Incremental Frequency Count – A post BWT-stage for the Burrows-Wheeler Compression Algorithm

Jürgen Abel
Ingenieurbüro Dr. Abel GmbH,
Lechstrasse 1,
41469 Neuss – Germany
Telephon: +49 2137 999333
Email: juergen.abel@data-compression.info

Summary

The stage after the Burrows-Wheeler Transform (BWT) has a key function inside the Burrows-Wheeler compression algorithm as it transforms the BWT output from a local context into a global context. This paper presents the Incremental Frequency Count stage, a post-BWT stage. The new stage is paired with a run length encoding stage between the BWT and entropy coding stage of the algorithm. It offers high throughput similar to a Move To Front stage, and at the same time good compression rates like the strong but slow Weighted Frequency Count stage. The properties of the Incremental Frequency Count stage are compared to the Move To Front and Weighted Frequency Count stages by their compression rates and speeds on the Calgary and large Canterbury corpora.

Key words: compression, Burrows-Wheeler Transform, BWT

1. Introduction

Within the last decade, the Burrows-Wheeler Compression Algorithm [3] has become one of the key players in the field of universal data compression. The reasons for its success are high compression and decompression speed combined with good compression rates. Many improvements for this algorithm have been presented. Some of them treat the calculation of the Burrows-Wheeler Transform (BWT) itself, some treat the entropy coding of the data stream at the end of the algorithm, and many publications concern the middle part of the algorithm, where the BWT output symbols are prepared for the following entropy coding. This paper reveals a new approach for the middle part that offers excellent compression features.

2. Burrows-Wheeler Compression Algorithm

The basic Burrows-Wheeler Compression Algorithm (BWCA) is divided into three parts, one of which is the actual Burrows-Wheeler Transform (BWT) stage, the second is the Global Structure Transformation (GST) stage, and the third part is the Entropy Coding (EC) stage as pictured in Figure 1. An overview of a BWCA is given by Fenwick in [1].

Figure 1: The basic Burrows-Wheeler Compression Algorithm

In contrast to most other compression approaches, which process the symbols sequentially, the BWCA is block oriented. A file to be compressed is first divided into data blocks of a fixed size and then processed separately by the algorithm of Figure 1. Block sizes range from 1 MB to 10 MB in general [2].

The first stage – the BWT – is the basis of the whole algorithm and performs a permutation of the input symbols of the data block [3]. The symbols are reordered according to their following context. Since many contexts tend to determine their preceding symbols, the BWT produces many runs of repeated symbols. Fast and efficient BWT implementations are presented in several publications such as the papers of Larsson and Sadakane [4], Sadakane [5], Itoh and Tanaka [6], Kao [7] and Kärkkäinen and Sanders [8]. Some implementations prefer to use the preceeding

context similar to PPM [9] and reverse the file before the BWT. Balkenhol et al. treated files with an alphabet size of 256 as binary files and reversed the file before the BWT [13]. Even for binary files, rearranging the symbol order does not always lead to better compression results [1]. It depends on the type of file if rearranging boosts or hampers the compression and the effect on the compression rate is not consistent. In some cases, e.g. for image files with 16 bit pixels, which store pixel values with the most significant byte first (big endian), a simple reversal of the file offers noticeable improvement. This implementation reverses binary files because it offers on average some advantages in the imaging field. Since not all binary files have an alphabet size of 256, e.g. image files with less than 256 colours, files which contain almost all symbols of the alphabet are treated as binary files too and are reversed. The threshold for the alphabet size is not critical for the files of the Calgary corpus because all values between 160 and 256 produce exactly the same results. In this implementation, all files which have an alphabet size higher than 230 are reversed before the BWT.

The second stage transforms the local structure of the BWT output sequence into a global structure, and is therefore called a Global Structure Transformation (GST). Some GST schemes like Inversion Frequencies (IF) from Arnavut and Magliveras [10] or Distance Coding (DC) from Binder [11] use a distance measurement between the occurrence of same symbols, but most GST stages use a recency ranking scheme similar to the Move To Front (MTF) algorithm for the List Update problem [12]. MTF is the most common algorithm for such a GST stage, and was used in the original BWCA approach from Burrows and Wheeler [3]. The MTF stage administers a list of the 256 alphabet symbols and processes the input symbols sequentially. For

each input symbol, the index of that symbol inside the list is output and the symbol is moved to the front of the list. This ensures that symbols which occur often within the near past, are transformed into small indices. Runs of equal symbols are transformed into runs of zeros. The main drawback of the MTF stage is that symbols are always moved to the very front of the list no matter how seldom they have occurred before. This tends to displace other frequent symbols to higher indices, which are more expensive to encode. Many authors have presented improved MTF stages, which are based on a delayed behavior like the MTF-1 and MTF-2 approaches of Balkenhol et al. [13] or a sticky version by Fenwick [14]. Another approach, which achieved a much better compression rate than MTF stages, is the Weighted Frequency Count (WFC) stage presented by Deorowicz [15]. Each time a symbol is processed, the WFC assigns weights to all symbols inside a sliding window of the near past. The weights depend on the distance to the current symbol. Weights of closer symbols are larger than weights of more distant symbols. The weights of the occurrences are summed up for each alphabet symbol and sorted in descending order, i.e. the symbol with the highest sum is placed at the front of the list. After sorting the list, the WFC outputs the index of the current symbol inside the list. This provides smaller indices than the MTF stage on average and leads to very good compression rates. Since the sums need to be recalculated and the list sorted for each input symbol, the WFC scheme has a very high cost of computation [15]. Some approaches try to encode the BWT output directly and without any GST stage as e.g. Giancarlo and Sciortino [16], but most BWCA implementations still use a GST stage in order to obtain the best compression rates.

