independent arithmetic coders. The first part is the exponent part and compresses the binary logarithm *l* of each distance, which represents the number of bits of the binary representation of the distance. The exponent *l* is saved in unary coding as a sequence of *l* 0s and a trailing 1 as a terminator symbol, e.g. an exponent of 6 is saved as 0000001. The second part is the mantissa part and compresses the binary representation of each distance. For each bit position starting with the second most significant position – the first bit position is always 1 – a binary coder with its own context is used. Since the exponents are stored alongside the binary representations, it is possible to decide how many bits are used for each distance and to decode the distance later unambiguously without a terminator symbol between the mantissa data.

3. Post BWT Stages

3.1 Definitions

For the description of the algorithms, the following notation will be used. Let A be an ordered set, called alphabet, with size $|A|$. Let $X = x_0 x_1 x_2 \dots x_{n-1}$ denote a sequence with length n and $x_i \in A$ with $0 \leq i \leq n-1$. The first index of a sequence is 0. Each stage has an input sequence X_{in} and an output sequence X_{out} as well as a corresponding input alphabet A_{in} and an output alphabet A_{out}. A stage processes the symbols of X_{in} and calculates the corresponding symbols of X_{out}. After finishing one

stage, X_{out} of this stage will be used as X_{in} of the following stage. The maximal size for X_{in} is called the blocksize b_n. For most stages, A_{in} and A_{out} will have a bit width of 8 bits resulting in $|A_{in}| = |A_{out}| = 256$. Stages using a distance measurement, and all stages in the algorithm following that stage, will have a bit width of $\log_2(b_n)$. In this case a bit width of 32 will be assumed in order to handle all values as blocksizes are smaller than 4 GB. Furthermore, the binary representation sequence of a symbol a is denoted as $BR(a)$, for example $BR(4)$ = '100' and $BR(7)$ = '111'.
In the following sections, several representatives of GST stages are discussed and compared. The recency ranking and distance measurement schemes can be used as the GST stage in Figure 3, the context based schemes includes the GST and EC stages.

3.2 Move-To-Front

3.2.1 Basic properties

The basic GST stage is the Move-To-Front stage (MTF), a recency ranking scheme used in many BWCA implementations. The MTF stage transforms the input symbol sequence into an index sequence. For each input symbol of X_{in}, an output index is written to X_{out}. The smaller the index, the closer is the last occurrence of the corresponding symbol. In order to calculate the index values, a list of the alphabet symbols is used, which is ordered by the last occurrences of the alphabet symbols. At the beginning, the list is sorted in ascending alphabet order. Each time a symbol of X_{in}, is processed, the corresponding alphabet symbol is located inside the list, the current index of that symbol is written to X_{out} and the symbol is moved to position 0 in the list. Such a ranking scheme transforms a run of repeated symbols with a run length of m into a sequence of 0s with length m - 1. The sequence of 0s has a length of m - 1, as the first 0 occurs one position later in the output sequence. Since the BWCA sorts the input symbols according to their context, the output of the BWCA contains many symbol runs. All these runs are transformed into runs of 0s by the MTF stage independent of the former symbol value, i.e. the local contexts are transformed into a global context. For an entropy coding stage, it is more efficient to compress many runs of the same symbol than to compress many runs of different symbols.
One problem of the MTF stage is that it moves each symbol directly to the front of the list, no matter how seldom it has occurred before. If a symbol appears only rarely, it removes other symbols, which might be more frequent, from the front of the list to higher ranks, which are more expensive to encode. An advantage of the MTF stage is its low complexity, which gives the stage a high throughput.

3.2.2 Improvements

Several enhancements try to improve the properties of the MTF stage like the M1 and M2 version of Schindler [15], the MTF-1 algorithm from Balkenhol, Kurtz and Shtarkov [50] and the MTF-2 algorithm from Balkenhol and Shtarkov [16]. The M1 and M2 versions use flags to achieve an output sequence, which increase the number of rank 0 symbols at the cost of rank 1 symbols, i.e. they produce more 0s [15]. The MTF-1 moves only symbols from the second position to the front of the list, whereas symbols with higher positions are moved to the second position. MTF-2 differs from MTF-1 in that symbols from the second position are moved to the front of the list only if the last ranking value was not zero, i.e. if the same symbol occurred again. Fenwick used a sticky MTF stage [51] in order to improve the compression rate. All these MTF variations try to make the MTF stage slower to adept when processing new appearing symbols.

Another improvement to the standard BWCA with an MTF stage is to place the RLE scheme directly in front of the MTF stage as shown in Figure 3 instead of in front of the EC stage. For the MTF based BWCA used in the results section, a standard MTF scheme is used with an RLE stage in front of it.

3.3 Inversion Frequencies

3.3.1 Basic properties

Several MTF replacements have been unveiled since the birth of the BWCA in 1994. Some of these improvements are based on a distance measurement like the Distance Coding (DC) algorithm from Binder [52, 53] and the algorithm from Arnavut and Magliveras [17]. which they named Inversion Frequencies (IF). In 2000, Arnavut compared the MTF stage with the IF stage [18]. The IF stage offered better compression than the MTF stage for almost all files. In 2004, several IF schemes have been compared against MTF like schemes by Arnavut [19]. Particularly for large image and DNA files, the IF schemes achieved better compression rates than the recency ranking schemes.

An theoretical analysis of the IF algorithm was presented by Ferragina, Giancarlo and Manzini in 2006 at the ICALP Conference [54]. The IF algorithm is not a recency ranking scheme like MTF but is based on distances between the occurrences of the same symbols. It produces for each symbol $a \in A_{in}$ a sequence S_a. For each alphabet symbol a, the input sequence X_{in} is scanned and if the current element of X_{in} is equal to a, the number of symbols greater than a between the current position and the last position of a is output. In order to reproduce X_{in} from the set of S_a, either the frequencies of the alphabet symbols or a terminator symbol behind each S_a is needed in addition. One advantage of the IF algorithm is the fact that the partial sequence of the last symbol z of the alphabet, called S_z, consists only of the symbol 0. Therefore, S_z is not needed in order to reproduce the original sequence and the length of X_{out} gets smaller than the length of X_{in}. X_{out} of IF is different from X_{out} of MTF in many aspects. X_{out} of the MTF stage contains many zero runs, which represent

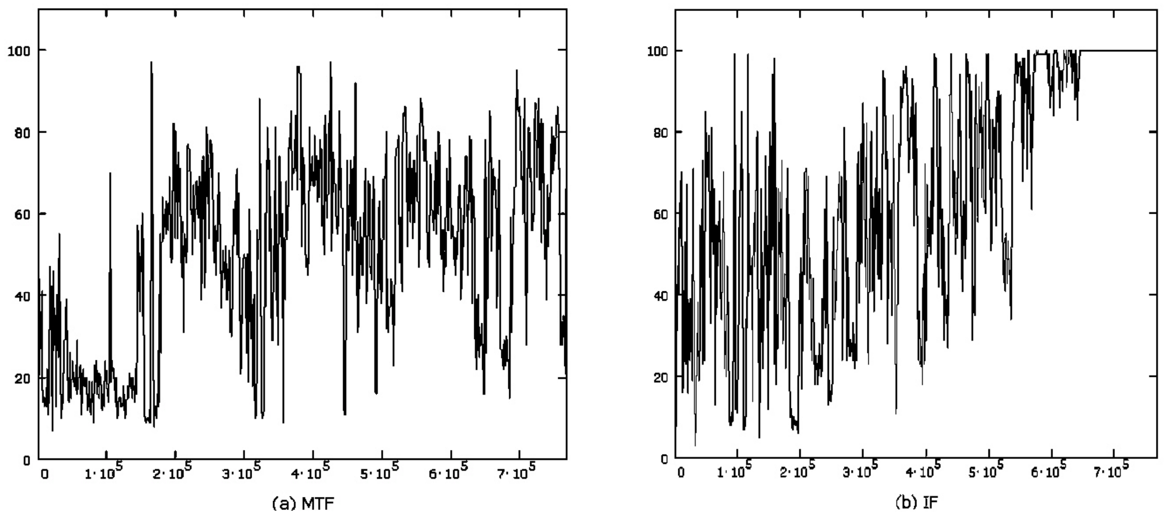


Figure 6: Fraction of zeros of book1 for (a) MTF output and (b) IF output

runs of equal symbols, and these runs are more or less equally distributed over the whole sequence as pictured in Figure 6. X_{out} of the IF stage consists of several sequences S_a, one for each $a \in A_{in}$ except for the last symbol z. S_a of higher symbols have typically smaller values than S_a of lower symbols, since the number of symbols, which are greater than the scanned symbol, is decreasing. S_a for the last symbols of A_{in} have usually many long runs of zeros. In order to represent this behavior, Figure 6 compares the fraction of the zeros of the file `book1` over the file position for both the MTF stage output and for the IF stage output. For a better comparison, S_z is included in X_{out} of IF. As can be seen, the average fraction of zeros in the output of IF is rising towards the end of the file until it reaches 100% at the end. In the output of MTF, the average fraction of zeros fluctuates around 60%.

