Received: 10 April 2019 | Revised: 2 May 2020 | Accepted: 1 June 2020
DOI: 10.1002/spe.2873

RESEARCH ARTICLE

WILEY

Burrows-Wheeler post-transformation with effective clustering and interpolative coding

Arto Niemi | **Jukka Teuhola**

Department of Future Technologies, University of Turku, Turku, Finland

Correspondence
Arto Niemi, Department of Future Technologies, University of Turku, Vesilinnantie 5, 20500 Turku, Finland.
Email: arvani@utu.fi

Summary
Lossless compression methods based on the Burrows-Wheeler transform (BWT) are regarded as an excellent compromise between speed and compression efficiency: they provide compression rates close to the PPM algorithms, with the speed of dictionary-based methods. Instead of the laborious statistics-gathering process used in PPM, the BWT reversibly sorts the input symbols, using as the sort key as many following characters as necessary to make the sort unique. Characters occurring in similar contexts are sorted close together, resulting in a clustered symbol sequence. Run-length encoding and Move-to-Front (MTF) recoding, combined with a statistical Huffman or arithmetic coder, is then typically used to exploit the clustering. A drawback of the MTF recoding is that knowledge of the character that produced the MTF number is lost. In this paper, we present a new, competitive Burrows-Wheeler posttransform stage that takes advantage of interpolative coding—a fast binary encoding method for integer sequences, being able to exploit clusters without requiring explicit statistics. We introduce a fast and simple way to retain knowledge of the run characters during the MTF recoding and use this to improve the clustering of MTF numbers and run-lengths by applying reversible, stable sorting, with the run characters as sort keys, achieving significant improvement in the compression rate, as shown here by experiments on common text corpora.

KEYWORDS
lossless compression, Burrows-Wheeler transform, move-to-front, interpolative coding

1 | INTRODUCTION

Since its first publication in 1994,[1] the Burrows-Wheeler transform (BWT) has received considerable attention from researchers working in the lossless compression field. Besides being of considerable theoretical interest, the BWT can be used as a basis for highly practical lossless compression schemes. BWT-based schemes typically represent a compromise between the high compression rate of the prediction by partial matching (PPM) methods and the speed of dictionary based methods such as the Lempel-Ziv variants. After the release of the industry-strength, highly optimized bzip2 implementation by Seward,[2] Burrows-Wheeler compression gained widespread usage as a general lossless compression tool.

The BWT reversibly sorts the input symbols, using as the sort key as many following characters as necessary to make the sort unique. Symbols having similar contexts are grouped together. Thus, the transformed sequence has many runs of the same character, and typically only a few symbols are possible within a region of similar contexts. A drawback of the transformation is that knowledge of the original contexts and the borders of the context regions is lost. It is not possible to recover these from the transformed sequence alone without resorting to some time-consuming techniques such as the context exhumation method of Deorowicz.[3]

The BWT itself does not result in compression, but makes the data more compressible by the subsequent application of *post-transform* algorithms. In this paper, we focus on improving the post-transformation stage. For more details on the BWT and its implementation techniques, see for example, Fenwick[4] and Adjeroh et al.[5] Also not covered in this paper are preprocessing methods (eg, Kruse et al[6] and Awan et al[7]) that are applied before the BWT. While these methods can result in substantial improvements in compression rate, they are also data-specific, utilizing, for example, the fact that the input is text in the English language.

1.1 | Local-to-Global Transform

When the context changes, so does the symbol distribution within the BW-transformed data. Often, these changes can be quite abrupt and dramatic, for example, when the first character of the context changes from “a” to “b”—characters which, in typical English text, have quite different distributions for the preceding character. Entropy coders, even when adaptive, are unable to react swiftly enough to such changes. For this reason, most methods employ a so-called Local-to-Global Transform (LGT)[5] stage after the BWT. The purpose of the LGT stage is to transform the BWT output’s local structure into a global one that has a more stable distribution over the entire file. In their original publication, Burrows and Wheeler suggested using a simple recency recoding method called *Move-to-Front* (MTF) as the LGT. The idea is to convert each input character to the number of distinct characters since its last occurrence. MTF will be described in more detail in Section 2.2. The MTF step can be called a *recoding*, since each symbol is converted to a non-negative integer, without changing its position in the sequence.