The last stage is the EC stage, which finally compresses the stream of indices from the

GST stage into a bit stream of small size. Some approaches use Huffman coding like BZIP2 [17], some use variable length codes [14] but most of them, including the present approach, use arithmetic coding, which offers the best compression rates [15].

3. Run Length Encoding

In some BWCA approaches, a Run Length Encoding (RLE) stage is used either in front of the BWT or in front of the EC stage. An RLE stage in front of the BWT is supposed to increase the compression speed because some implementations sort runs of equal symbols very slowly. The usage of an RLE in front of the BWT slightly deteriorates the compression rate [19]. The newer BWT implementations, like the one of Kao [7], are able to sort runs in linear time or even sort the whole array in linear time like the algorithm of Kärkkäinen and Sanders [8], so there is no need to use an RLE anymore in front of the BWT for speed reasons. Besides speed, there is another reason why an RLE in front of the BWT might be useful in some cases. If the block size is small, an RLE stage before the BWT can help to process a larger amount of bytes at one go with the fixed block size of the BWT [20].

Most BWT implementations use an RLE stage in front of the EC stage, since the GST output contains many long runs of zeroes. These runs tend to overestimate the global symbol probability of zeroes inside other parts of the GST output, where zeroes occur only occasionally and that leads to a lower compression. Balkenhol and Shtarkov name this phenomenon "the pressure of runs" [21]. The RLE stage helps to decrease this pressure. One common kind of RLE in front of the EC stage is the Zero Run Transformation (RLE0) from Wheeler [18]. Wheeler suggested coding only the runs

of zeroes and no runs of other symbols, since zero is the symbol with the highest number of runs in front of the EC stage. The run length is incremented by one and all bits of its binary representation except the most significant bit – which is always '1' – are written out explicitly by the symbols '0' and '1' similar to the Elias Gamma code [27]. In order to distinguish between a '1' of the run length encoding and a '1' of the normal symbol alphabet, all symbol values greater than zero are increased by one, resulting in an alphabet size of 257. This way, the run length representation and the rest of the symbols can be decoded uniquely.

Apart from the front of the BWT stage and the front of the EC stage, there is another place where an RLE stage can work very well, which goes back to an idea of Gringeler [22]: directly after the BWT stage. There are two reasons for the new position. Since the length of an RLE output stream is usually smaller than the length of the RLE input stream and since an RLE stage is usually much faster than a GST stage, the whole process becomes faster. The second reason is that the coding of the runs lowers the pressure of runs inside the GST stage and that leads to a more compressible GST output stream [21].

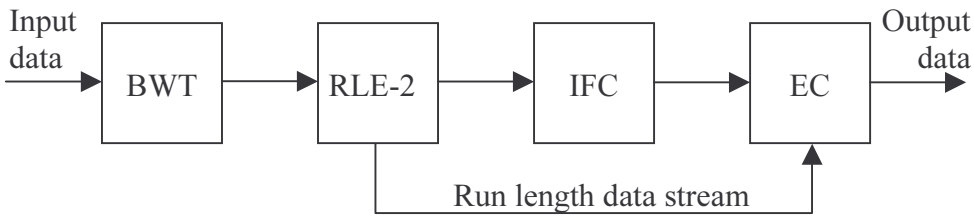

Figure 2: The improved BWCA with an RLE-2 stage after the BWT stage

In this paper, an RLE-2 stage is used after the BWT stage. The RLE-2 stage replaces all runs of two or more symbols by a run consisting of exactly two symbols. In contrast to other approaches, the length of the run is not placed behind the two symbols inside the symbol stream but transmitted into a separate data stream as shown in Figure 2. This way, the length information does not disturb the context of the main data stream. Each run length n inside the run length data stream is encoded by an arithmetic coder, which codes the length of the binary representation first ($\log_{bin}(n)$) and then the actual bits of the binary representation without the most significant bit similar to the Elias Gamma code [27].

Figure 3 shows a typical source code example for the RLE-2 stage. The length information is stored in a separate buffer called `RLE_Buffer`, which is compressed independently from the main output sequence in the `Output_Buffer` afterwards.

```
{ Initialize variables }
Run_Symbol       := -1;
Run_Start_Index  := 0;
Output_Index     := 0;
RLE_Buffer_Index := 0;

{ Loop through input buffer }
For Input_Index := 0 to Buffer_Size -1 do
  Begin { For }
    Current_Symbol := Input_Buffer^ [Input_Index];
    If Current_Symbol = Run_Symbol then
      Begin { then }
        { Inside a run }
        Run_Length := I - Run_Start_Index + 1;
        If Run_Length <= 2 then
          Begin { then }
            { Output run symbol }
            Output_Buffer^ [Output_Index] := Run_Symbol;
            Output_Index                  := Output_Index + 1;
          End; { then }
      End { then }
        else
      Begin { else }
        { Outside a run }
        Run_Length := Input_Index - Run_Start_Index;
        If Run_Length >= 2 then
          Begin { then }
            { Directly behind a run, save run length }
            RLE_Buffer^ [RLE_Buffer_Index] := Run_Length;
            RLE_Buffer_Index               := RLE_Buffer_Index + 1;
          End; { then }

        { Output symbol }
        Output_Buffer^ [Output_Index] := Current_Symbol;
        Output_Index                  := Output_Index + 1;
        Run_Symbol                    := Current_Symbol;
        Run_Start_Index               := Input_Index;
      End; { else }
  End; { For }

{ Encode last run }
Run_Length := Input_Index - Run_Start_Index;
If Run_Length >= 2 then
  Begin { then }
    { Save length }
    RLE_Buffer^ [RLE_Buffer_Index] := Run_Length;
    RLE_Buffer_Index               := RLE_Buffer_Index + 1;
  End; { then }
```