3.3.2 Improvements

For each sequence S_a, only symbols which are greater than a are counted. Hence, if symbols with a high probability are processed first, the partial sequences of the following symbols, with a lower frequency distribution, have smaller values. On the other hand, the partial sequences for symbols with a high frequency distribution are longer than the partial sequences for symbols with a lower frequency distribution. In order to point out the influence of the frequency distribution, A_{in} of the IF stage is sorted in ascending frequency order as well as in decreasing frequency order. Table 6 denotes the compression rates for the original alphabet and for both sorted alphabets. In most cases, the ascending frequency alphabet permutation produces better compression rates than the original alphabet.

Table 6: Compression rates in bps for IF stage with an original alphabet, a permuted alphabet sorted by ascending frequencies and a permuted alphabet sorted by descending frequencies. Best compression rates are printed in bold font.

File	Original Alphabet	Asc. Frequ. Alphabet	Desc. Frequ. Alphabet
bib	1.926	**1.919**	1.936
book1	**2.230**	2.231	2.242
book2	1.941	**1.933**	1.948
geo	4.196	4.243	**4.161**
news	2.420	**2.405**	2.433
obj1	3.880	3.892	**3.793**
obj2	2.495	**2.481**	2.487
paper1	2.436	**2.417**	2.449
paper2	2.350	**2.340**	2.358
pic	0.709	0.709	**0.706**
progc	2.482	**2.473**	2.498
progl	1.716	**1.703**	1.723
progp	1.730	**1.719**	1.737
trans	1.519	**1.506**	1.544
Avg.	**2.288**	**2.284**	**2.287**

While most files achieve a better compression rate with an ascending frequency alphabet, some files have better results with a descending frequency alphabet. Only book1 achieves the best result with the original alphabet, but the result is very close to one of the ascending frequency alphabet. In order to find the optimal sorting direction for each file, some characteristics of the frequency distribution of A_{in} can be used. For each symbol a of A_{in}, let f_a be the number of occurrences of a within X_{in}, i.e. the symbol distribution. Let F_{avg} denote the average frequency count of X_{in} with length n by

$$F_{avg} = \frac{n}{|A_{in}|} \ . \qquad (1)$$

Further, G is defined as the set of symbols for which f_a is greater to 2 F_{avg} by

$$G = \{a \mid f_a > 2F_{avg}\} \ . \qquad (2)$$

Then S describes the percentage share of symbols a of the alphabet A_{in}, for which f_a is greater to 2 F_{avg} by

$$S = 100 \frac{|G|}{|A_{in}|} \ . \qquad (3)$$

Table 7: Percentage share S of symbols for which $f_a > 2\,\mathrm{F}_{\mathrm{avg}}$.

File	*S*
`bib`	13.41
`book1`	15.85
`book2`	15.46
`geo`	5.47
`news`	15.15
`obj1`	10.55
`obj2`	9.38
`paper1`	14.58
`paper2`	16.30
`pic`	11.32
`progc`	13.98
`progl`	18.18
`progp`	14.44
`trans`	17.17
Avg.	**13.66**

Table 7 reveals *S* for each file of the Calgary Corpus. The values of *S* for the files `geo`, `obj1`, `obj2` and `pic` are the lowest ones in the table. All of them except `obj2` achieve the best compression rates with a descending frequency sort order. As a heuristic approach in the presented implementation, the alphabet inside the IF stage is permuted depending on the symbol distribution. The IF stage calculates the frequency distribution of the symbols together with the corresponding value of *S* at first, and performs afterwards a permutation of A_{in}. If *S* is greater than or equal to 10, the alphabet is sorted in ascending frequency order, otherwise in decreasing frequency order.

Another improvement for IF based BWCAs is a special RLE stage in front of the IF stage. The RLE stage calculates for each run of length *n* two parts: the exponent part and the mantissa part, similar to the EC model of Figure 5. The exponent part consists of a run of the same symbol with length $\log_2(n\text{-}1)$, i.e. the exponential part of the original run length. This run is usually much shorter – especially for longer runs – and is written into X_{out}. The binary representation $BR(n\text{-}1)$ as the mantissa part is written out as a bit sequence without any terminator symbol and is the RLE data stream shown in Figure 3. The RLE data stream bypasses the GST stage and is encoded independently from X_{out} in the EC stage.

3.4 Weighted Frequency Count

3.4.1 Basic properties

The Weighted Frequency Count algorithm (WFC) was presented by Deorowicz in 2002 [53]. As a representative of a ranking scheme, it is closer to MTF than to IF. It replaces the input symbol *x* with a corresponding ranking value *y*(*x*). The difference between WFC and MTF is the function, which calculates *y*(*x*). Inside the MTF algorithm, *y*(*x*) is the index of the current input symbol *x* within a list *L* of alphabet symbols. Upon each request of *x*, the current index *y*(*x*) is output and *x* is moved to the front of *L*. Since a symbol is moved straight to the front of *L* without taking the former frequency distribution of this symbol into account, the MTF stage might push more frequent symbols aside by less frequently used symbols leading to sub optimal compression rates. The WFC stage calculates *y*(*x*) by a function, which takes into account the symbol frequencies and the distance of the last occurrences of *x* inside a sliding window of size t_{max} [53]. Hereto, each position inside the sliding window is assigned a weight. The weights of closer distances are higher than the weights of distances more far away. For each alphabet symbol, the weights of its occurrences inside the window are summed up into a corresponding counter. The counter list of the alphabet symbols is sorted in descending order, i.e. the largest counter is at index position 0. The weighting and the sorting have to be recalculated for each symbol processed. This way, more frequently used symbols get a lower index value than less frequently used symbols, which supports the following EC probability estimation. Table 8 presents the average ranking values r_x of the MTF and WFC stage for the files of the Calgary Corpus. The MTF and WFC stage are both performed with an RLE stage processed beforehand. In all cases, the average ranking values of the WFC stage are smaller than the corresponding values of the MTF stage leading to better compression rates. The main drawback of the WFC stage is the high time consumption as described at the results section.

3.4.2 Improvements

The WFC implementation of Deorowicz gets the best compression rate with 2.249 bps for the Calgary Corpus by the weight function w_{6q}, which uses 5 logarithmic quantized levels [53]. All symbols within a level get the same weight. For the implementation of his algorithm, a wide set of different weight functions based on logarithmic levels was examined by Deorowicz. Since the compression rate depends on several parameters beside the weight function and the number of logarithmic levels, like the kind of RLE algorithm and the model of the EC stage, it is not easy to predict which weight function and number of levels will lead generally to the best compression results. The WFC approach described in this paper is used in the ABC compression program [47, 55] and uses a finer graduation by using more levels, which leads to improved results but needs more time to calculate. In the present implementation, the best compression rate is achieved at 12

Table 8: Average ranking values r_x for the MTF and WFC stage.

File	**MTF Average r_x**	**WFC Average r_x**
bib	5.66	5.42
book1	3.86	3.51
book2	4.39	4.07
geo	49.61	44.76
news	6.82	6.24
obj1	46.32	43.66
obj2	18.38	17.59
paper1	5.86	5.57
paper2	4.85	4.54
pic	8.21	7.08
progc	7.21	6.88
progl	5.02	4.78
progp	5.64	5.45
trans	6.15	5.91
Avg.	**12.71**	**11.82**

logarithmic levels instead of 5. For the size of the sliding window t_{max}, the same value as in [53] is used:

$$t_{max} = 2048 \ . \quad (4)$$

The individual weights of the weight function are of central significance for the compression rate. Since the structure and symbol distribution varies from file to file, a weight function with fixed weights independent from the file structure will not lead to optimal compression rates for all files. For some files, a function with stronger weights for symbols of the immediate past rather than for older symbols is best suited. For other files, a weight function, which weights older symbols almost the same as more recent symbols, offers better results. Therefore, the present implementation does not use fixed weights, but calculates the weights depending on the symbol distribution.

Hereto the same parameter S is used as before in the IF section. S describes the percentage share of symbols a of the alphabet A_{in}, for which the frequency count f_a of a inside X_{in} is greater to 2 F_{avg}, defined in equations (1) to (3). Further, $f(l)$ is defined as an integer function with parameters p_0, p_1 and S:

$$f_{p_0,p_1,S}(l)=\begin{cases}2^{17} & l=0\\ 2^{14} & l=1\\ \dfrac{f_{p_0,p_1,S}(l-1)\cdot p_0}{p_1+(l\cdot S^2)} & l\ge 2\end{cases}. \tag{5}$$

The distance from the current position inside the sliding window is described by *t*, starting with 0 as the next symbol to the left from the current position. Then, the weight function $w_{p_0,p_1,S}(t)$ for the present implementation is defined as:

$$w_{p_0,p_1,S}(t)=\begin{cases}f_{p_0,p_1,S}(0) & t=0\\ f_{p_0,p_1,S}(1) & 2^0\le t\le 2^1-1\\ f_{p_0,p_1,S}(2) & 2^1\le t\le 2^2-1\\ f_{p_0,p_1,S}(3) & 2^2\le t\le 2^3-1\\ \ldots & \\ f_{p_0,p_1,S}(11) & 2^{10}\le t\le 2^{11}-1\\ 0 & t\ge 2048\end{cases}. \tag{6}$$

Table 9 displays the compression rates for different values of p_0 and p_1. As to be seen, there is no

Table 9: Compression rates in bps for different $w_{p_0,p_1,S}(t)$.
Best compression rates are printed in bold font.