The major disadvantage of MTF and other recency recoding schemes is that knowledge of the character that produced the MTF number is lost. This results in additional loss of contextual information. Many authors have viewed the simple MTF recoding quite sceptically—for example, Fenwick[4] remarked that “the suspicion remains that the Move-To-Front stage is an embarrassing and unnecessary complication”—some have suggested getting rid of it completely, achieving good compression rates, albeit with a significant slowdown.[8] Results by Ferragina et al suggest that MTF may not be as inefficient as previously thought,[9] but may indeed represent an excellent compromise between high compression rate and speed.[10]

The MTF stage converts runs of a character to runs of the integer zero. For text files, zeros make up around 60% of the MTF output. Long runs of zeros can disturb the probability estimation of adaptive coders. This phenomenon is sometimes called “the pressure of runs.”[11] As a countermeasure, run-length encoding (RLE) is often applied to the MTF number sequence before entropy coding. A popular choice is Wheeler’s 0/1 RLE method, which increments other MTF numbers by one, and uses the symbols 0 and 1 to encode run-lengths of the MTF number zero.[12]

Entropy coding. The final stage in any BWT-based compression method is an entropy coder. It is this stage that performs the actual compression. In the original paper, Burrows and Wheeler used a Huffman coder as the final stage for encoding the MTF numbers. In a later research, Wheeler suggested using an arithmetic coder instead to improve the compression rate. The current bzip2 implementation uses a Huffman coder for higher speed and better parallelizability. Most new published algorithms use an arithmetic coder. Both of these coders require symbol probabilities as an input. BWT-specific modeling techniques for the arithmetic coder have been investigated by, for example, Fenwick.[12]

Interpolative coding, first presented by Moffat and Stuiver[13] in 2000, is a *nonstatistical* entropy coding scheme for nonnegative integers. It operates directly on the input numbers, without requiring their probabilities as a parameter. Its main application is inverted index compression, where it is regarded as a benchmark method with excellent compression rate but relatively slow speed.[14] Teuhola[15] provides an equivalent, but computationally more efficient, interpretation of interpolative coding, which we use in this article. Teuhola’s approach allows interpolative coding to be implemented in a cache-friendly manner, making the method substantially faster compared to the original approach used by Moffat[13] and Petri.[14] Besides its main application in inverted indexes, Interpolative coding has previously been successfully applied to the entropy coding of MTF numbers[15] and sorted bilevel images.[16] Recently, Žalik[17] et al used interpolative coding as the final stage of a Burrows-Wheeler-based compression for chain codes. In contrast to the method of Žalik et al, our scheme is intended for generic lossless (text) compression, takes advantage of Teuhola's breadth-first interpolative coding approach, and introduces a new reversible sorting stage between MTF and interpolative coding, substantially improving the clustering and thereby the compression rate.

2 | SUGGESTED POST-BWT STAGES

Our post-BWT compression procedure consists of RLE, move-to-front recoding (MTF), character-based, stable sorting (Sort), and interpolative coding (InterEncode). These four stages are described in the sections below. The basic flow of our method is shown in Figures 1 and 2 and the top-level pseudocode for the encoding process is given in Algorithm 1.

The decompression process consists of the corresponding decoding steps in reverse order. Especially, the desorting of MTF numbers is new and called here the *Move-From-Front* (MFF) transform, due to its characteristic way of operation. Actually, it involves also reversing the MTF recoding as a side effect. Reversible sorting is one of the key ideas of our coder, and studied in more detail below.