Figure 3: Source code for the RLE-2 stage in PASCAL

4. Incremental Frequency Count

The GST stage builds a global structure from the local structure of the BWT output symbols. The Move To Front (MTF) stage was the first GST stage used and has the advantage of being simple and fast. The disadvantage of the MTF is that it sets each new symbol directly to the front of the list no matter how seldom the symbol has appeared in the near past. The Weighted Frequency Count (WFC) stage of Deorowicz tries to solve this problem by weighting the frequency of all symbols in the near past [15]. Symbols outside the sliding window are not taken into account anymore. By

choosing the proper window size and weights, the WFC achieves very good results [15]. The main disadvantage of the WFC is the high cost of computation, since the weighting of the symbols within the sliding window and the sorting of the list has to be recalculated for each symbol processed.

Below, a new GST stage is presented which weights frequent and close symbols stronger than rare and distant symbols, but which allows a much faster computation of the weighting process than the WFC. A short abstract has been presented at the Data Compression Conference 2005 [23]. Here, a brief overview of the algorithm is given first followed by a more extended description of the separate steps.

Similar to the WFC stage of Deorowicz, the stage presented here uses a list of counters. Each alphabet symbol has a counter assigned. The counters are sorted in descending order, i.e. the highest counter is placed at position 0. Each time a symbol of the input stream is processed, the position of the corresponding counter, i.e. the index inside the counter list, is output. The difference to the WFC stage is the calculation of the counters. The main idea here is to use an adaptive increment, which is recalculated for each symbol processed and added to the counter of the processed symbol. This way, only one counter needs to be resorted inside the list, which makes the process much faster than the one for the WFC stage. In order to weight common and recent symbols stronger than rare and older symbols, the counters can be increased or decreased; on average they are increased. Therefore, the stage is named "Incremental Frequency Count" (IFC). The counters are rescaled frequently in order to prevent overruns and to emphasize closer symbols.

Most parts of the algorithm are simple; the emphasis is set on the calculation of the increment. Figure 4 lists the IFC source code in PASCAL. The IFC consists of one large loop, which processes all symbols of the RLE-2 output stream. The loop is divided into 5 parts:

1. Writing the counter index of the current symbol into the IFC output stream.
2. Calculating the difference between the last index average and the current index average. The index average is calculated for a sliding window of limited size.
3. Calculating the increment depending on the difference and the last increment.
4. Rescaling the increment and the counters if needed.
5. Sorting the counter array.

The following variables and parameters are used:

i	: position inside the input buffer
$index_i$	: index of the corresponding counter for the input symbol at position i
avg_i	: average index value at position i
$window\_size$	: size of the sliding window for the calculation of the average
inc_i	: increment for the input symbol at position i
dif_i	: difference between current and last average
$difl_i$	: limited average between current and last average
dm	: maximum for difference between current and last average
q	: scaling factor for the increment
rb	: boost factor for runs
t	: threshold for rescaling

The first part of the algorithm reads the next symbol of the input stream. The counter index $index_i$ is defined as the position of the corresponding symbol counter inside the list and written out into the IFC output stream.

The second part calculates the difference between the current index average avg_i and the last index average avg_{i-1}. The index average is the average value of the last indices

Table 1

Average compression rates of the Calgary Corpus for different values of window_size

window_size	***1***	***2***	***4***	***8***	***16***	***32***	***64***	***128***
Avg.	2.238	2.238	2.240	2.239	2.240	2.242	2.243	2.243

with focus on the nearest symbols. The average avg_i is calculated by:

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

Table 1 presents the influence of the compression rate on the *window_size*. Larger window sizes make the stage less agile, smaller thresholds make the stage adapt faster to context changes. The optimum window size is between 1 and 16, depending on the type of file. In the present implementation a value of 8 is used for *window_size*.

The third part of the IFC is the most important part: the calculation of the increment inc_i, which is used for the increment of the counter values. In order to obtain an output sequence with indices as low as possible, inc_i has to be chosen very carefully. It is not sufficient to simply increase inc_i by a constant, linear or exponential factor. Instead, inc_i depends on several statistical properties of the previous indices within a sliding window. First of all, the difference between the last value avg_{i-1} and the current value avg_i is calculated:

$$dif_i := avg_i - avg_{i-1} \quad . \tag{2}$$

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:

Table 2

Average compression rates of the Calgary Corpus for different values of dm

dm	2	4	8	16	32	64	128	256
Avg.	2.244	2.244	2.243	2.239	2.239	2.239	2.239	2.239

$$difl_i := \min(|dif_i|, dm) * \text{sign}(dif_i) \ . \tag{3}$$

The limiting of dif_i has a similar effect as the sticky MTF stages [14]: it lowers the influence of large index differences. Note that large differences in the index values occur quite often inside the BWT output, e.g. during the context changes. Table 2 shows the compression rates in dependence of the value of *dm*. Values above 8 result in about the same compression rates and even smaller values only have a small influence on the result. Here, a value of 16 is used for *dm*. Finally, the increment inc_i is calculated by a formula which decreases inc_i for symbols with rising averages, i.e. when a context starts to change, and which increases inc_i for symbols with falling averages, i.e. when a context becomes stable. This way, frequent symbols in a stable context are weighted more strongly than new symbols of a changing context:

$$inc_i := inc_{i-1} - (\frac{inc_{i-1} * difl_i}{q}) \ . \tag{4}$$

The divisor q is a scaling factor. Table 3 lists the results for different values of q. Scaling factors smaller than 64 strongly hamper the compression rates, larger scaling factors slightly hamper. A good compromise is 64, which is used in this implementation.

