Proceedings of 10th IOE Graduate Conference
Peer Reviewed
ISSN: 2350-8914 (Online), 2350-8906 (Print)
Year: 2021 Month: October Volume: 10

Burrows-Wheeler Post-Transformation with Effective Clustering and Interpolative Coding

Amit Kumar Yadav [a], Sanjeeb Prasad Panday [b]

[a, b] *Department of Electronics and Computer Engineering, Pulchowk Campus, IOE, Tribhuvan University, Nepal*
Corresponding Email: [a] 075mscsk002.amit@pcampus.edu.np [b] sanjeeb@ioe.edu.np

Abstract
Burrows-Wheeler Transformation, in simple words, sorts the data reversibly. Generally, sorted data can't be reversed to its original form but BWT uses the last index to get the original data. In this research the main focus is to manipulate the result of BWT for better compression. The BWT, RLE and MFT are used together. The order of using these may differ according to the requirement. We have used RLE and MFT after BWT in this research. The result of BWT has been passed through RLE module to get the Run characters, Run Length and Run Character frequencies. The Run characters and Run Character Frequencies has been passed through MFT module to get MFT number and final MFT list. Then the MFT number and Run length, which are non-negative integers, has been sorted using counting sort module. This module sorts the numbers according to Run characters. The result is non-negative integer which has been coded using interpolative coding method. This method does not uses any statistics like calculating probabilities to produce final result. Hence the method is also called Non-statistical Coding. The data has been decompressed using the same module as compression but in reverse order. The data is decoded using interpolative decoder and then MFF module has been used to unsort the data and provide Run Characters and Run. These are passed through RLE decoder to get the original file. It results lossless output.

Keywords
BWT, Lossless Compression, Non-statistical Coding, Run Length Encoding, Move to Front coding

1. Introduction

The conversion process of the original data into the compressed data, having smaller size is called Data Compression. The original data can be of any format. It can be a file or bits passing through communication Channel. Coding is a general term which is often used to define data compression. Information theory is characterized as the study of efficient coding and its outcomes, in the form of transmitting speed and possibility of error [1]. Information theory is the base on which compression algorithms have been built.

Lossy compression of data like image, audio and video can be used to obtain highest savings but also at someplace lossless compression is essential. Lossless compression is a technique which makes sure of recovering the original data, while the decompression of lossy methods result an approximation of original data. The compression process usually consists of two main parts: modeling and coding. Modeling usually produces the input for coding, together with the probability distribution, a set of (possibly transformed) items. The source data is transformed in many modelling methods into a series of smaller integer which can be encoded more compactly than the original items [6].

Since the first publication in 1994, researchers in the lossless compression field have been particularly interested in the Burrows Wheeler Transformation (BWT) [2]. Besides the theoretical interest of BWT, highly practical lossless compression schemes can use it as a basis. BWT-based schemes usually compromise between the high compression rate of the prediction by partial matching (PPM) method with the pace of dictionary-based methods such as the variants Lempel-Ziv.

The BWT uses necessary input symbols as the sort key to sort the data uniquely. Similar data are grouped together creating a cluster. The data in same cluster contains same character a number of times and only few symbols can be there based on similar contexts.

The major drawback of BW transformation is that the idea of boarder line between original contexts and context regions is lost. Without using some time consuming and complex techniques like context exhumation method, it is not possible to recover original data from transformed sequence.

The BWT is not a compression algorithm but a technique to prepare data for post-transform algorithms. Usually, the MTF and Run Length Encoding (RLE) is used in series after BWT to achieve compression. In this research, we focus on implementing proper methods as post-transformation stage to achieve better compression. Also the main focus of this research is to obtain better compression than statistical method by using non-statistic coding methods like Binary Interpolation Method.

2. Related Literature

2.1 Local-to-Global Transform

Compression is directly dependent on the context of the source. When it changes, the symbol distribution within the BW transformed data also changes. These changes can perform dramatic effect on the performance of the algorithm. Using entropy coders to react to these changes cannot be enough even when it is adaptive. For this reason, Local to Global Transform (LGT) is employed in most methods. The purpose of this stage is to transform the local structure of the BWT output into global structure which can be considered more stable over the entire file.