File	p_0=2400 p_1=4000	p_0=2600 p_1=4200	p_0=2800 p_1=4400	p_0=2400 p_1=4400	p_0=2800 p_1=4000	p_0=2500 p_1=4300
bib	1.879	1.882	1.885	**1.876**	1.897	1.877
book1	2.251	**2.249**	2.250	2.256	2.255	2.252
book2	1.932	**1.931**	1.933	1.935	1.940	1.932
geo	4.114	4.105	4.099	4.134	**4.086**	4.121
news	2.384	2.380	**2.378**	2.391	**2.378**	2.386
obj1	**3.675**	3.676	**3.675**	3.681	3.681	3.676
obj2	2.389	2.394	2.401	**2.386**	2.422	2.388
paper1	2.374	2.377	2.381	**2.373**	2.395	**2.373**
paper2	**2.326**	**2.326**	**2.326**	2.331	2.332	2.327
pic	0.708	0.706	0.705	0.711	**0.703**	0.708
progc	**2.411**	2.413	2.416	2.413	2.427	2.412
progl	1.657	1.655	**1.654**	1.659	1.655	1.657
progp	1.653	1.654	1.656	**1.651**	1.666	1.652
trans	1.431	**1.430**	**1.430**	1.432	**1.430**	1.432
Avg.	**2.227**	**2.227**	**2.228**	**2.231**	**2.233**	**2.228**

best combination of p_0 and p_1 for all files, because the results differ from file to file. Therefore, the parameters p_0 and p_1 were chosen empirically [55], whereas the value of S is determined by the symbol distribution of the respective file. The best overall compression rate is achieved by the following values:

$$p_0 = 2600 \ , \tag{7}$$

$$p_1 = 4200 \ . \tag{8}$$

Even though this choice leads to the best overall compression rate, many files obtain better results with different settings, e.g. the file geo. The EC model used for the WFC based BWCA is the EC model of Figure 4 for ranking schemes described in the Entropy Coding section.

3.5 Incremental Frequency Count

3.5.1 Basic properties

The WFC stage offers strong compression rates, but it has a high cost of computation, because the weighting of the symbols within the sliding window and the sorting of the list has to be recalculated for each symbol processed. At the Data Compression conference 2005, a new scheme was presented by Abel which tries to achieve compression rates as good as the WFC stage but with a much lower complexity [22, 23]. The main idea is to use counters for symbols within a sliding window of the past like the WFC stage, but to update only one counter for each symbol processed. This way, only one counter needs to be resorted inside the list, which makes the stage much faster than the WFC stage. In order to weight common and close symbols stronger than rare and distant symbols, the counters can be increased or decreased; on average they are increased. Therefore, the stage is named "Incremental Frequency Count" (IFC) [22]. The counters are rescaled frequently in order to prevent overruns.

In front of the IFC stage, an RLE scheme is used which replaces all symbol runs with length l by a run of the same symbol with a length of 2 [23]. The original run length l is sent to the RLE data stream and bypasses the IFC stage as shown in Figure 3. The coding model is the same as the model for distance measurement and shown in Figure 5 [23].

The IFC stage consists of a loop, which processes all input symbols. The loop is divided into 5 parts. For each alphabet symbol, a counter is used. All counters are sorted in a list in descending order, i.e. the counter with the highest value is placed at index position 0. At the beginning, all counters are reset to 0 and ordered in ascending alphabet order. The first part of the algorithm reads the next symbol of the input stream and outputs the current index of the corresponding counter. The second part calculates the difference between two index averages. An index average is the average value of the last indices inside a sliding window of size *window_size*. The average avg_i at position i inside the input stream with the current index $index_i$ is calculated by:

$$avg_i := \frac{(avg_{i-1} \cdot (window\_size - 1)) + index_i}{window\_size} \quad . \tag{9}$$

In the approach presented here, a value of 8 is used for *window_size*. Larger values make the stage adapt slower, whereas smaller values make the stage adapt faster to context changes. The third part of the IFC calculates the increment of the counter values. In order to obtain an output sequence with a low index average, the increment has to be chosen very carefully. It is not sufficient to increment the respective counter by a constant, linear or exponential value. Instead, the increment is chosen depending on statistical properties of the recent indices. First, the difference between the last value avg_{i-1} and the current value avg_i is calculated:

$$dif_i := avg_i - avg_{i-1} \quad . \tag{10}$$

In order to ensure that small differences are treated accordingly but bigger differences are not overweighted, dif_i is limited by a fixed maximum dm:

$$difl_i := \min(|dif_i|, dm) \cdot \operatorname{sign}(dif_i) \quad . \tag{11}$$

The limitation has a similar effect as in the sticky MTF stage of Fenwick [51], as it lowers the influence of large index differences, which occur during context changes. Here, a value of 16 is used for dm. Finally, the increment inc_i is calculated in a way that it decreases when a context starts to change and that it increases when a context becomes stable. This way, frequent symbols in a stable context are weighted more strongly than new symbols of a changing context.

$$inc_i := inc_{i-1} - (\frac{inc_{i-1} \cdot difl_i}{64}) \quad . \tag{12}$$

Since an RLE stage is in front of the IFC stage, the run length of every run has been cut to 2 inside the IFC stage. In order to emphasize the weight of runs, inc_i is increased by 50% if the current symbol equals the last symbol. Finally, the counter of the current symbol is increased by inc_i. The fourth part of the IFC stage is concerned with rescaling. If the counter of the current symbol exceeds 256, all counters and inc_i are halved. At the fifth and last part of the IFC stage, the counter list is sorted in descending order. Only one counter has changed since the last symbol was processed and needs to be moved inside the list to the proper position. If the counter has a value equal to other counters in the list, it is moved to the smallest index of these counters. Sorting one counter is much faster than the sorting of the whole list at the WFC stage [23].

3.5.2 Improvements

Because of the RLE stage of Figure 3 in front of the IFC, a zero in the IFC output stream occurs only separately and is always followed by a non zero symbol. This property can be exploited by an appropriate EC model, which switches to a different context after a zero has occurred. This context switching improves the compression rate.

3.6 The M03 algorithm

3.6.1 Basic properties

In 2003 Michael Maniscalco developed a new post BWT stage based on context properties of the BWT called M03 [28]. The algorithm was presented in 2004 at the comp.compression group [26] by a rough draft, which described some basics of the new algorithm [27]. An implementation of M03 with source code is available from Atsushi Komiya [56]. In this paper a complete description of the algorithm is published for the first time. The main idea of M03 is to divide the BWT output data into several context substrings iteratively until each substring contains only one kind of symbol. These intervals determine the BWT output data completely. The M03 stage is not a normal GST stage, because the output has not a global structure but still many local properties. Therefore, the M03 stage includes the entropy coding stage in order to take the utmost advantage of these properties.

A simple example using the string *ABRAKADABRA* will be used to elucidate the modus operandi of M03. The columns *F* and *L* of the string *ABRAKADABRA* are listed in Table 10. Column *L* together with the BWT index 2 represent the BWT output of the input string *ABRAKADABRA*. The output of the M03 encoder is written below in bold letters.

Table 10
Column F and L of ABRAKADABRA.

Index	**F**	**L**
0	A	R
1	A	D
2	A	A
3	A	K
4	A	R
5	B	A
6	B	A
7	D	A
8	K	A
9	R	B
10	R	B

Table 11
Decoder model after the first iteration of M03

Index	**1**
0	A
1	A
-> 2	A
3	A
4	A
5	B
6	B
7	D
8	K
9	R
10	R

M03 uses an iterative process by dividing column *L* into several sequences. During each iteration, every sequence of the former iteration is divided into smaller subsequences. The number of subsequences for each sequence is given by the number of different symbols inside the respective interval at column *L* for that sequence. The length of each subsequence is determined by the frequency count of the respective symbol inside that interval. If a subsequence contains only one kind of symbol, it is completely determined and is not divided any more.

At the beginning, M03 transmits the BWT index and the number of substrings of the first iteration. The first iteration of M03 divides the complete interval of column *L*. The number of substrings is given by the number of different symbols at column *L*. In the present example the interval contains five different symbols: *A*, *B*, *D*, *K* and *R*. These symbols are transmitted. For each of these symbols a subsequence is created. The length of each subsequence is determined by the frequency of the respective symbol at column *L*. These frequencies are transmitted in lexicographical order of the symbols.