Table 3

Average compression rates of the Calgary Corpus for different values of q

q	***2***	***4***	***8***	***16***	***32***	***64***	***128***	***256***
Avg.	2.342	2.341	2.321	2.358	2.267	2.239	2.241	2.242

If the current symbol equals the last symbol, a run has occurred inside the RLE-2 output stream. The run length of the BWT output stream has been cut to 2 by the RLE-2 stage and the former run length is not available in the symbol stream of the IFC input. It would be possible to use the information of the run length data stream inside the IFC stage for the reconstruction of the exact run length, but in order to keep the IFC compatible with MTF and WFC stages, runs inside the IFC stage are treated by a different approach. Here, inc_i is boosted by a factor *rb*, if the current symbol equals the last symbol.

$$inc_i := inc_i * rb \ . \tag{5}$$

Table 4 shows the influence of different factors *rb* on the compression result. The best results are obtained for an *rb* value of 1.5. If *rb* is slightly larger or smaller than 1.5, the compression rates deteriorates remarkably. Finally, the counter of the current symbol is increased by inc_i.

Table 4

Average compression rates of the Calgary Corpus for different values of rb

rb	1	1.25	1.375	1.5	1.625	1.75	2	3
Avg.	2.376	2.274	2.242	2.239	2.242	2.244	2.248	2.255

The fourth part of the IFC stage checks for rescaling. All counters and inc_i are halved if the counter of the current symbol is larger than a threshold t. Smaller thresholds have the effect of more frequent rescaling. The value for t is not very critical as to be seen in Table 5. There are only minor differences in compression rates for different thresholds. Here, a value of 256 is used.

In the fifth and last part of the algorithm, the counter list is sorted in descending order, i.e., the highest counter value is placed at index 0. The sorting of the IFC stage can be processed quickly because only one counter has changed since the last symbol and needs to be moved to the proper position inside the list. If other counters contain the same value as the current counter, the current counter is moved to the lowest index position of these counters. Sorting one counter is in contrast to the WFC stage, which sorts the whole list every time a symbol is processed.

Figure 4 presents an example of an IFC source code. Note that the length information of the `RLE_Buffer` is not used inside the IFC stage. Since the RLE-2 stage is in front of the IFC, a zero in the IFC output stream occurs only separately and is always followed by a non zero symbol. That means that after a '0' has appeared, no code space is needed for '0' inside the EC model for the next symbol to encode. This probability exclusion inside the model boosts the compression rate remarkably as to

Table 5

Average compression rates of the Calgary Corpus for different values of t

t	64	128	256	512	1024	2048	4096	8192
Avg.	2.241	2.239	2.239	2.240	2.240	2.241	2.242	2.243

be seen in the next section.

```
{ Initialize variables }
For I := 0 to 256 - 1 do
  Begin { For }
    Symbol_List [I]    := I;
    Symbol_Index [I]   := I;
    Symbol_Counter [I] := 0;
  End; { For }
Last_Symbol := -1;
Avg1        := 0;
Inc1        := 16;

{ Transform symbols to indices }
For Input_Index := 0 to Buffer_Size - 1 do
  Begin { For }
    { Calculate current index }
    Symbol := Input_Buffer^ [Input_Index];
    If Symbol = Last_Symbol then
      Index := 0
        else
      Begin { else }
        If Symbol_Index [Symbol] > Symbol_Index [Last_Symbol] then
          Index := Symbol_Index [Symbol]
            else
          Index := Symbol_Index [Symbol] + 1;
      End; { else }
    Output_Buffer^ [Input_Index] := Index;

    { Calculate difference from average rank }
    Dif1  := Avg1;
    Avg1  := ((Avg1 * (WINDOW_SIZE - 1)) + Index) div WINDOW_SIZE;
    Dif1  := Avg1 - Dif1;

    { Calculate increment }
    If Dif1 >= 0 then
      Begin { then }
        If Dif1 > DM then Dif1 := DM;
        Inc1 := Inc1 - ((Inc1 * Dif1) div 64);
      End { then }
        else
      Begin { else }
        If Dif1 < -DM then Dif1 := -DM;
        Inc1 := Inc1 + ((Inc1 * (-Dif1)) div 64);
      End; { else }

    { Check for run, increment counter }
    If (Symbol = Last_Symbol) then Inc1 := Inc1 + (Inc1 div 2);
    Last_Symbol             := Symbol;
    Symbol_Counter [Symbol] := Symbol_Counter [Symbol] + Inc1;

    { Check for rescaling }
    If Symbol_Counter [Symbol] > T  then
      Begin { then }
        Inc1 := (Inc1 + 1) div 2;
        For I := 0 to 255 do
          Begin { For }
            Symbol_Counter [I] := (Symbol_Counter [I] + 1) div 2;
          End; { For }
      End; { then }

    { Sort list }
    List_Index_Old := Symbol_Index [Symbol];
    If List_Index_Old > 0 then
      Begin { then }
        List_Index_New := List_Index_Old;
        While (List_Index_New > 0) and (Symbol_Counter [Symbol_List [
          List_Index_New - 1]] <= Symbol_Counter [Symbol_List [List_Index_Old]]) do
          Begin { While }
            Dec (List_Index_New);
          End; { While }
        If List_Index_New < List_Index_Old then
          Begin { then }
            { Move list }
            For J := List_Index_Old downto List_Index_New + 1 do
              Begin { For }
                Symbol_List [J]                 := Symbol_List [J - 1];
                Symbol_Index [Symbol_List [J]] := J;
              End; { For }
            Symbol_List [List_Index_New] := Symbol;
            Symbol_Index [Symbol]        := List_Index_New;
          End; { then
      End; { then }
  End; { For }
```