In the original publication of BWT, Burrows and Wheeler suggested a recoding method as LGT. They used Move-to-Front (MTF) as LGT. This method converts a given input characters into integer denoting their last position and move it to the front. It is described in more detail in next section. Many authors believe that using this method is adding unnecessary complication. Some has suggested to get rid of it to obtain better compression but at slower rate. The major drawback of using MTF is that the contextual information is lost in the way. But other authors have mentioned that MTF can represent a better compromise between speed and compression rate.

Entropy Coding It is the final stage of the compression method. In this stage the actual compression happens. There are various ways to encode data. One of them is semi fixed coding. In this coding variable length of bits is assigned to each data. Since this research is about integer coding, one of the very popular integer coding called *interpolative coding* has been implemented.

2.2 Semi-fixed Length Coding

When the upper bound of an integer that needs to be coded is not a power of two, Semi-Fixed length coding is used [3]. It's also called "truncated binary coding" by Golomb. It is a prefix code which contains two code word lengths: n and n-1 bits. There are many number of possible assignments of semi-fixed-length codewords to the n integers, but only the following four are used as it is easily computable:

Table 1: Semi Fixed Coding

Number	Low Short	Mid Short	High Short	Mid Long
0	100	0000	0000	100
1	111	0011	0011	0000
2	0000	0100	100	0011
3	0011	0111	111	0100
4	0011	100	0100	0111
5	0100	111	0111	111

2.2.1 Interpolative Coding

One of the non-statistical coding method for bounded coding of non-negative integers is Interpolative coding [4]. This coding technique is simple and very efficient to encode sequence of integers. For all other non-statistical, non-parametric coding methods, interpolative coding is practically superior in case of compression gain. It was originally invented for inverted indexes which consist of strictly increasing sequence of integers but any sequence of integers can be converted into ascending order by computing cumulative values. Conceptually, it can be divided into two stages:

1. For each integer in the source, compute the upper bounds
2. Encoding each integer with either log2(n) or log2(n-1) bits, where n is the upper bound for that integer

3. Methodology

BWT is not a compression algorithm. But it is used to process the data and feed it to other algorithms to compress. Generally, RLE and MFT are used for compression in different order with BWT. Figures 1 and 2 show the order of different modules used in this

research and their order. Also the general algorithm used in this research is given in Algorithm 1.

The compression and decompression works in reverse fashion. For decompression same modules are used but in reverse order. In this research for desorting the MTF numbers a new method has been used called Move-From-Front. This name is given according to its work. Its moves the data from front to its original location using final mtf list.

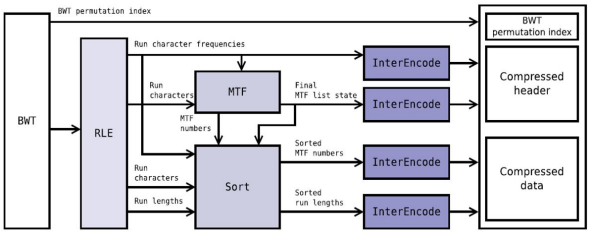

Figure 1: Encoding Mechanism

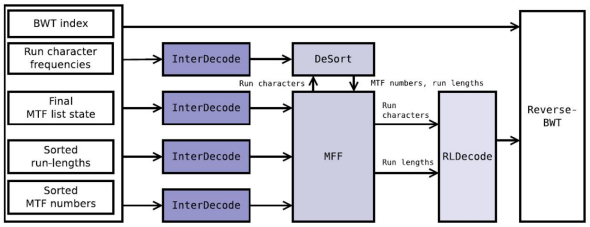

Figure 2: Decoding Mechanism

Changing the context changes the symbol distribution in the BW-transformed data. These changes directly affect the encoder performance. Even the Encoder is adaptive, it may not react smooth enough to adapt such changes. To overcome this effect, a method called Local-to-Global Transform is employed after BWT. This implementation gives more stable spreading of symbols over the complete file. MTF has been used in the original paper of BWT as the LGT. The purpose is to change each character in the input to number of distinctive characters from its previous incidence. The basic algorithm is as below:

Algorithm 1: BWClusteringCoder

Input: *orig* : Input File
Output: *b* : Compressed file
1: (bwt, bwtIndex) ← BWTransform(orig)
2: (runCharacters, runLengths, runCharacterFrequencies) ← RLEncode(bwt)
3: (mtfNumbers, finalMtfListState) ← MTFRecode(runCharacters, runCharacterFrequencies)
4: (sortedMtfNumbers, sortedRunLengths) ← CountingSort(mtfNumbers, runCharacters, runLengths, runCharacterFrequencies)
5: b ← bwtIndex
6: b **append** InterpolativeEncode(finalMtfListState)
7: b **append** InterpolativeEncode(runCharacterFrequencies)
8: b **append** InterpolativeEncode(sortedMtfNumbers)
9: b **append** InterpolativeEncode(sortedRunLengths)

Figure 3: BW Clustering Coder

3.1 Run Length Encoding

The traditional way of run-length encoding has been used. The two-vector approach has been used along with it. Due to interpolative coder characteristics, run characters and their lengths are kept separately. The run characters are converted into MTF numbers by using MTF coder.

The RLE of our post-transformation method is shown in figure 4. The run character frequencies are also computed and stored in alphabetical order. The MTF recoding uses this data to determine unique alphabets. In next section, it is used for computing the indices of sort bin. The method is called clustering based on sorting.

Input	a	a	b	b	a	a	b	b	c	c	d	a	c	d	d	a	a	d	d
Run Characters	a		b		a		b		c		d	a	c	d		a		d	
Run Lengths	2		2		2		2		2		1	1	1	2		2		2	
Run Character Frequencies	4	2	2	3															

Figure 4: Example of Run Length Encoding of BWT Output

3.2 Move-to-Front-Coding

Each run character is converted to the number of distinct characters using MTF since its last presence. The two-vector method of RLE lets to decrements each MTF number by one as we described in above section. The MTF process is explained in Algorithm 2. Figure 6 shows a descriptive example.

Algorithm 2: MTFRecode

Input: *runChar* : Sequence of run characters produced by the two-vector RLE
Input: *runCharFreq*: Global run character frequencies
Output: *mtf*: The MTF numbers
Output: *finalMtfList*: Final state of the MTF list
1: j ← 0
2: last_ix ← length(runChar) − 1
3: // Initialize mtfList
4: **for** i **from** 0 **upto** 255 **do**
5: **if** runCharFreq[i] > 0 **then**
6: mtfList[j] ← *i*
7: *j* ← *j* + 1
8: **end if**
9: **end for**
10: // Perform the MTF recoding
11: **for** i **from** 0 **upto** last_ix **do**
12: *c* ← runChar[*i*]
13: *j* ← 0
14: // Find the position of c in MtfList
15: **while** mtfList[j] != c **do**
16: *j* ← *j* + 1
17: **end while**
18: // Update mtfList by moving c to the front
19: **for** *k* **from** *j* **downto** 1 **do**
20: mtfList[*k*] ← mtfList[*k* − 1]
21: **end for**
22: mtfList[0] ← c
23: // Distinct neighbors: can subtract one from MTF number
24: mtf[*i*] ← *j* − 1
25: **end for**
26: finalMtfList ← mtfList
27: **return** (mtf, finalMtfList)

Figure 5: MTF Coder

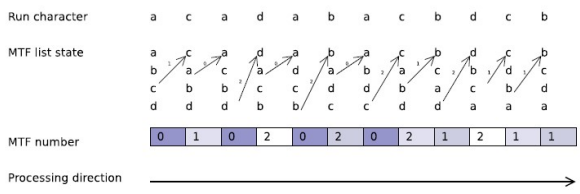

Figure 6: MTF of Run Characters

3.3 Clustering by reversible sorting

MTF numbers are small and run lengths are long for common characters while it's an opposite case for uncommon characters. The triples of [MTF number, run character, run length] can be used to exploit this knowledge by clustering it. It is done by using run characters [5]. Also, it must be remembered that the reversible operation of Clustering should be possible to recover the original triples.