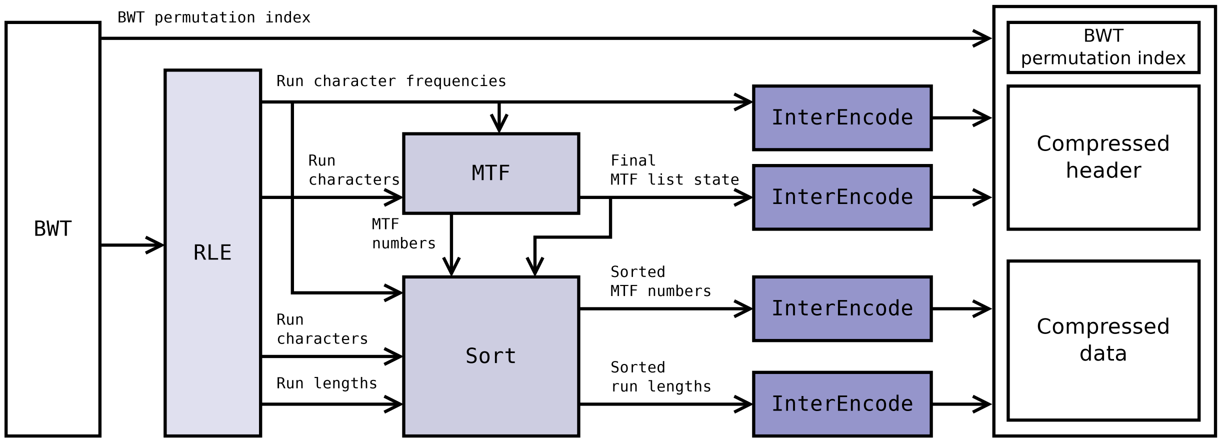

FIGURE 1 Encoding. [Colour figure can be viewed at wileyonlinelibrary.com]

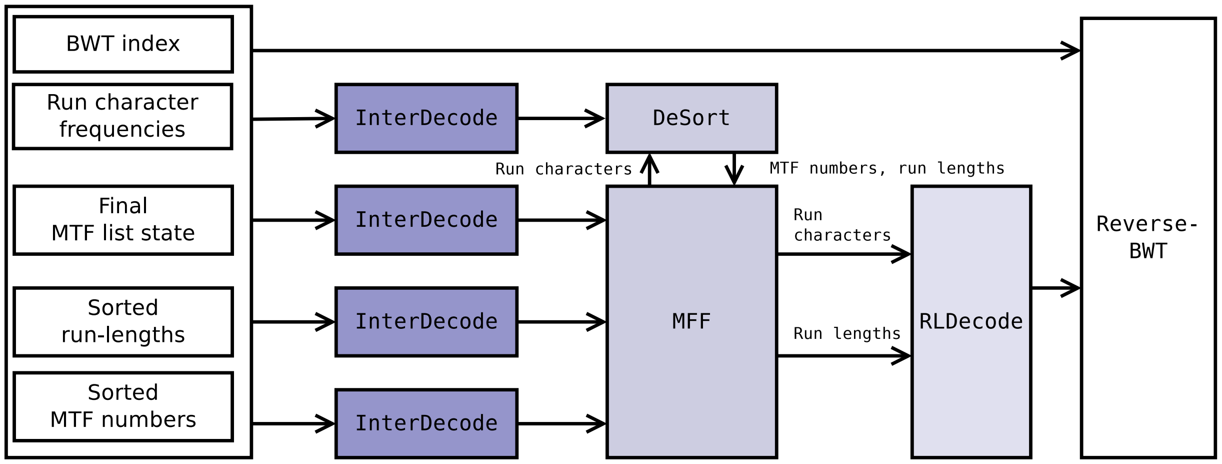

FIGURE 2 Decoding [Colour figure can be viewed at wileyonlinelibrary.com]

Algorithm 1. BWClusteringCoder

```
Input: orig : The original data
Output: b : The compressed data
1: (bwt, bwtIndex) ← BWTransform(orig)
2: (runCharacters, runLengths, runCharacterFrequencies) ← RLEncode(bwt)
3: (mtfNumbers, finalMtfListState) ← MTFRecode(runCharacters, runCharacterFrequencies)
4: (sortedMtfNumbers, sortedRunLengths) ← CountingSort(mtfNumbers, runCharacters, runLengths, runCharacter
   Frequencies)
5: b ← bwtIndex
6: b append InterpolativeEncode(finalMtfListState)
7: b append InterpolativeEncode(runCharacterFrequencies)
8: b append InterpolativeEncode(sortedMtfNumbers)
9: b append InterpolativeEncode(sortedRunLengths)
```

2.1 | Run-length coding

As mentioned in the introduction, most Burrows-Wheeler based compression schemes use RLE. There are three locations where run-length coding can be used: (i) before the BWT, (ii) between the BWT and MTF coding, or (iiii) after the MTF coding.