Figure 4: Source code for the IFC stage in PASCAL

5. Entropy Coding

The coding type of the IFC output inside the Entropy Coding (EC) stage has a strong influence on the compression rate. It is not sufficient to compress the index stream just by a simple arithmetic coder with a common order-n context. The index frequency of the IFC output has a non-linear decay as shown in Table 6. Even after the use of an RLE-2 stage, the index 0 is still the most common index symbol on average. As discussed by Fenwick, a hierarchical coding model offers better compression results for skew distributions [18]. Similar to the model of Fenwick, this implementation uses a hierarchical coding model, consisting of three levels as shown in Figure 5 for the index stream of the IFC output.

Table 6
Percentage share of the index frequencies of the IFC output

File	0	1	2	3	4	5	6	7	8	9
Bib	24.63	21.44	9.32	6.47	4.87	4.23	3.38	3.05	2.68	2.27
book1	20.81	28.59	12.15	7.89	5.53	4.22	3.36	2.72	2.29	1.93
book2	23.55	28.10	11.17	6.96	4.95	3.76	3.04	2.52	2.16	1.82
geo	8.94	12.66	7.17	4.53	2.97	2.30	1.74	1.30	1.23	1.06
news	24.48	21.24	9.79	6.54	4.83	3.71	3.16	2.62	2.20	1.90
obj1	17.10	11.66	4.85	3.64	2.81	2.12	1.78	1.74	1.54	1.46
obj2	26.58	18.31	8.28	5.04	3.59	2.66	2.18	1.82	1.54	1.34
paper1	25.45	21.81	9.73	6.94	5.12	3.78	3.13	2.64	2.30	2.11
paper2	23.10	23.54	11.16	7.31	5.68	4.27	3.47	3.00	2.35	2.19
pic	20.42	24.05	7.07	5.10	4.07	3.55	2.98	2.65	2.44	2.15
progc	26.84	21.53	9.45	5.72	4.11	3.20	2.62	2.16	1.95	1.73
progl	31.50	23.54	9.14	5.61	4.01	3.20	2.56	2.22	1.82	1.34
progp	32.90	22.59	9.10	5.74	4.02	2.75	2.24	1.90	1.70	1.41
trans	35.15	18.65	7.99	5.24	3.93	3.04	2.70	2.22	2.03	1.78
Avg.	**24.4**	**21.3**	**9.0**	**5.9**	**4.3**	**3.3**	**2.7**	**2.3**	**2.0**	**1.7**

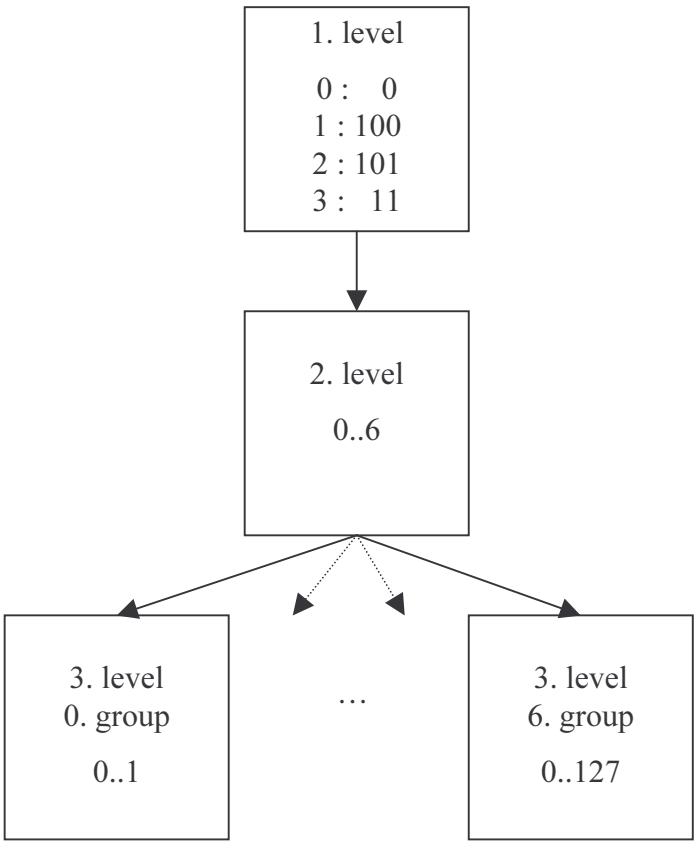

Figure 5: Hierarchical coding model

The first level encodes the most common indices, which range from 0 to 2, and which make up more than 50 percent of the output indices. Larger indices are encoded by all three levels. If the current index is smaller than 3, the information encoded by the first level is sufficient to uniquely decode the index and the following levels are not

Table 7
Coding of the first level

Index	1. binary coder	2. binary coder	3. binary coder
0	0		
1	1	0	0
2	1	0	1
Escape	1	1	

Table 8
Coding of the second level (groups)

Index	Group (output level 2)	Offset (output level 3)	Alphabet size of group
3..4	0	0..1	2
5..8	1	0..3	4
9..16	2	0..7	8
17..32	3	0..15	16
33..64	4	0..31	32
65..128	5	0..63	64
129..256	6	0..127	128

needed. If the index is larger or equal to 3, an escape symbol is output in the first level and the index is categorized into seven disjoint groups, listed in Table 7 and Table 8. The third level handles the offset of the current index in each group as shown in Table 9. [18].