Up to now, the output of M03 consists of the following data:

- **2**	(BWT index)
- **5**	(number of substrings of the first iteration)
- **A**, **B**, **D**, **K**, **R**	(symbols of substrings of the first iteration)
- **5**, **2**, **1**, **1**, **2**	(frequency counts of the symbols of the first iteration)

During the decoding process, the decoder builds a model from the already transmitted data. After the first iteration, the model of the decoder in column 1 is equal to column *F* as depicted in Table

Table 12
Decoder model after the second iteration of M03
(symbols at the final position are printed in bold)

Index	**1**	**2**
0	A	A
1	A	D
-> 2	A	K
3	A	R
4	A	R
5	B	**A**
6	B	**A**
7	D	**A**
8	K	**A**
9	R	**B**
10	R	**B**

11.
In the second iteration of M03, the five subsequences of the first iteration are further divided. For each symbol, the frequency count at column *L* inside every subsequence is output. The first subsequence has a length of 5 and reaches from index 0 to index 4. At column *L*, the symbol *A* has a frequency count of 1 inside this interval. The second subsequence has a length of 2 and reaches from index 5 to index 6. The frequency count of symbol *A* is 2 inside this interval. The third subsequence has a length of 1 and reaches from index 7 to index 7 with a frequency count of 1 for symbol *A*. The fourth subsequence has a length of 1 and reaches from index 8 to index 8 with a frequency count of 1 for symbol *A*. Finally, the fifth and last subsequence has a length of 2 reaching from index 9 to index 10 with a frequency count of 0 for symbol *A*. Accordingly, the frequency counts of the other four symbols are calculated for all subsequences.
The output of the second iteration of M03 is determined by the frequency counts of the symbols in lexicographical order:

- **1**, **2**, **1**, **1**, **0**	(frequency counts of symbol *A* of second iteration)
- **0**, **0**, **0**, **0**, **2**	(frequency counts of symbol *B* of second iteration)
- **1**, **0**, **0**, **0**, **0**	(frequency counts of symbol *D* of second iteration)
- **1**, **0**, **0**, **0**, **0**	(frequency counts of symbol *K* of second iteration)
- **2**, **0**, **0**, **0**, **0**	(frequency counts of symbol *R* of second iteration)

After transmitting these frequency counts, the decoder is able to construct the model presented in

Table 13
Decoder model after the third iteration of M03
(symbols at the final position are printed in bold)

Index	**1**	**2**	**3**
0	A	A	**R**
1	A	D	**D**
-> 2	A	K	**A**
3	A	R	K
4	A	R	R
5	B	**A**	**A**
6	B	**A**	**A**
7	D	**A**	**A**
8	K	**A**	**A**
9	R	**B**	**B**
10	R	**B**	**B**

column 2 of Table 12. Note that the symbols of subsequence 2 until subsequence 5 from index 5 to index 10 contain only one kind of symbol and therefore are completely determined. These symbols found their final destination inside column *L*. The symbols of the first subsequence from index 0 to index 4 are not completely determined and need to be further sorted.

The third iteration of M03 transmits the frequency counts for the symbols, which are not located at their final position, i.e. the symbols of the first subsequence. The first subsequence consists of four different symbols, therefore four new subsequences are created. The first, second and third of the new subsequences each contain one symbol, the fourth and last of the new subsequences contains two symbols.

The output of the third iteration of M03 transmits the frequency counts of the symbols in lexicographical order:

- **0**, **0**, **1**, **0**	(frequency counts of symbol *A* of third iteration)
- **0**, **1**, **0**, **0**	(frequency counts of symbol *D* of third iteration)
- **0**, **0**, **0**, **1**	(frequency counts of symbol *K* of third iteration)
- **1**, **0**, **0**, **1**	(frequency counts of symbol *R* of third iteration)

The decoder builds the model presented in column 3 of Table 13. Since the first three of the new subsequences each contain only one symbol, they are completely determined. The fourth of the new subsequences contains two symbols and need to be further sorted. Note that the order of the two symbols could be *KR* as well as *RK*.

The fourth iteration of M03 transmits the frequency counts for subsequence between index 3 and index 4 in lexicographical order:

- **1**, **0**	(frequency counts of symbol *K* of third iteration)
- **0**, **1**	(frequency counts of symbol *R* of third iteration)

After the fourth iteration of M03 the complete column L is determined as listed in Table 10.
Note that no RLE stage is used behind the BWT stage in order to keep the full context of the BWT output unchanged.

3.6.2 Improvements

The basic M03 scheme described before, can be improved in several ways by cutting of data, which is already determined and therefore redundant.
At each iteration, the total number of symbols inside each subsequence is known beforehand to the encoder as well to the decoder. Therefore, after the sum of the transmitted frequencies reaches the total number of symbols inside that subsequence, no more information needs to be transmitted, which makes the trailing zeros at the sequence of frequency counts unnecessary.
For example, the original output of the second iteration consisting of

- **1**, **2**, **1**, **1**, **0**	(frequency counts of symbol *A* of second iteration)
- **0**, **0**, **0**, **0**, **2**	(frequency counts of symbol *B* of second iteration)
- **1**, **0**, **0**, **0**, **0**	(frequency counts of symbol *D* of second iteration)
- **1**, **0**, **0**, **0**, **0**	(frequency counts of symbol *K* of second iteration)
- **2**, **0**, **0**, **0**, **0**	(frequency counts of symbol *R* of second iteration)

can be reduced to

- **1**, **2**, **1**, **1**	(frequency counts of symbol *A* of second iteration)
- **0**, **0**, **0**, **0**, **2**	(frequency counts of symbol *B* of second iteration)
- **1**	(frequency counts of symbol *D* of second iteration)
- **1**	(frequency counts of symbol *K* of second iteration)
- **2**	(frequency counts of symbol *R* of second iteration)

without any loss of information.
Furthermore, the output of the frequency counts can be compressed by an arithmetic coder. For each frequency count to output, the maximum of that frequency count is known beforehand and decreases along the output sequence. Adjusting the length of the intervals inside the model of the arithmetic coder for the possible numbers to encode leads to shorter output sequences.
For example, the frequency counts of symbol *A* of the second iteration consists of the sequence:

- **1**, **2**, **1**, **1**	(frequency counts of symbol *A* of second iteration)

leading to the following possible output ranges:

- **1** (possible 0, 1, 2, 3, 4, 5), **2** (possible 0, 1, 2, 3, 4), **1** (possible 0, 1, 2,), **1** (possible 0, 1).

3.7 Other post BWT Stages

Beside the aforementioned schemes, there have been more Global Structure Transformation (GST) stages published. One scheme which is close to the IF stage described above is the Distance Coding (DC) algorithm from Binder [52, 53]. The DC algorithm is based on the Interval Encoding scheme from Elias [49]. For each symbol of the input sequence, the DC algorithm outputs the distance to the next occurrence of the same symbol. If the symbol does not occur again, a zero is output. Binder proposed three improvements to the basic algorithm [52]. If the length of the input sequence is transmitted too, the last sequence of ending zeros is redundant. Furthermore, for calculating the distance to the next occurrence of the same symbol, only unknown symbols have to be counted. The last improvement is that if the last symbol is equal to the current symbol, nothing has to be output and DC proceeds to the next symbol. The main difference to the Interval Encoding of Elias is that DC does not count known symbols and skips repeated symbols. Further analysis, theoretical properties and efficiency of the MTF, DC and IF stages are studied in 2007 by Gagie and Mancini [57], who present improved versions of the three stages, which are locally optimal on low-entropy strings.

Another approach for post BWT stages is the direct coding of the BWT output. Balkenhol and Shtarkov described an approach in 1999, which handles the BWT output as a concatenation of uniform fragments, i.e. the symbol distribution is changing in discrete intervals and not continuously [16]. A uniform fragment is a symbol sequence with a constant probability distribution of the symbols [16].

A switching scheme between different post BWT stages using the snake algorithm was developed by Chapin in 2000 [20]. Different post BWT stages were paired. The best result was achieved with the Best x of $2x - 1$ algorithm and a variant of a sticky MTF algorithm [20].

Wirth and Moffat discussed in 2001 direct symbol encoding similar to PPM techniques instead of GST schemes [58, 59]. They used a hierarchical model similar to the one from Balkenhol and Shtarkov [16]. Also, Ferragina, Giancarlo and Manzini, used in 2006 an RLE stage together with an EC stage based on an order-zero arithmetic coder without any GST stage, called RleAc, and received strong results [60].

A post BWT compression scheme based on wavelet trees was introduced 2004 by Foschini, Grossiy, Guptaz and Vitter [61]. They used a wavelet tree in conjunction with RLE and gamma encoding instead of an arithmetic coder and proposed the compression format WZIP. Further theoretical analysis of wavelet trees in BWT compression is provided by Ferragina, Giancarlo and Manzini in 2006 [54], who improved the asymptotic performance by presenting the so called generalized wavelet trees.