The advantage of the first option is that it makes the BWT itself faster by reducing the amount of symbols that need to be sorted. This is nowadays deemed unnecessary,[18] due to the discovery of fast, linear-time algorithms for the BWT sorting (eg, Kärkkäinen et al[19]). A drawback of this option is a slight degradation in the compression rate. Most schemes in the literature use the last option, performing run-length coding after MTF. Our method, like the scheme of Abel,[20] uses the middle option, where the RLE stage is inserted between the BWT and the MTF. As a result, MTF recoding is performed only on the run characters—a sequence in which each character is distinct from its neighbours—so each MTF number can be decremented by one. Smaller numbers are compressed more effectively by interpolative coding, so this small optimization is actually quite significant.

In the Burrows-Wheeler compression literature, several variants of the RLE algorithm have been proposed. A major difference is whether to code the run-lengths and run-characters into the same output vector or into different vectors. The former approach is used, for example, in Wheeler's 0/1 RLE described by Fenwick,[21] which is called *Zero Run Transformation* or RL0 in Abel,[18] 2010, while the two-vector approach is used in the RL2 method described in Abel,[22] 2006. Our method also takes the two-vector approach, but with traditional run-length encoding. In our case, it is useful to keep the run lengths and run characters (which are to be converted into MTF numbers) separate due to the characteristics of our final interpolative coder. Interpolative coding can encode clusters of low and high numbers very effectively. Using the single-vector approach would mean mixing low MTF numbers with large run-length numbers (eg, the MTF number zero would often be followed by large a run-length). The two-vector approach, in contrast, preserves the clusters.

The RLE of our posttransformation method is shown in Figure 3. Besides the run characters and run lengths, our run length encoding routine also computes the run character frequencies, storing them in alphabetical order. These are needed by the subsequent MTF recoding to determine the active alphabet, and the counting sort-based clustering method of Section 2.3, in which the run character frequencies are used for computing the sort bin indices.

Input	a	a	a	c	c	a	a	d	a	a	a	b	b	b	a	c	b	b	b	b	d	d	c	b	b
Run characters	a			c		a		d	a			b			a	c	b				d		c		b
Run lengths	2			1		1		0	2			2			0	0	3				1		0		2
Run character frequencies					4	3	3	2																	

FIGURE 3 Example of run-length encoding of Burrows-Wheeler transform output

2.2 | Move-To-Front

As the next step, we apply standard MTF[1,23] recoding to the vector of run characters. The MTF recoding converts each character to the number of *distinct* characters since its last appearance. It can be implemented by maintaining an ordered list of the source alphabet characters, such that each character is moved to the front of the list after recoding. The recoding of a character c is then the *index* of c in the MTF list, a nonnegative integer which we call an MTF number. As the base alphabet, we use the byte values from 0 to 255. Before the MTF recoding process, we initialize the MTF list using the active alphabet, that is, with those run characters that occur at least once in the input. As noted in the previous section, the two-vector RLE allows us to decrement each MTF number by one. The MTF recoding process is detailed in Algorithm 2. Figure 4 contains an example. A small optimization, to be described in Section 2.4, is possible: the first MTF number of each character need not be encoded. This optimization is not used in the examples and pseudocodes.

Move-to-Front has two properties that make it especially suitable for our method. First, it converts the run characters into a sequence of nonnegative integers, which is the data type that our final stage, interpolative coding, operates on. Second, the integers produced by MTF tend to approximately follow Zipf's law,[24,25] which is advantageous for interpolative coding, especially if the numbers appear in clusters. In the next section, we aim to improve the clustering.