In order to exploit the properties of the IFC output indices in an efficient manner, the

Table 9
Examples of complete index codings

Index	1. Level (binary coding)	2. Level (decimal coding)	3. Level (decimal coding)
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

Table 10
Average compression rates of the Calgary Corpus with and without the output of the first binary coder in case a zero has occurred before the current symbol

Output	WITH OUTPUT	WITHOUT OUTPUT	GAIN IN PERCENTAGE
Avg.	2.257	2.239	0.896

first level uses three binary arithmetic coders instead of one arithmetic coder with an alphabet size of four as shown in Table 7. The first binary coder of the first level encodes, whether the index is zero or greater than zero. Note that a zero inside the IFC output stream occurs only separately. Since the output of the first binary coder is always 1 after a 0 has occurred inside the input stream, the first bit of the level 1 can be omitted in that case. This bit skip makes the omission fast and efficient. Table 10 presents the improvement, which is achieved by omitting the output of the first binary coder in case a zero has occurred before. The gain is around 0.9 percent on average for the files of the Calgary Corpus.

The second level and each group of the third level use own arithmetic coders with respective alphabet sizes and separate contexts. The complete encoding of the hierarchical model is shown in Table 9.

The difference between a BWCA with and a BWCA without an RLE-2 stage in front of the GST stage is listed in Table 11. The gain of the RLE-2 stage is about 0.7 percent.

6.

Table 11
Average compression rates of the Calgary Corpus with and without the RLE-2 stage

RLE stage	With RLE2 stage	Without RLE2 stage	Gain in percentage
Avg.	2.239	2.255	0.721

Results

Table 12 displays the compression rates for different BWT and LZ77 based compression programs on the example of the files of the Calgary Corpus and Table 13 the compression rates for the large Canterbury Corpus. For a better comparison, the compression scheme of Figure 2 is used with an IFC stage, an MTF and a WFC stage with fixed weights. In contrast to the previous approaches, here an RLE-2 stage is used before the MTF and WFC stage respectively. Running time results represent the average over ten runs measured in seconds on a 2.13 GHz Pentium M with 2 GB RAM running under WINDOWS XP and include all I/O times including loading and linking of the programs.

In total, the results for the following algorithms are listed below where available:

Table 12
Compression rates for the Calgary Corpus in bits per symbol

File	GZIP93	BW94	F96	BS99	D02	MTF05	IFC05	WFC05
bib	2.516	2.02	1.95	1.91	1.896	1.912	**1.887**	1.884
book1	3.256	2.48	2.39	2.27	2.274	2.320	**2.257**	2.252
book2	2.702	2.10	2.04	1.96	1.958	1.981	**1.941**	1.934
geo	5.355	4.73	4.50	4.16	4.152	4.236	**4.098**	4.148
news	3.072	2.56	2.50	2.42	2.409	2.449	**2.406**	2.377
obj1	3.839	3.88	3.87	3.73	3.695	3.765	**3.712**	3.676
obj2	2.628	2.53	2.46	2.45	2.414	2.423	**2.403**	2.388
paper1	2.792	2.52	2.46	2.41	2.403	2.414	**2.386**	2.377
paper2	2.880	2.50	2.41	2.36	2.347	2.373	**2.336**	2.328
pic	0.816	0.79	0.77	0.72	0.717	0.748	**0.722**	0.707
progc	2.679	2.54	2.49	2.45	2.431	2.454	**2.429**	2.417
progl	1.807	1.75	1.72	1.68	1.670	1.683	**1.666**	1.656
progp	1.812	1.74	1.70	1.68	1.672	1.665	**1.662**	1.656
trans	1.611	1.52	1.50	1.46	1.452	1.446	**1.441**	1.432
Avg.	**2.697**	**2.40**	**2.34**	**2.26**	**2.249**	**2.276**	**2.239**	**2.231**

- GZIP93–V1.2.4 with option -9 for maximum compression – from Jean-loup Gailly and Mark Adler, based on LZ77 [24],
- BW94 – from Michael Burrows and David Wheeler, based on BWT [3],
- F96 – from Peter Fenwick, based on BWT [18],
- BS99 – from Bernhard Balkenhol and Yuri Shtarkov, based on BWT [21],
- D02 – from Sebastian Deorowicz, based on BWT [19],
- MTF05 – the approach presented in this paper, where the IFC stage of Figure 2 is replaced by an MTF stage,
- IFC05 – the approach presented in this paper, where the scheme with the IFC stage of Figure 2 is used,
- WFC05 – the approach presented in this paper, where the IFC stage of Figure 2 is replaced by a WFC stage with fixed weights.

The results of the last 3 columns of Table 12 and Table 13 represent the BWCA scheme of Figure 2, where the IFC stage is replaced by an MTF and a WFC stage respectively. For the Calgary Corpus, the results of the IFC stage are only 0.008 bps worse than the results of the WFC stage and 0.037 bps better than the results of the MTF stage. For the files of the large Canterbury Corpus, the difference between the IFC and WFC stage is 0.011 bps, whereas the difference between the IFC and MTF stage is 0.026 bps.