A context based approach, which uses the properties of the BWT output in order to calculate information of the original context of the BWT input string, was presented by Deorowicz in 2005 and called context exhumation. By transmitting the frequency of the alphabet symbols beside the normal BWT output, the algorithm is able to calculate context information of the original BWT input string during the encoding and decoding of the BWT output string similar to M03 [62].

More details of post BWT stages are discussed by Adjeroh, Bell and Mukherjee in 2008 [42].

4. Results

4.1 Compression Rates

For the comparison of compression rates, the following algorithms are itemized:

- GZIP93-V1.2.4 with option -9 for highest compression – from Jean-loup Gailly and Mark Adler, based on LZ77 [63],
- BW94 – from Michael Burrows and David Wheeler, based on BWT [1],
- F96 – from Peter Fenwick, based on BWT [24],
- BS99 – from Bernhard Balkenhol and Yuri Shtarkov, based on BWT [16],
- D02 – from Sebastian Deorowicz, based on BWT [53],
- MTF06 – from Jürgen Abel, described in section Move-To-Front,
- IF06 – from Jürgen Abel, described in section Inversion Frequencies,
- WFC06 – from Jürgen Abel, described in section Weighted Frequency Count,
- IFC06 – from Jürgen Abel, described in section Incremental Frequency Count,
- M03C – from Atsushi Komiya [56, 64], described in section M03.

The corresponding files are derived from four different file collections, all available at http://www.data-compression.info/Corpora/. The first three collections, the Calgary Corpus [65], the Canterbury Corpus and the large Canterbury Corpus [66] contain different file types and are the most popular file sets for lossless compression benchmarks. The last file collection, the Lukas 2D 16 Bit Medical Image Corpus [41], is a set of large two dimensional 16 bit radiographs in TIF format and represents the impact of different post BWT stages in the imaging field. Medical image compression is an important field of lossless data compression. Files of the Lukas 2D 16 Bit Medical Image Corpus have typical sizes around 5 MB. Compression rates are presented in bits per symbol (bps) for all algorithms.

For all files but the large files of the Lukas Corpus, the M03 implementation achieves the best compression rates, followed by WFC06 and IFC06. For larger files, especially for the large medical image files of the Lukas Corpus, IF06 achieves in most cases the best results, followed by WFC06 und IFC06. On average M03, WFC06, IFC06 and IF06 attain the strongest results, but there is one exception worth mentioning. The file `kennedy.xls`, which is called `excl` by Fenwick [51], obtains with M03 by far the best result, which is double as good as the results of WFC06, IFC06 and IF06. It seems that the context based method with recursive intervals of M03 offers a better accommodation than the ranking schemes with indices.

It is interesting to note, that the superiority of the M03 implementation becomes smaller with larger files. Maybe the bit streams to encode inside the entropy coding stage, e.g. the different order-n bits of a data stream could be arranged in separated order-n groups so that more context information could be exploited. The context information between the bit streams are small compared to the context information of the symbols, but for larger files their influence rises.

The last 5 columns of Table 17 present the results for the Lukas Corpus if the BWT input data is not reversed for binary files as described in section Basic Concepts. For all BWCA schemes and all files, the results are worse than with context reversing. The radiographs are image files with 16 bit pixels. Their values are stored with the most significant byte first (big endian). Using the preceeding context inside the BWT achieves about 5% better results on average for this type of file.

4.2 Compression and Decompression Times

Running time results are calculated as the average over ten runs measured in seconds on a 2.13 GHz Pentium M with 2 GB RAM running under WINDOWS XP. All I/O times including loading and linking of the programs are included. Time variations among different runs were negligible.

In about 90% of all cases, the GZIP93 algorithm achieves the highest compression speed. For the last 10%, GZIP is close behind MTF06, IFC06 and IF06 with two exceptions. The files `kennedy.xls` and `E.coli` are compressed three times faster by MTF06, IF06 and IFC06 than by GZIP93. Both files contain many repeating strings, which are handled faster by a sorting scheme like BWCA than by a dictionary scheme like LZ77.

For decompression, GZIP is always the fastest algorithm, on average about three times as fast as MTF06. IFC06, which achieves a better compression rate than MTF06, is about 10% slower than MTF06. IF06 is about 10% slower on average than IFC06. M03, which obtains in most cases the best compression rates, is up to ten times slower than the other BWT based implementations.

Table 14: Compression rates in bps for the Calgary Corpus.
Best compression rates are printed in bold font

File	GZIP93	BW94	F96	BS99	D02	MTF06	IF06	WFC06	IFC06	M03C
bib	2.516	2.02	1.95	1.91	1.896	1.912	1.919	1.882	1.887	**1.829**
book1	3.256	2.48	2.39	2.27	2.274	2.320	2.231	2.249	2.257	**2.199**
book2	2.702	2.10	2.04	1.96	1.958	1.981	1.933	1.931	1.941	**1.881**
geo	5.355	4.73	4.50	4.16	4.152	4.236	4.161	4.105	**4.098**	4.165
news	3.072	2.56	2.50	2.42	2.409	2.449	2.405	2.380	2.406	**2.321**
obj1	3.839	3.88	3.87	3.73	3.695	3.765	3.892	**3.676**	3.712	3.710
obj2	2.628	2.53	2.46	2.45	2.414	2.423	2.487	2.394	2.403	**2.291**
paper1	2.792	2.52	2.46	2.41	2.403	2.414	2.417	2.377	2.386	**2.335**
paper2	2.880	2.50	2.41	2.36	2.347	2.373	2.340	2.326	2.336	**2.276**
pic	0.816	0.79	0.77	0.72	0.717	0.748	0.709	**0.706**	0.722	0.712
progc	2.679	2.54	2.49	2.45	2.431	2.454	2.473	2.413	2.429	**2.348**
progl	1.807	1.75	1.72	1.68	1.670	1.683	1.703	1.655	1.666	**1.582**
progp	1.812	1.74	1.70	1.68	1.672	1.665	1.719	1.654	1.662	**1.576**
trans	1.611	1.52	1.50	1.46	1.452	1.446	1.506	1.430	1.441	**1.377**
Avg.	**2.697**	**2.40**	**2.34**	**2.26**	**2.249**	**2.276**	**2.278**	**2.227**	**2.239**	**2.186**

Table 15: Compression rates in bps for the Canterbury Corpus.
Best compression rates are printed in bold font

File	GZIP93	MTF06	IF06	WFC06	IFC06	M03C
alice29.txt	2.849	2.192	2.148	2.149	2.152	**2.093**
asyoulik.txt	3.118	2.462	2.404	2.409	2.411	**2.372**
cp.html	2.594	2.400	2.436	2.353	2.372	**2.304**
fields.c	2.249	2.062	2.173	2.060	2.067	**2.001**
grammar.lsp	2.670	2.509	2.672	2.498	2.503	**2.464**
kennedy.xls	1.579	0.620	1.034	0.816	0.978	**0.444**
lcet10.txt	2.704	1.940	1.892	1.893	1.900	**1.841**
plrabn12.txt	3.229	2.336	2.253	2.272	2.275	**2.220**
ptt5	0.816	0.748	0.709	**0.706**	0.722	0.712
sum	2.672	2.563	2.683	2.503	2.521	**2.451**
xargs.1	3.320	3.096	3.187	**3.062**	3.098	3.075
Avg.	**2.527**	**2.084**	**2.145**	**2.066**	**2.091**	**1.998**

Table 16: Compression rates in bps for the large Canterbury Corpus. Best compression rates are printed in bold font

File	GZIP93	MTF06	IF06	WFC06	IFC06	M03C
`bible.txt`	2.330	1.508	1.453	1.463	1.471	**1.423**
`E.coli`	2.244	1.989	1.964	**1.954**	1.973	2.017
`world192.txt`	2.337	1.333	1.309	1.298	1.309	**1.259**
Avg.	**2.304**	**1.610**	**1.575**	**1.572**	**1.584**	**1.566**

Table 17 Compression rates in bps for the Lukas 2D 16 Bit Medical Image Corpus. The last 5 columns contain the results for a BWT with no context reversing (following context). Best compression rates are printed in bold font