Two arrays are created as output by the encoder. The one output is for MTF numbers and another output is for run lengths. In algorithm 3, we have said that the arrays are divided into bins. Figure 8 shows that the bins are implicit and the joined in alphabetic order. To show the next available position, a pointer is used. Each pointer initially points to the start of the respective bin. The triples, MTF number, run character and run length, are evaluated from left to right. The current run character is used to sort the current run length and MTF number. Finally, the pointers pointing to bin are increased by one, maintaining the stability.

Algorithm 3: CountingSort

Input: runChar : Sequence of run characters produced by the two-vector RLE
Input: runCharFreq: Run character frequencies
Input: mtf: Sequence of MTF numbers produced by the MTF recoding of the run characters
Input: runLen: Sequence of run lengths produced by the RLE
Input: runCount: Total count of runs
Output: sortedRunLen: run lengths sorted by the associated run characters
Output: sortedMtf: MTF numbers sorted by the associated run characters

```
// Compute sort bin indices
binPtr[0] ← 0
for i from 1 upto 255 do
    binPtr[i] ← binPtr[i − 1] + runCharFreq[i-1]
end for

// Loop over the run length and MTF vectors
for i from 0 upto runCount−1 do
    c ← runChar[i]
    m ← mtf[i]
    r ← runLen[i]
    // Sort the MTF number produced by this occurence of c
    sortedMtf[binPtr[c]] ← m
    // Sort the run length produced by this occurence of c
    sortedRunLen[binPtr[c]] ← r
    binPtr[c] ← binPtr[c] + 1
end for
return (sortedMtf, sortedRunLen)
```

Figure 7: Counting Sort

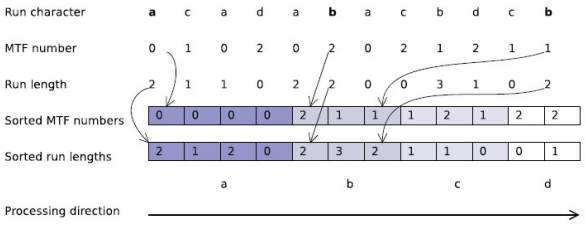

Figure 8: MTF numbers and Run Length sorting

3.4 Interpolative Encoder/Decoder

Mid-short codebook is used by encoder to assign shorter codes to the middle part. In case of sorted MTF numbers and Run lengths, Center short codebook is used for each nodes [6].

The decoder works in reverse order. It takes the combined output of encoder as shown in figure 11. As it can be seen the decoder can decode the encoded inputs but it will be still sorted. So a unsort algorithm has to be applied along with reverse Move-To-Front mechanism which is coined as Move-From-Front.

Algorithm 4: InterpolativeEncoder

Input: Sequence of n nonnegative integers $S[0 \cdots n-1]$
Output: Binary code sequence B

```
for i from 0 to n − 1 do
    T[n + i] ← [S][i]
end for
for i from n − 1 downto 1 do
    T[i] ← T[2 ∗ i] + T[2 ∗ i + 1]
end for
B ← universal-encode(T[1])
for i from 1 to n − 1 do
    bound ← T[i]
    if bound >0 then
        B append semi-fixed-length-encode(T[2 ∗ i], bound)
    end if
end for
return <B>
```

Figure 9: Interpolative Encoder

3.5 Move-From-Front Decoder

The decoder looks at the final state of MTF list and read the character at the front of the list. Then it applies the process called desorting to next number in the MTF list and Run length from the container or bin of that particular character [7]. It reads the MTF number and then moves the character downward in MTF list by that number. This way it operates in reverse order than MTF [8].

Run character frequencies are used by the decoder to reverse the sorting. Since the decoder is operating in reverse order now, the pointer pointing to bins are now set to point to the rear of sort bins. MTF list is updated as well as bin pointer is decremented after desorting. This process continues as iterative process.

The process is shown in Algorithm 5 and MFF transform is shown in Figure 11.