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:         $j \leftarrow j + 1$
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:     $j \leftarrow 0$
14:     // Find the position of c in MtfList
15:     while mtfList[j] != c do
16:         $j \leftarrow j + 1$
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)
```

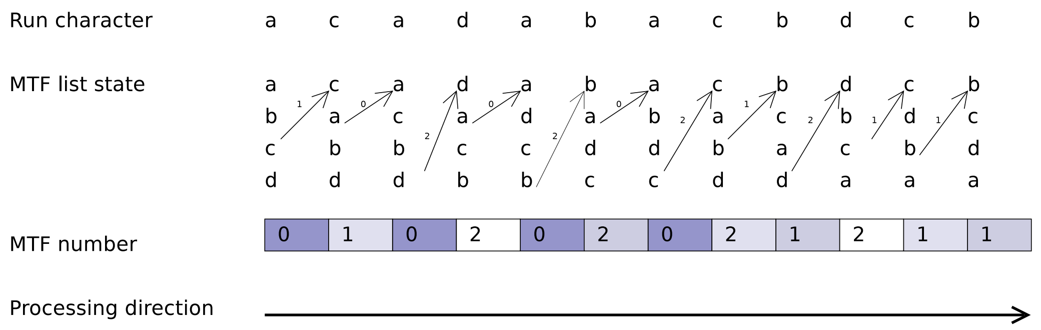

FIGURE 4 Move-to-Front recoding of the run characters from Figure 3 [Colour figure can be viewed at wileyonlinelibrary.com]

2.3 | Clustering by reversible sorting

Common characters tend to generate small MTF numbers and large run lengths, while the opposite holds for uncommon ones. It is possible to exploit this information by clustering the <run character, MTF number, run length> triples according to the run characters. Clustering must be reversible to allow recovering the original order of the triples. The desired result can be obtained by sorting the triples *stably*, with run characters as sorting keys; triples having the same run character keep their original order. Each distinct key value (here within 0..255) defines a cluster of MTF numbers and run lengths. One way to measure the clustering effect is to compare the average run-length of the MTF number zero before and after the sorting. Table 1 shows this for some selected files.

Our sorting algorithm is based on the simple, linear-time counting sort pp75-79 of Reference 26. Algorithm 3 outlines the method, tuned for the current purpose.

The encoder creates two output arrays, one for MTF numbers and another for run lengths. The arrays are implicitly divided into bins (lines 3 to 4 in Algorithm 3). The bins are implicit, and concatenated in alphabetic order, as shown in Figure 5. For normal ASCII characters there would be maximally 256 bins per array. A pointer (”cursor,’ ”finger”) is maintained for each bin, showing the next free position, with each initially pointing to the head of the corresponding bin. The run character, MTF number and run length vectors are processed from left to right. The current MTF number and run length are sorted (on lines 12 to 15) according to the current run character, which is the character that produced them. Finally, the bin pointers are incremented by one (line 16), satisfying the stability requirement.

As can be seen from sorted MTF numbers row of Figure 5, the common character “a” has produced a well-compressible stream of zeros, while the rarer characters “c” and “d” have produced higher MTF numbers. Thus the example, though small, demonstrates the potential of the sorting to significantly increase clustering. In addition to the sorted MTF numbers, the character frequency vector (4, 3, 3, 2) and the final state of the MTF list (b, c, d, a) must be encoded in the header as auxiliary data. These together allow decoding of the original string of run characters.

According to our experimental results (see Section 4), reversible sorting substantially increases the compression rate of the final interpolative coder, thanks to the increase of clusteredness among the MTF number and run length sequences.

TABLE 1 Average run-length of the Move-to-Front number zero before and after the sorting

File	Before	After	Change
bib	1.8052	1.9592	+8.5%
book1	1.8573	2.1510	+15.8%
book2	1.9569	2.2161	+13.2%
E.coli	1.6076	1.6450	+2.3%

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
1: // Compute sort bin indices
2: binPtr[0] ← 0
3: for i from 1 upto 255 do
4:     binPtr[i] ← binPtr[i − 1] + runCharFreq[i-1]
5: end for
6:
7: // Loop over the run length and MTF vectors
8: for i from 0 upto runCount−1 do
9:     c ← runChar[i]                          ▷ Run character (sort key)
10:    m ← mtf[i]                              ▷ MTF number produced by this occurence of c
11:    r ← runLen[i]                           ▷ Run length of this occurrence of c
12:    // Sort the MTF number produced by this occurence of c
13:    sortedMtf[binPtr[c]] ← m
14:    // Sort the run length produced by this occurence of c
15:    sortedRunLen[binPtr[c]] ← r
16:    binPtr[c] ← binPtr[c] + 1
17: end for
18: return (sortedMtf, sortedRunLen)
```