File	GZIP93	MTF06	IF06	WFC06	IFC06	M03C	MTF06 flw.cnt.	IF06 flw.cnt.	WFC06 flw.cnt.	IFC06 flw.cnt.	M03C flw.cnt.
`lukas_2d_16_breast_0`	2.146	1.232	**1.170**	1.200	1.190	1.185	1.253	1.211	1.220	1.219	1.211
`lukas_2d_16_breast_1`	2.483	1.404	**1.336**	1.376	1.358	1.351	1.428	1.387	1.394	1.390	1.389
`lukas_2d_16_foot_0`	2.865	1.392	**1.349**	1.371	1.356	1.351	1.412	1.389	1.406	1.383	1.373
`lukas_2d_16_foot_1`	3.850	1.869	**1.823**	1.830	1.824	1.832	1.981	1.957	2.031	1.945	1.902
`lukas_2d_16_hand_0`	3.313	1.723	1.653	**1.647**	1.664	1.677	1.806	1.777	1.777	1.762	1.721
`lukas_2d_16_hand_1`	3.273	1.629	1.566	**1.557**	1.573	1.584	1.717	1.698	1.693	1.676	1.630
`lukas_2d_16_head_0`	2.221	1.124	**1.083**	1.091	1.092	1.093	1.134	1.110	1.110	1.108	1.113
`lukas_2d_16_head_1`	2.659	1.358	**1.311**	1.317	1.319	1.319	1.375	1.344	1.344	1.344	1.343
`lukas_2d_16_knee_0`	3.170	1.513	1.462	**1.459**	1.469	1.473	1.585	1.556	1.604	1.545	1.515
`lukas_2d_16_knee_1`	3.370	1.628	**1.573**	1.582	1.579	1.589	1.700	1.670	1.728	1.659	1.629
`lukas_2d_16_leg_0`	2.967	1.593	**1.547**	1.551	1.557	1.551	1.662	1.620	1.646	1.609	1.608
`lukas_2d_16_leg_1`	3.606	1.943	**1.889**	1.894	1.901	1.895	2.021	1.975	2.033	1.963	1.973
`lukas_2d_16_pelvis_0`	4.642	2.755	**2.624**	2.643	2.676	2.666	2.746	2.641	2.643	2.675	2.679
`lukas_2d_16_pelvis_1`	4.424	2.497	**2.386**	2.390	2.427	2.410	2.563	2.475	2.482	2.488	2.481
`lukas_2d_16_sinus_0`	3.589	1.708	1.668	1.681	**1.663**	1.679	1.829	1.816	1.865	1.800	1.751
`lukas_2d_16_sinus_1`	3.641	1.760	1.718	1.720	**1.714**	1.728	1.882	1.868	1.907	1.851	1.808
`lukas_2d_16_spine_0`	3.922	2.444	**2.334**	2.399	2.380	2.370	2.506	2.431	2.459	2.436	2.451
`lukas_2d_16_spine_1`	3.899	2.484	**2.365**	2.372	2.407	2.385	2.514	2.424	2.421	2.434	2.438
`lukas_2d_16_thorax_0`	4.344	2.384	**2.264**	2.324	2.314	2.300	2.429	2.339	2.353	2.360	2.373
`lukas_2d_16_thorax_1`	4.020	2.198	**2.107**	2.110	2.142	2.133	2.248	2.190	2.209	2.194	2.205
Avg.	**3.420**	**1.832**	**1.761**	**1.776**	**1.780**	**1.779**	**1.890**	**1.844**	**1.866**	**1.842**	**1.830**

Table 18: Compression and decompression times in seconds for the Calgary Corpus. Fastest times are printed in bold font

File	comp. GZIP 93	comp. MTF 06	comp. IF 06	comp. WFC 06	comp. IFC 06	comp. M03C	decp. GZIP 93	decp. MTF 06	decp. IF 06	decp. WFC 06	decp. IFC 06	decp. M03C
`bib`	**0.02**	0.06	0.07	0.10	0.07	0.20	**0.02**	0.05	0.05	0.08	0.05	0.12
`book1`	**0.22**	0.25	0.29	0.52	0.29	1.88	**0.08**	0.22	0.28	0.47	0.25	1.25
`book2`	**0.14**	0.21	0.24	0.39	0.23	1.41	**0.06**	0.15	0.20	0.33	0.17	1.03
`geo`	**0.09**	0.11	0.11	0.40	0.11	0.27	**0.02**	0.08	0.08	0.37	0.08	0.14
`news`	**0.08**	0.17	0.19	0.34	0.20	0.83	**0.03**	0.11	0.14	0.27	0.13	0.50
`obj1`	**0.01**	0.06	0.06	0.11	0.06	0.06	**0.01**	0.03	0.03	0.08	0.03	0.03
`obj2`	**0.08**	0.13	0.16	0.32	0.14	0.53	**0.02**	0.08	0.13	0.29	0.10	0.31
`paper1`	**0.02**	0.08	0.09	0.11	0.09	0.10	**0.01**	0.05	0.06	0.06	0.05	0.06
`paper2`	**0.02**	0.09	0.09	0.13	0.09	0.16	**0.01**	0.05	0.06	0.08	0.06	0.10
`pic`	0.20	**0.13**	0.14	0.19	**0.13**	5.28	**0.02**	0.09	0.11	0.16	0.09	0.99
`progc`	**0.01**	0.08	0.08	0.09	0.08	0.08	**0.01**	0.05	0.05	0.06	0.05	0.05
`progl`	**0.02**	0.09	0.09	0.11	0.09	0.14	**0.01**	0.05	0.05	0.07	0.05	0.08
`progp`	**0.02**	0.08	0.08	0.09	0.08	0.11	**0.01**	0.05	0.05	0.06	0.05	0.07
`trans`	**0.02**	0.09	0.09	0.11	0.09	0.19	**0.01**	0.05	0.06	0.08	0.05	0.11
Sum	**0.95**	**1.63**	**1.78**	**3.01**	**1.75**	**11.24**	**0.32**	**1.11**	**1.35**	**2.46**	**1.21**	**4.84**

Table 19: Compression and decompression times in seconds for the Canterbury Corpus. Fastest times are printed in bold font

File	comp. GZIP 93	comp. MTF 06	comp. IF 06	comp. WFC 06	comp. IFC 06	comp. M03	decp. GZIP 93	decp. MTF 06	decp. IF 06	decp. WFC 06	decp. IFC 06	decp. M03C
`alice29.txt`	**0.03**	0.08	0.09	0.13	0.09	0.30	**0.02**	0.05	0.06	0.11	0.06	0.17
`asyoulik.txt`	**0.03**	0.08	0.08	0.13	0.08	0.25	**0.02**	0.05	0.06	0.09	0.06	0.14
`cp.html`	**0.01**	0.06	0.06	0.08	0.06	0.05	**0.01**	0.03	0.03	0.05	0.03	0.03
`fields.c`	**0.01**	0.08	0.08	0.08	0.08	0.02	**0.01**	0.03	0.03	0.03	0.03	0.02
`grammar.lsp`	**0.01**	0.08	0.08	0.08	0.08	**0.01**	**0.01**	0.03	0.03	0.03	0.03	**0.01**
`kennedy.xls`	1.06	0.35	**0.33**	2.98	0.38	2.03	**0.08**	0.25	0.27	2.87	0.28	0.89
`lcet10.txt`	**0.09**	0.17	0.19	0.30	0.19	0.94	**0.04**	0.11	0.14	0.23	0.13	0.70
`plrabn12.txt`	**0.16**	0.20	0.23	0.37	0.23	1.10	**0.05**	0.14	0.19	0.31	0.17	0.66
`ptt5`	0.20	**0.13**	0.14	0.19	**0.13**	5.28	**0.02**	0.09	0.11	0.16	0.09	1.00
`sum`	**0.02**	0.06	0.06	0.11	0.07	0.08	**0.01**	0.05	0.06	0.09	0.05	0.05
`xargs.1`	**0.01**	0.08	0.08	0.08	0.08	**0.01**	**0.01**	0.04	0.05	0.05	0.04	**0.01**
Sum	**1.63**	**1.37**	**1.42**	**4.53**	**1.47**	**10.07**	**0.28**	**0.87**	**1.03**	**4.02**	**0.97**	**3.68**

Table 20: Compression and decompression times in seconds for the large Canterbury Corpus. Fastest times are printed in bold font

File	comp. GZIP 93	comp. MTF 06	comp. IF 06	comp. WFC 06	comp. IFC 06	comp. M03C	decp. GZIP 93	decp. MTF 06	decp. IF 06	decp. WFC 06	decp. IFC 06	decp. M03C
`bible.txt`	**1.19**	1.47	1.67	2.31	1.63	11.44	**0.33**	0.97	1.18	1.79	1.10	6.18
`E.coli`	6.54	**1.91**	2.51	3.26	2.25	12.31	**0.41**	1.49	1.56	2.80	1.81	6.11
`world192.txt`	**0.50**	0.78	0.86	1.28	0.86	6.38	**0.20**	0.56	0.70	1.05	0.64	3.48
Sum	**8.23**	**4.16**	**5.04**	**6.85**	**4.74**	**30.13**	**0.94**	**3.02**	**3.44**	**5.64**	**3.55**	**15.77**

Table 21: Compression and decompression times in seconds for the Lukas 2D 16 Bit Medical Image Corpus. Fastest times are printed in bold font.