Algorithm 5: MFFTransform

Input: sortedMtf: Sequence of sorted MTF numbers (concatenated sort bins)
Input: sortedRunLen: Sequence of sorted run lengths (concatenated sort bins)
Input: runCharFreq: Run character frequencies
Input: mtfList: Final state of the MTF list after MTF stage
Output: runChar: De-sorted run characters
Output: runLen: De-sorted run lengths
1: runCount ← length(sortedRunLen)
2: // Compute sort bin rear indices for MTF numbers and run lengths
3: binPtr[0] ← runCharFreq[0] − 1
4: **for** i **from** 1 **upto** 255 **do**
5: binPtr[i] ← binPtr[i − 1] + runCharFreq[i]
6: **end for**
7:
8: // Loop over the run length and MTF vectors from end to start
9: **for** i **from** runCount−1 **downto** 0 **do**
10: c ← mtfList[0] ▷ Run character (sort key)
11: m ← sortedMtf[binPtr[c]]
12: r ← sortedRunLen[binPtr[c]]
13: runChar[i] ← c
14: runLen[i] ← r
15: binPtr[c] ← binPtr[c] − 1
16: // Update MTF list: move front character c backwards m+1 steps
17: **for** j **from** 0 **upto** m **do**
18: mtfList[j] ← mtfList[j + 1]
19: **end for**
20: mtfList[m+1] ← c
21: **end for**
22: **return** (runChar, runLen)

Figure 10: MFF Transform

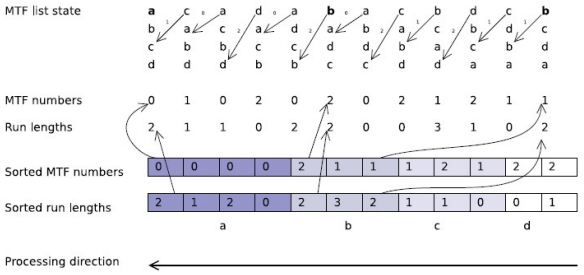

Figure 11: Move-From-Front Transform

4. Experimental Results

This research has been evaluated on standard dataset for data compression: Calgary, Canterbury, Large and Artificial corpora. This research is mainly focused on compressing text files using integer coding technique. So, files which are not text are not included. The compression results are shown below:

File	Original Size (Bytes)	Compression Factor	Bzip2 Compression Factor
bib	111261	4.076092	4.050715
book2	610856	3.999529	3.879855
news	377109	3.257989	3.179671
paper1	53161	3.248854	3.210593
paper2	82199	3.317419	3.282577
paper3	46526	2.957412	2.937804
paper4	13286	2.565856	2.56091
paper5	11954	2.481628	2.471367
paper6	38105	3.120803	3.099984
progc	39611	3.19392	3.157765
progl	71646	4.666884	4.598883
progp	49379	4.6562	4.610551

Figure 12: Compression Result on Calgary Corpus

File	Original Size (Bytes)	Compression Factor	Bzip2 Compression Factor
alice29.txt	152089	3.60665	3.52042
asyoulik.txt	125179	3.19595	3.16356
cp.html	24603	2.95354	3.22705
fields.c	11150	3.5806	3.66897
grammar.lsp	3721	2.85352	2.90023
lcet10.txt	426754	4.07998	3.96221
plrabn12.txt	481861	3.36852	3.31001
sum	38240	2.70859	2.96227
xargs.1	4227	2.38006	2.39898

Figure 13: Compression Result on Canterbury Corpus

File	Original Size (Bytes)	Compression Factor	Bzip2 Compression Factor
bible.txt	4638690	5.21462	5.48545
E.coli	4047392	5.87371	3.23531
world192.	2473400	3.36265	5.05205

Figure 14: Compression Result on large Corpus

File	Original Size (Bytes)	Compression Factor	Bzip2 Compression Factor
a.txt	1	0.05263	0.02703
aaa.txt	100000	1.08421	2127.66
alphabet.txt	100000	833.333	763.359
random.txt	100000	1.19304	1.32128

Figure 15: Compression Result on Artificial Corpus

4.1 Compression Results

The compression depends on the content of files. As it can be seen the compression result is close to bzip2.