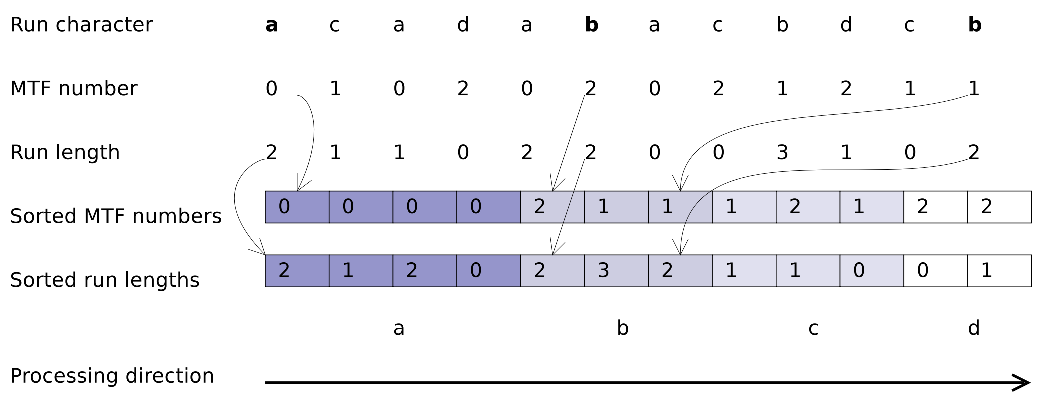

FIGURE 5 Sorting of the Move-to-Front (MTF) numbers from Figure 4 and the run lengths from Figure 3. Sort bins are highlighted with colors. Arrows illustrate the sorting of selected MTF numbers and run lengths [Colour figure can be viewed at wileyonlinelibrary.com]

2.4 MFF transform

The big question is, how will the decoder recover the original, unsorted order of the MTF numbers and run lengths because sorting, in general, is not reversible. The trick is to perform the corresponding MTF list modifications as the MTF encoder, but now working *backwards*. Starting from the final state of the encoder's MTF list, the decoder examines the character at the front of the list, de-sorts the next MTF number and run length from the bin of that character, and then

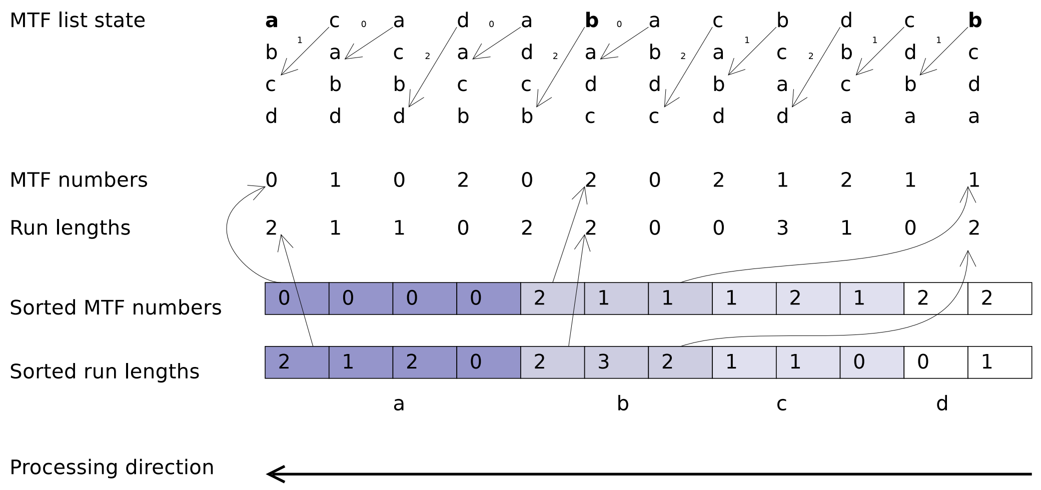

FIGURE 6 Move-From-Front transform. Starting from the final state of the encoder's Move-to-Front (MTF) list (b,c,d,a), the decoder undoes the encoder's list update and sort operations, illustrated here by reversing the arrows from Figure 4 and Figure 5. The original run characters can be retrieved from the top-most row of the reproduced MTF list states [Colour figure can be viewed at wileyonlinelibrary.com]