Table 13
Compression rates for the large Canterbury Corpus in bits per symbol

File	GZIP93	MTF05	IFC05	WFC05
bible.txt	2.330	1.508	**1.471**	1.458
E.coli	2.244	1.989	**1.973**	1.963
World192.txt	2.337	1.333	**1.309**	1.297
Avg.	**2.304**	**1.610**	**1.584**	**1.573**

Table 14 displays the execution times for the forward and backward transformations of an MTF, an IFC and a WFC stage with fixed weights for the files of the Calgary Corpus. There is no big difference between the time of the forward and backward transformation of each stage. In a standard BWCA, the total speed is limited by the BWT and EC stage. Each of which takes about three to six times more time than the MTF stage, depending on the data structure. Therefore, the total running times of a BWCA based on an MTF stage and a BWCA based on an IFC stage do not differ much as to be seen in Table 15, even though the execution time of the IFC stage is about four times the time of an corresponding MTF stage. The execution time of an WFC stage is about 30 times as high as an MTF stage. Hence, the WFC stage plays an

Table 14
Running times in milliseconds for the GST stages

File	forward MTF	backw. MTF	forward IFC	backw. IFC	forward WFC	backw. WFC
Bib	1.56	1.56	**6.87**	**6.71**	37.66	36.88
book1	13.44	13.91	**56.57**	**55.16**	282.97	280.78
book2	8.90	9.22	**40.16**	**39.22**	199.07	196.88
geo	8.28	8.28	**11.72**	**11.41**	317.97	317.35
news	7.04	7.18	**30.93**	**30.62**	177.03	175.78
obj1	1.25	1.25	**4.07**	**4.22**	52.65	52.50
obj2	6.09	6.25	**26.25**	**25.94**	208.59	207.97
paper1	0.94	0.94	**4.38**	**4.22**	22.97	22.81
paper2	1.40	1.41	**6.41**	**6.09**	32.50	32.03
pic	2.96	2.96	**11.41**	**11.10**	73.91	73.59
progc	0.63	0.78	**3.28**	**3.28**	19.06	18.75
progl	0.78	0.78	**4.53**	**4.37**	19.84	19.53
progp	0.47	0.63	**3.13**	**3.12**	14.38	14.37
trans	0.94	0.93	**5.31**	**5.15**	23.44	23.28
Sum	**54.68**	**56.08**	**215.02**	**210.61**	**1,482.04**	**1,472.50**

important role for the total running time of a WFC based BWCA.

The compression and decompression times of complete MTF, IFC and WFC based BWCAs are compared in Table 15 for the files of the Calgary Corpus and in Table 16 for the files of the large Canterbury Corpus together with the times of the GZIP program. Note that the compression speed of the IFC based BWCA for the Calgary Corpus is only 8% slower than the speed of the MTF based BWCA but 70% faster than the WFC based version. In comparison to GZIP, the IFC version is 31% slower. On the files of the large Canterbury Corpus, the compression speeds of all three BWCAs are even faster than the compression speed of GZIP, mainly because the DNA sequence from the e-coli bacteria file contains only 4 different alphabet symbols. A small number of alphabet symbols is very hard for dictionary based

Table 15
Compression times in seconds for the Calgary Corpus in seconds

File	compr. time GZIP93	decmpr. time GZIP93	compr. time MTF05	decmpr. time MTF05	compr. time IFC05	decmp. time IFC05	compr. time WFC05	decmpr. time WFC05
Bib	0.02	0.02	0.06	0.05	**0.07**	**0.05**	0.11	0.08
book1	0.22	0.08	0.25	0.22	**0.29**	**0.25**	0.50	0.47
book2	0.14	0.06	0.21	0.15	**0.23**	**0.17**	0.39	0.33
geo	0.09	0.02	0.11	0.08	**0.11**	**0.08**	0.41	0.38
news	0.08	0.03	0.17	0.11	**0.20**	**0.13**	0.33	0.27
obj1	0.01	0.01	0.06	0.03	**0.06**	**0.03**	0.11	0.08
obj2	0.08	0.02	0.13	0.08	**0.14**	**0.10**	0.33	0.28
paper1	0.02	0.01	0.08	0.05	**0.09**	**0.05**	0.11	0.07
paper2	0.02	0.01	0.09	0.05	**0.09**	**0.06**	0.13	0.08
pic	0.20	0.02	0.13	0.09	**0.13**	**0.09**	0.19	0.16
progc	0.01	0.01	0.08	0.05	**0.08**	**0.05**	0.09	0.06
progl	0.02	0.01	0.09	0.05	**0.09**	**0.05**	0.11	0.06
progp	0.02	0.01	0.08	0.05	**0.08**	**0.05**	0.09	0.06
trans	0.02	0.01	0.09	0.05	**0.09**	**0.05**	0.11	0.08
Sum	**0.95**	**0.32**	**1.63**	**1.11**	**1.75**	**1.21**	**3.01**	**2.46**

programs to compress [25]. Many strong compression programs achieve compression rates above 2.0 bps [26], whereas all BWCAs here achieve compression rates below 2.0 bps. Again, the IFC based version is only 13% slower than the MTF based version in compression speed but 44% faster than the WFC based version.