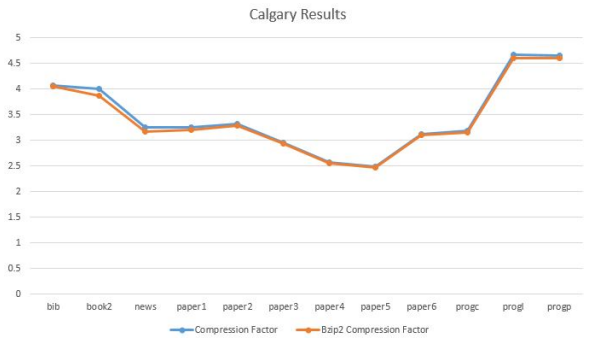

Figure 16: Compression Factor on Calgary Corpus

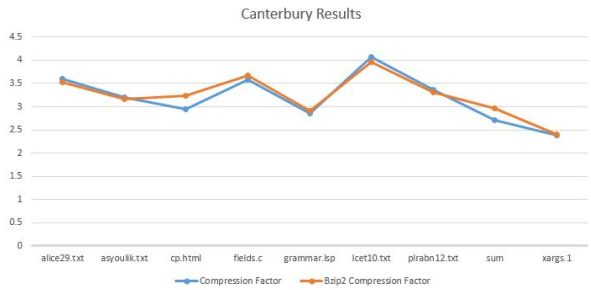

Figure 17: Compression Factor on Canterbury Corpus

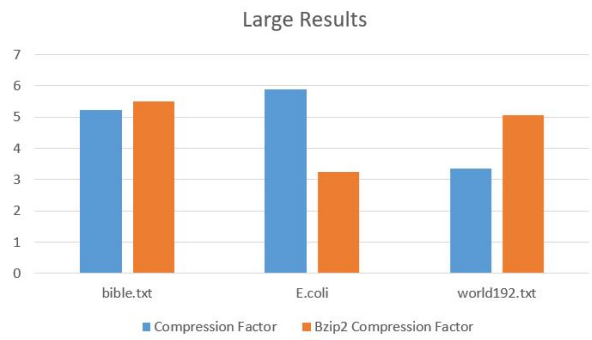

Figure 18: Compression Factor on large Corpus

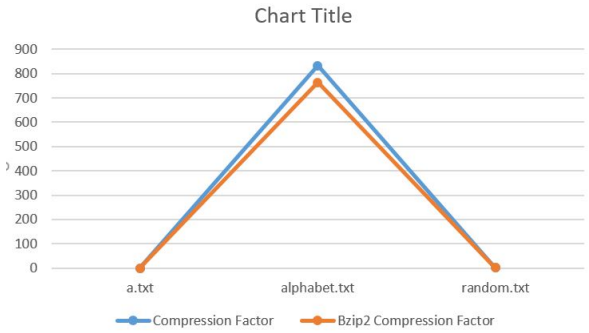

Figure 19: Compression Factor on Artificial Corpus

4.2 Speed

The speed to compression data is almost equal to the bzip2 compression speed. Since this research is mainly focused on obtaining compression factor near to bzip2, speed is not optimized. But still speed is close to optimal. Also, the decompression time is a lot faster than compression time for most of the files. Below we can see the result of Artificial Corpus.

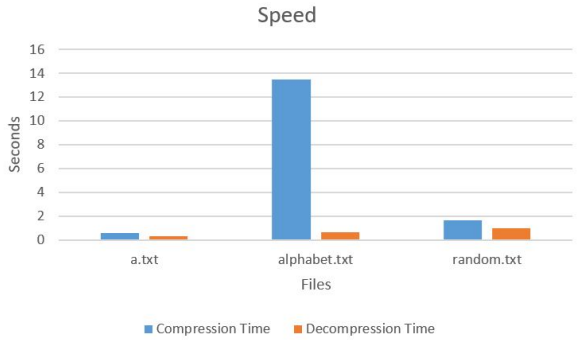

Figure 20: Speed of Artificial Corpus

As we see Decompression speed is a lot more faster than compression speed.

5. Discussion

This research has used non-statistical coding method to perform compression. Due to which the calculation is easy but the time consumption slightly increases with increase in MTF numbers and RLE numbers. Also it works only for alphanumeric data. The performance is not an issue here but it can be better.

The preprocessed data, BW-transformed data, is needed to be used as an input in this research. A better implementation of BWT method reduced time significantly. The traditional method could not be used to compute because of memory Error. But by using Suffix Array it's only a matter of second to get preprocessed data.