moves the character downward in the MTF list as many positions as indicated by its MTF number. In essence, the decoder is using the MTF numbers as list update instructions, but the meaning of each MTF number *m* is reversed. The MTF encoder moved the character *to the front* of the list from position *m*, but the decoder moves the character *m* steps *from the front* to return it to the position from whence it came. Accordingly, we call the decoder's operation the MFF transform, in contrast to the *MTF* recoding used by the encoder.

To reverse the sorting, the decoder first uses the run character frequencies to compute the sort bin indices. However, because the decoder operates in reverse direction, the bin pointers are now set to point to the *rear* of the sort bins. The character at front of the final MTF list state of the encoder is the last character encoded—the rear position of the character's sort bin contains the last MTF number. Correspondingly, the last run length is obtained from the rear of the related run length bin. After desorting, the bin pointer is decremented and the MTF list is updated. The list update step maintains the invariant that the previously encoded character is at the front position of the list, and the desorting process continues iteratively. A sample MFF transform is shown in Figure 6.

The crucial feature of the MFF transform is that it gives the decompressor access to each intermediate MTF list state. The original run characters can then be recovered as the sequence of the top-most characters from each intermediate list state, in reverse order. The characters and the associated run lengths can be collected from back to front for the run-length decoding. Finally, the decoded BWT output string is passed to reverse-BW transform to recover the original uncompressed input, as shown in Figure 2.

Because both the encoder and decoder have access to the run character frequencies, there is one optimization that can be made. The last MTF number (the first one encoded) can be decoded implicitly: the decoder can maintain a table of character counts, decrementing them after decoding a character—when the count drops to one, the character at the front of the MTF list is known to be the last occurrence of this character. Now the character can be *removed* from the MTF list, instead of being moved downward according to the MTF number. Therefore the last MTF number is superfluous and need not be encoded. Again, this minor optimization can actually be quite significant, especially for small files. For clarity, this optimization is *not* shown in the examples and pseudocodes.

To allow the decoder to perform the MFF transform, the final state of the MTF list and the run character frequencies must be encoded into the header of the compressed file. Because all of these are nonnegative and integers, and because interpolative coding is parameterless[1], interpolative coding can be used to directly compress the header. The run character frequencies are somewhat clustered because they are stored in alphabetical order, so they compress well. The MTF list is a sequence of byte values 0 to 255 (ASCII characters). They are not especially clustered, but interpolative coding them as well brings a small benefit. The interpolative-coded sorted MTF numbers and run-lengths constitute the main part of the compressed file. The structure of the compressed file and typical output component sizes are described in Section 2.6.

[1]To be precise, interpolative coding does require one parameter: the count of the original integers. However, this information is easily available in most applications, including this one. We include the count as a header to each interpolative-coded compressed block.