During the decompression of the Calgary Corpus, the IFC based BWCA has about one quarter the speed of GZIP. In comparison to the MTF based version, the IFC based version is 11% slower but 73% faster than the WFC based version. During the decompression of the large Canterbury Corpus, the IFC based version is 16% slower than the MTF based version but 59% faster than the WFC based approach.

7. Conclusion and Outlook

An improved BWCA scheme was presented, which offers competitive compression rates in the range of a WFC stage, in combination with a much higher compression and decompression speed. These properties make the IFC stage very attractive for all BWT based compression algorithms. The improved performance of the new scheme is a result of the combination of three features:

- RLE-2 stage with run length separation,

Table 16
Compression times in seconds for the large Canterbury Corpus in seconds

File	compr. time GZIP93	decompr. time GZIP93	compr. time MTF05	decompr. time MTF05	compr. time IFC05	decomp. time IFC05	compr. time WFC05	decompr. time WFC05
bible.txt	1.19	0.33	1.47	0.97	**1.63**	**1.10**	2.28	1.76
E.coli	6.54	0.41	1.91	1.49	**2.25**	**1.81**	3.36	2.88
World192.txt	0.50	0.20	0.78	0.56	**0.86**	**0.64**	1.24	1.05
Sum	**8.23**	**0.94**	**4.16**	**3.02**	**4.74**	**3.55**	**6.88**	**5.69**

- IFC stage,
- EC stage with bit skip.

The RLE-2 stage replaces runs of two or more symbols inside the BWT symbol stream by exactly two symbols, whereas the run length is placed into a different data stream. This way, the length information does not disturb the context of the symbol stream as in common RLE approaches. The gain of the RLE-2 stage is around 0.7 percent in comparison to no RLE stage in front of the GST stage. The IFC stage uses adapting increments and a counter list in order to generate an index stream with low index values. The increment depends on the average index value of the near past. Since for each symbol processed only one counter is changed and resorted, the IFC stage provides a higher throughput per time than the WFC stage. The EC stage uses a hierarchical coding model of three levels and exploits the fact that the next index after a zero must be greater than zero. In that case, the output of the first binary encoder can be skipped, which improves the compression rate by almost one percent.

The parameters and thresholds of the IFC stage presented in this paper are the result of many experiments and different IFC versions. One possibility for improvement might be the use of an array of weights instead of simply limiting the difference as in equation (3). Furthermore, the information of a length of a run could be used from the run length data stream of the RLE-2 stage in order to calculate a more proper weighting than the constant factor of equation (5). This way, the IFC stage would not be replaceable by an MTF or WFC stage but fixed coupled on the RLE-2 stage. Future work in this direction might lead to even better results than the WFC stage offers today.

Acknowledgment

Special thanks go to Brenton Chapin and Szymon Grabowski for several remarks and fruitful discussions on this work.

References

[1] Fenwick, P. Burrows Wheeler Compression. Lossless Data Compression Handbook, K. Sayood, Editor, Academic Press, 182-3, 2003.

[2] Abel, J. Advanced blocksorting compressor (ABC). 2003, URL (December 2005): http://www.data-compression.info/ABC/.

[3] Burrows, M, Wheeler, D J. A Block-Sorting Lossless Data Compression Algorithm. Technical report, Digital Equipment Corporation, Palo Alto, California, 1994, URL (December 2005): http://citeseer.ist.psu.edu/76182.html.

[4] Larsson, N J, Sadakane, K. Faster Suffix Sorting, Technical report, 1999, URL (December 2005): http://citeseer.ist.psu.edu/larsson99faster.html.

[5] 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.

[6] 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.

[7] Kao, T H. Improving Suffix-Array Construction Algorithms with Applications, Master's thesis, Gunma University, Kiryu, 376–8515, Japan, 2001.

[8] 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.

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

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

[11] Binder, E. Distance Coder, Usenet group: comp.compression, 2000, URL (December 2005): http://groups.google.com/groups?selm=390B6254.D5113AD2%40T-Online.de.

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

[13] 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. A. Storer and M. Cohn, Eds. 188–197, 1999.

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

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

[16] Giancarlo, R, Sciortino, M. Optimal Partitions of Strings: A New Class of Burrows-Wheeler Compression Algorithms. Proceedings of Combinatorial Pattern Matching: 14th Annual Symposium, CPM 2003, Morelia, Michoacán, Mexico, 129–143, 2003.

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

[18] Fenwick, P. Block Sorting Text Compression – Final Report, Technical Report 130, University of Auckland, New Zealand, Department of Computer Science, 1996, URL (December 2005): http://citeseer.ist.psu.edu/fenwick96block.html.

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

[20] 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.

[21] Balkenhol, B, Shtarkov, Y M. 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 (December 2005): http://citeseer.ist.psu.edu/balkenhol99one.html.

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

[23] 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. A. Storer and M. Cohn, Eds., 449, 2005.

[24] Gailly, J L. GZIP – The data compression program – Edition 1.2.4., 1993, URL (December 2005): http://www.gzip.org.

[25] Yao, Z, Rajpoot, N. Less Redundant Codes for Variable Size Dictionaries. Proceedings of the IEEE Data Compression Conference 2002, Snowbird, Utah, J. A. Storer and M. Cohn, Eds., 481, 2002.

[26] Powell, M. Table of compression ratio results of the large Canterbury Corpus, URL (December 2005): http://corpus.canterbury.ac.nz/details/large/RatioByLex.html.

[27] Elias, P. Universal Codeword Sets and Representations of the Integers. IEEE Transactions on Information Theory, 21(2), 194–203, 1975