Then, the only problem was traversing through the binary tree during interpolative coding and decoding. Better implementation of this coding/ decoding method could result faster and better result than Bzip2.

5.1 Read/ Write Structure

The input file has been preprocessed by passed through BWT module which results the transformed file. This module does not perform the compression but it helps other modules to compress data effectively.

Figure 21: Preprocessing

The index has been passed to final result as it is. First, it is converted into binary format along with the number of bits required to represent it. So,

$$BwtHeader = Binary[index + length(binaryindex)] \quad (1)$$

Then, the Transformed file has been passed through RLE module. This module produces three intermediate files: Run chars, Run length, Run char frequency. The Run chars are passed through MTF module which provides MTF list and MTF Final State. The Run chars, MTF list and Run char frequency are passed Sorting module. This module sorts the MTF list and Run chars to provide effective clusters for compression. The Run char frequency and MTF Final State are passed through Interpolative encoder to generate Compression Header. The compression data is made up of sorted MTF list and sorted Run length.

Since InterEncode module does not know where to stop, the length of leaves are also encoded along with the number of bits needed to represent it. The leaf bit has been used to dynamically code the length of leaves instead of fixed binary representation of leaves. The leaf bit is of 2 bit length which provides the information of how many bits has been used to represent the length of the leaf node. Since the output of InterEncode module is self-explanatory, combining the output of other intermediate is not a problem.

The decompression process starts in reverse order. First the BWT index is taken care of using its leaf bit. Then the remaining file is passed through InterDecode module to list the intermediate files recursively. After the intermediate files are generated they are passed through their respective decoder modules to generate original file. It seems the length of compression files may be heavy. But because of using dynamic binary representation it is less than 1% of the total file. Thus the compression overhead seems reasonable. Table 22shows the components of different files in the testing corpora.

Intermediate Components	bib	Book2	Alice29.txt
BWT Index	16 bits	24 bits	8 bits
MTF_Final_list	657 bits	786 bits	602 bits
Run Char Freq	852 bits	1160 bits	812 bits
Sorted MTF numbers	145402 bits	807003 bits	225284 bits
Sorted Run lengths	71352 bits	412865 bits	110522 bits

Figure 22: Compression Component Sizes

5.2 Time and Space Analysis

The analysis performed on a 64-bit machine having 6 GB Ram and 1 TB Hard disk. The compression algorithm is directly dependent on the size and content of the input file. Similarly, in case of decompression even less space and time is required.

6. Conclusion and Future Work

In this research, a data compression using BWT has been described. The processes that follows the BWT are order and modified to generate better clustering of data. Move-From-Front method has been used to unsort the data reversibly. It can be seen that applying sorting increases the cluster of similar data which becomes beneficial while coding using non-statistical and non-parametric coding method called Interpolative coding. The combination of sorting and Interpolative coding provides us a better compression rate in significant time.

Further in this research, the sorting can be combined with Arithmetic coder to use statistical approach for better compression. As we can see the compression results looks good but there's still room for modification. Also, codebook for interpolative coding can play a vital role in compression. So, choosing the best codebook can be beneficial. Since our final result before coding is non-negative integer, it can be coded using other methods as well which deals with these type of inputs. Hence, this research can be further analyzed for better compression using more suitable methods.

References

[1] Witten I.H. Bell T.C. and J.G Cleary. Modeling for text compression. 1989.

[2] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. 1994.

[3] S.W. Golomb. Run-length encodings. 1966.

[4] A. Moffat and L. Stuiver. Exploiting clustering in inverted file compression. 1996.

[5] A. Moffat and L Stuiver. Binary interpolative coding for effective index compression. 2000.

[6] Luka˘c N. Žalik B., Mongus D. and Žalik K. R. Efficient chain code compression with interpolative coding. 2018.

[7] Teuhola J Niemi A. Burrows-wheeler post-transformation with effective clustering and interpolative coding. 2020.

[8] P.M. Fenwick. *Burrows–Wheeler Compression. In Sayood, K. (ed.), Lossless Compression Handbook*. Academic Press, San Diego, CA, 2003.