File	comp. GZIP 93	comp. MTF 06	comp. IF 06	comp. WFC 06	comp. IFC 06	comp. M03C	decp. GZIP 93	decp. MTF 06	decp. IF 06	decp. WFC 06	decp. IFC 06	decp. M03C
`lukas_2d_16_breast_0`	1.65	1.48	**1.47**	3.47	1.56	23.36	**0.65**	1.80	2.35	3.77	1.88	8.90
`lukas_2d_16_breast_1`	**1.14**	1.56	1.58	3.90	1.67	22.16	**0.70**	1.80	2.37	4.10	1.88	7.02
`lukas_2d_16_foot_0`	**1.13**	1.37	2.09	2.59	1.66	15.52	**0.63**	1.54	2.65	2.73	1.78	7.96
`lukas_2d_16_foot_1`	**0.89**	1.26	1.62	2.50	1.41	11.02	**0.56**	1.22	1.84	2.42	1.32	5.58
`lukas_2d_16_hand_0`	**1.30**	1.40	1.56	2.71	1.55	11.33	**0.55**	1.28	2.08	2.56	1.42	6.52
`lukas_2d_16_hand_1`	**1.20**	1.41	1.58	2.66	1.56	12.99	**0.56**	1.29	2.09	2.51	1.44	5.74
`lukas_2d_16_head_0`	**0.82**	0.96	1.38	1.82	1.12	23.06	**0.47**	1.15	1.77	1.98	1.29	7.49
`lukas_2d_16_head_1`	**0.92**	1.07	1.53	2.11	1.25	38.68	**0.51**	1.25	1.97	2.26	1.39	7.68
`lukas_2d_16_knee_0`	**0.95**	1.30	1.49	2.50	1.45	12.89	**0.59**	1.27	2.02	2.45	1.40	6.85
`lukas_2d_16_knee_1`	**0.95**	1.37	1.89	2.50	1.53	11.71	**0.63**	1.35	2.06	2.44	1.48	6.64
`lukas_2d_16_leg_0`	**0.71**	0.77	0.87	1.84	0.86	8.29	**0.35**	0.86	1.27	1.89	0.92	4.12
`lukas_2d_16_leg_1`	**0.51**	0.70	0.78	1.66	0.77	5.89	**0.31**	0.73	1.10	1.66	0.78	3.27
`lukas_2d_16_pelvis_0`	**1.52**	2.16	3.17	4.75	2.53	15.78	**0.91**	2.29	3.86	4.80	2.60	9.01
`lukas_2d_16_pelvis_1`	**1.30**	2.05	2.39	5.42	2.19	14.09	**0.87**	2.12	2.77	5.59	2.20	7.09
`lukas_2d_16_sinus_0`	**1.09**	2.41	2.78	3.63	2.55	10.73	**0.57**	1.21	1.85	2.47	1.33	5.67
`lukas_2d_16_sinus_1`	**1.28**	1.39	1.55	2.62	1.53	11.14	**0.54**	1.18	1.86	2.41	1.28	5.98
`lukas_2d_16_spine_0`	**0.71**	1.12	1.16	2.91	1.20	7.80	**0.49**	1.09	1.59	2.86	1.16	4.50
`lukas_2d_16_spine_1`	**0.75**	1.15	1.55	2.59	1.28	8.41	**0.48**	1.13	1.77	2.53	1.24	4.52
`lukas_2d_16_thorax_0`	**1.47**	2.34	2.42	6.16	2.50	16.38	**0.97**	2.36	3.37	6.28	2.50	7.87
`lukas_2d_16_thorax_1`	**1.11**	1.69	1.97	4.08	1.80	12.66	**0.76**	1.82	2.36	4.19	1.89	6.70
Sum	**21.40**	**28.96**	**34.83**	**62.42**	**31.97**	**293.89**	**12.10**	**28.74**	**43.00**	**61.90**	**31.18**	**129.11**

5. Conclusions

The Burrows-Wheeler Compression Algorithm (BWCA) achieves good compression rates combined with high speed. Within this field, the post BWT stages, i.e. those that follow the Burrows-Wheeler Transformation (BWT) stage, play a central role in order to realize the best possible results. Implementations of post BWT stages typically consist of three parts: a Global Structure Transformation (GST), a Run Length Encoding (RLE) stage and an Entropy Coder (EC) stage. Context based approaches often include the GST and EC stage and skip the RLE stage.

This paper compares the function and results of different post BWT stages: Move-To-Front (MTF), Inversion Frequencies (IF), Weighted Frequency Count (WFC), Incremental Frequency Count (IFC) and M03. All versions except the last one have placed an RLE stage in front of the post BWT stage as shown in Figure 3 and all used the improvements described in the respective sections.

For the Calgary, Canterbury and large Canterbury corpora, the M03 based BWCA achieves the best compression rates. The IFC variant of the BWCA, offering good compression rates for all sizes of the files, has a higher throughput than the M03 based BWCA and is almost as fast as an MTF based BWCA.

For large 16 bit medical images with big endian, the IF based BWCA achieves the best compression rates with context reversing, together with a moderate speed.

At this point of time, it is difficult to predict if further improvements to recency ranking based GST algorithms will give better compression rates or whether the development of context based approaches will lead to better compression rates. In any case, the speed of the context based approaches must be improved in order to be comparable with other BWCA and PPM based implementations.

6. Acknowledgements

Special thanks go to William Teahan, who provided extensive comments on this work, and to Michael Maniscalco, who discussed the details of the M03 algorithm. The RLE discussions with Yaakov Gringeler and several remarks by Sebastian Deorowicz, Szymon Grabowski, Uwe Herklotz and Vadim Yoockin as well as the M03 implementation of Atsushi Komiya are appreciated very much.

7. References

[1] Burrows, M, Wheeler, D. A Block-Sorting Lossless Data Compression Algorithm. Technical report, Digital Equipment Corporation, Palo Alto, California, 1994, URL (March 2006): http://citeseer.ist.psu.edu/76182.html.

[2] Anderson, A, Nilsson, S. A New Efficient Radix Sort. 35th Symposium on Foundations of Computer Science, 714–721, 1994.

[3] Anderson, A, Nilsson, S. Implementing Radixsort. The ACM Journal of Experimental Algorithmics, Volume 3, Article 7, 1998.

[4] Fenwick, P. Block Sorting Text Compression. ACSC'96, Melbourne, 1996.

[5] Kurtz, S. Reducing the Space Requirement of Suffix Trees. Software – Practice and Experience, 29(13), 1149–1171, 1999.

[6] Kurtz, S, Balkenhol, B. Space Efficient Linear Time Computation of the Burrows and Wheeler-Transformation. In Numbers, Information and Complexity. I. Althöfer, N. Cai, G. Dueck, L. Khachatrian, M. Pinsker, A. Sarközy, I. Wegener, and Z. Zhang, Eds. Kluwer Academic Publishers, 375-383, 2000.

[7] Sadakane, K. Improvements of Speed and Performance of Data Compression Based on Dictionary and Context Similarity. Master's thesis, Department of Information Science, Faculty of Science, University of Tokyo, Japan, 1997, URL (March 2006): http://citeseer.ist.psu.edu/sadakane97improvements.html.

[8] Sadakane, K. Unifying Text Search and Compression – Suffix Sorting, Block Sorting and Suffix Arrays, Ph.D. Dissertation, Department of Information Science, Faculty of Science, University of Tokyo, 2000, URL (March 2006): http://citeseer.ist.psu.edu/sadakane00unifying.html.

[9] Larsson, N. Structures of String Matching and Data Compression. PhD thesis, Department of Computer Science, Lund University, Sweden, 1999, URL (March 2006): http://citeseer.ist.psu.edu/larsson99structures.html.

[10] Seward, J. On the performance of BWT sorting algorithms. Proceedings of the IEEE Data Compression Conference 2000, Snowbird, Utah, J. Storer and M. Cohn, Eds., 173–182, 2000.

[11] Itoh, H, Tanaka, H. An Efficient Method for in Memory Construction of Suffix Arrays. Proc. IEEE String Processing and Information Retrieval Symposium (SPIRE'99), 81–88, September 1999.

[12] Kao, T. Improving Suffix-Array Construction Algorithms with Applications, Master's thesis, Gunma University, Kiryu, 376–8515, Japan, 2001, URL (March 2006): http://citeseer.ist.psu.edu/692550.html.

[13] Manzini, G, Ferragina, P. Engineering a Lightweight Suffix Array Construction Algorithm. Lecture Notes in Computer Science, Springer Verlag, Volume 2461, 698–710, 2002.

[14] Kärkkäinen, J, Sanders, P. Simple Linear Work Suffix Array Construction. 30th International Colloquium on Automata, Languages and Programming, number 2719 in LNCS, 943–955. Springer, 2003.

[15] Schindler, M. A Fast Block-sorting Algorithm for lossless Data Compression. In Proceedings of the IEEE Data Compression Conference 1997, Snowbird, Utah, J. Storer and M. Cohn, Eds., 469, 1997.

[16] Balkenhol, B, Shtarkov, Y. One attempt of a compression algorithm using the BWT. SFB343: Discrete Structures in Mathematics, Falculty of Mathematics, University of Bielefeld, Preprint, 99–133, 1999, URL (March 2006): http://citeseer.ist.psu.edu/balkenhol99one.html.

[17] Arnavut, Z, Magliveras, S. Block Sorting and Compression. Proceedings of the IEEE Data Compression Conference 1997, Snowbird, Utah, J. Storer and M. Cohn, Eds. 181–190, 1997.

[18] Arnavut, Z. Move-to-Front and Inversion Coding. ," In Proceedings of the IEEE Data Compression Conference 2000, Snowbird, Utah, J. Storer and M. Cohn, Eds. 193, 2000.

[19] Arnavut, Z. Inversion Coder. The Computer Journal, 47(1), 46-57, 2004.

[20] Chapin, B. Switching Between Two On-line List Update Algorithms for Higher Compression of Burrows-Wheeler Transformed Data. In Proceedings of the IEEE Data Compression Conference 2000, Snowbird, Utah, J. Storer and M. Cohn, Eds. 183-192, 2000.

[21] Deorowicz, S. Improvements to Burrows-Wheeler Compression Algorithm. Software – Practice and Experience, 30(13), 1465–1483, 2000.

[22] Abel, J. A fast and efficient post BWT-stage for the Burrows-Wheeler Compression Algorithm. Proceedings of the IEEE Data Compression Conference 2005, Snowbird, Utah, J. Storer and M. Cohn, Eds., 449, 2005.

[23] Abel, J. Incremental frequency count - a post BWT-stage for the Burrows-Wheeler compression algorithm. Software – Practice and Experience, 37(3), 247-265, 2007.

[24] Fenwick, P. The Burrows-Wheeler Transform for Block Sorting Text Compression -- Principles and Improvements. The Computer Journal, 39(9), 731-740, 1996.

[25] Fenwick, P. Block-Sorting Text Compression - Final Report. The University of Auckland, New Zealand, Technical Report 130; 1996.

[26] Maniscalco, M. comp.compression group. http://groups.google.com/group/comp.compression/browse_thread/thread/7555a0bc297d6691/460f529474809a92?q=#460f529474809a92. 2004.

[27] Maniscalco, M. A solution for context based blocksort compression - The M03 algorithm. http://www.michael-maniscalco.com/papers/m03.pdf . 2004.

[28] Maniscalco, M. The M03 algorithm. Private correspondence. 2009.

[29] Grabowski, S. Text Preprocessing for Burrows-Wheeler Block-Sorting Compression. In VII Konferencja Sieci i Systemy Informatyczne - Teoria, Projekty, Wdrozenia, Lodz, Poland, 1999.

[30] Kruse, H, Mukherjee, A. Improving Text Compression Ratios with the Burrows-Wheeler Transform. In Proceedings of the IEEE Data Compression Conference 1999, Snowbird, Utah, J. Storer and M. Cohn, Eds. 536, 1999.

[31] Franceschini, R, Kruse, H, Zhang, N, Iqbal, R, Mukherjee, A. Lossless, Reversible Transformations that Improve Text Compression Ratios. Project paper, University of Central Florida, USA, 2000.

[32] Awan, F, Zhang, N, Motgi, N, Iqbal, R, Mukherjee, A. LIPT: A reversible lossless text transform to improve compression performance. In Proceedings of the IEEE Data Compression Conference 2001, Snowbird, Utah, J. Storer and M. Cohn, Eds. 481, 2001.

[33] Isal, R, Moffat, A. Parsing Strategies for BWT Compression. In Proceedings of the IEEE Data Compression Conference 2001, Snowbird, Utah, J. Storer and M. Cohn, Eds. 429–438, 2001.

[34] Isal, R, Moffat, A, Ngai, A. Enhanced Word-Based Block-Sorting Text Compression. In Proceedings of the twenty-fifth Australasian conference on Computer science, Volume 4, January 2002, 129–138, 2002.

[35] Abel, J, Teahan, W. Universal Text-Preprocessing for Data Compression. IEEE Transactions on Computers, 54(5), 497-507, 2005.

[36] Abel, J. Record Preprocessing for Data Compression. In Proceedings of the IEEE Data Compression Conference 2004, Snowbird, Utah, J. Storer and M. Cohn, Eds. 521, 2004.

[37] Balkenhol, B, Kurtz, S. Universal Data Compression Based on the Burrows-Wheeler Transformation: Theory and Practice. IEEE Transactions on Computers, 49(10), 1043–1053, 1998.

[38] Bentley, J, Sleator, D, Tarjan, R, Wei, V. A locally adaptive data compression scheme. Communications of the ACM, 29, 320–330, 1986.

[39] Cleary, J, Witten, I. Data compression using adaptive coding and partial string matching. IEEE Transactions on Communications, 32(4), 396–402, 1984.

[40] Sayood, K. (Editor) Lossless Compression Handbook. Academic Press, 2003.

[41] Abel, J. Lukas 2D 16 Bit Medical Image Corpus – A set of two dimensional 16 bit radiographs in TIF format. URL (March 2006): http://www.data-compression.info/Corpora/LukasCorpus/.

[42] Adjeroh, D, Bell, T, Mukherjee, A. The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching. Springer New York, 2008.

[43] Maniscalo, M. A Run Length Encoding Scheme For Block Sort Transformed Data. Technical paper, 2000, URL (March 2006): http://www.geocities.com/m99datacompression/papers/rle/rle.html.

[44] Maniscalo, M. A Second Modified Run Length Encoding Scheme For Block Sort Transformed Data. Technical paper, 2001, URL (March 2006): http://www.geocities.com/m99datacompression/papers/rle2.html.

[45] Chapin, B. Higher Compression from the Burrows-Wheeler Transform with new Algorithms for the List Update Problem, Ph.D. Dissertation, University of North Texas, 2001.

[46] Gringeler, Y. Private correspondence, 2002.

[47] Abel, J. Advanced blocksorting compressor (ABC). 2003, URL (March 2006): http://data-compression.info/ABC/

[48] Nelson, M, Gailly, JL. The Data Compression Book, Second Edition, M&T Books, New York, 113-136, 1996.

[49] Elias, P. Interval and Recency Rank Source Coding: Two On-Line Adaptive Variable-Length Schemes. IEEE Transactions on Information Theory, Vol. 21 (2), 194–203, 1987.

[50] Balkenhol, B, Kurtz, S, Shtarkov, Y M. Modifications of the Burrows and Wheeler Data Compression Algorithm. Proceedings of the IEEE Data Compression Conference 1999, Snowbird, Utah, J. Storer and M. Cohn, Eds. 188–197, 1999.

[51] Fenwick, P. Burrows Wheeler Compression with Variable Length Integer Codes. Software – Practice and Experience, 32(13), 1307–1316, 2002.

[52] Binder, E. Distance Coder, Usenet group: comp.compression, 2000, URL (March 2006): http://groups.google.com/groups?selm=390B6254.D5113AD2%40T-Online.de.

[53] Deorowicz, S. Second step algorithms in the Burrows-Wheeler compression algorithm. Software – Practice and Experience, 32(2), 99–111, 2002.

[54] Ferragina, P, Giancarlo, R, Manzini, G. The myriad virtues of wavelet trees. International Colloquium on Automata, Languages and Programming (ICALP), Lecture Notes in Computer Science vol. 4051, Venezia, 561-572, 2006.

[55] Abel, J. Improvements to the Burrows-Wheeler compression algorithm: after BWT stages. http://www.data-compression.info/JuergenAbel/Preprints/Preprint_After_BWT_Stages.pdf, 2003.

[56] Komiya, A. m03c – Compression program based on m03. http://. 2009.

[57] Gagie, T, Mancini, G. Move-to-Front, Distance Coding, and Inversion Frequencies Revisited. Lecture Notes in Computer Science vol. 4580, 71-82, 2007.

[58] Wirth, A, Moffat, A. Can We Do without Ranks in Burrows Wheeler Transform Compression?. Proceedings of the IEEE Data Compression Conference 2001, Snowbird, Utah, J. Storer and M. Cohn, Eds. 419, 2001.

[59] Fenwick, P. Burrows–Wheeler compression: Principles and reflections. Theoretical Computer Science, vol. 387(3), 200-219, 2007.

[60] Ferragina, P, Giancarlo, R, Manzini, G. The Engineering of a Compression Boosting Library: Theory vs Practice in BWT Compression. Lecture Notes in Computer Science vol. 4168, 756-767, 2006.

[61] Foschini, L, Grossiy, R, Guptaz, A, Vitter, J, S. Fast Compression with a Static Model in High-Order Entropy. In Proceedings of the IEEE Data Compression Conference 2004, Snowbird, Utah, J. Storer and M. Cohn, Eds. 62, 2004.

[62] Deorowicz, S. Context exhumation after the Burrows-Wheeler transform. Information Processing Letters, vol. 95, 313-320, 2005.

[63] Gailly, JL. GZIP – The data compression program – Edition 1.2.4., 1993, URL (March 2006): http://www.gzip.org.

[64] Komiya, A. m03c. Private correspondence. 2009.

[65] Bell, T, Witten, I, Cleary, J. Modeling for Text Compression. ACM Computing Surveys Vol. 21, 557-591, 1989.

[66] Arnold R, Bell, T. A Corpus for the Evaluation of Lossless Compression Algorithms. Proceedings of the IEEE Data Compression Conference 1997, Snowbird, Utah, J. Storer and M. Cohn, Eds. 201–210, 1997.