Algorithm 4. 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]]                ▷ MTF number produced by c
12:     r ← sortedRunLen[binPtr[c]]             ▷ Run length of c
13:     runChar[i] ← c                          ▷ Run char always from front of MTF list
14:     runLen[i] ← r                           ▷ Run characters and run lengths go together
15:     binPtr[c] ← binPtr[c] − 1               ▷ Bin ptr one step back
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)
```

The pseudo-code for the MFF transform is given in Algorithm 4.

MFF is a combination of desorting and reverse-MTF, because the former cannot be realized without the latter, and vice versa. In the encoding phase above, these stages were kept separate for simplicity, but they could have been combined as well, because their main loops are similar.

2.4.1 | Example

The MFF transform process is shown in Figure 6. The input data for the example are the run lengths from Figure 3 and the MTF numbers and the final MTF list state from Figure 4, namely (b, c, d, a). The four bins, each shown with a different color and with their MTF values and run lengths, are the result of counting sort, and represent the clusters of the four characters of the alphabet. The original sequence can be restored by walking through the bins, and applying MFF operations to the MTF list. For example, the last run of the BWT output was εbbb,ε and the run length is $3 - 1 = 2$. There were two runs between this and the previous b-run, namely c and d. Thus, the MTF number of the last b-run is $2 - 1 = 1$. Character "b" is pushed to index 2 in the MTF list (see the next to last column in Figure 4). This can be continued step by step to the left.

2.4.2 | Discussion

The MFF transform solves the following problem, noted already by p184 of Fenwick[4] and Wirth[8] et al in the MTF recoding process, useful contextual information—knowledge of the characters that produced the MTF numbers—is lost. For example, the MTF number 0 can stand for both ”e’ and ”z” - characters with very different properties. Typically, ”e” is much more likely to produce a 0 than ”z.” The MFF transform allows us to exploit this knowledge both in the encoder and the decoder, and encode the 0s produced by the character e more efficiently.

An interesting viewpoint to the MTF and the MFF transform is to think of the MTF numbers not as recordings of characters but as instructions according to which the encoder and decoder must synchronously modify their MTF lists, in order to recover the original run character string. We could encode the instructions with any suitable method, but it is hard to envisage a better way than by using MTF numbers.

At least two other MTF-like recoding schemes have been proposed that also have the property of retaining character knowledge: Inversion Frequencies pp43-44 of References 5,27 and Distance Coding pp44-45 of References 5,28. However, these are in general more complex than the simple MTF, and slower than our MTF+MFF approach.

A disadvantage of the MFF transform is that, since the encoder and decoder work in opposite directions and must always keep their MTF list states synchronized, it is not possible to exploit knowledge of the surrounding characters when producing the MTF numbers[2]. This means that MTF improvements such as sticky MTF,[29] MTF-1,[30] and MTF-2,[18] are not compatible with the MFF approach. These improvements typically make up a significant share of the compression improvements in other Burrows-Wheeler-based compression methods. However, according to our results, the clustering advantage of the MFF transform seems to outweigh the advantage of these MTF improvements.

2.5 | Interpolative coding

As the last step, we compress each intermediate output (the final MTF list state, the run character frequencies, the sorted MTF numbers and the sorted run lengths) separately using interpolative coding.

Interpolative coding is a nonstatistical method for bounded coding of nonnegative integers. It can be conceptually divided into two stages: (i) computing upper bounds for each integer to be coded and (ii) encoding each integer with either $\lceil \log_2(n) \rceil$ or $\lceil \log_2(n) \rceil - 1$ bits using a semi-fixed length prefix code, where n is the upper bound for that integer.

The encoding stage is straightforward. Bounded, semi-fixed length coding of nonnegative integers is a well-known topic; the idea initially appeared in Golomb coding[31] and was exploited by Moffat in the initial publication of interpolative coding[1][3]. The remaining task is to decide which integers to assign the shorter codewords. We shall return to this question below.

2.5.1 | Computing upper bounds

This critical stage can be efficiently implemented as follows:[15] insert the integers to be coded as leaves of a binary tree. Then for each pair of integers, write their sum into the parent node. The result is a sum-up tree with the total sum of all integers in the root node at index 1. The tree can be stored using a heap-like implicit data structure (see Figure 7): the left child of node i is at index $2i$ and the right child is at index $2i + 1$. Combined with breadth-first encoding and decoding, this makes the data access cache-friendly. The only disadvantage of this approach is that we need to temporarily store $m - 1$ internal nodes, if m is the number of the integers to be coded, but this is usually not a problem in practice. Note that the tree is left-balanced for any number n of source elements, that is, all levels except the lowest are full, and the lowest level is pushed to the left. After filling up the sum-up tree, we store the root node using a universal code, such as the Elias delta code.[33] Next, we encode each left child in the sum-up tree using a semi-fixed length prefix code. There is no need to encode right children, as their value can always be computed as the parent minus the left child. In addition, there is no need to store all-zero subtrees, because, when encountering a zero internal node, both the encoder and decoder will know that the subtree rooted at that node will consist of only zeros. Since the magnitude of the parent node will restrict the number of bits needed in the encoding of the left child, clustered sequences, where large integers occur together, can be compressed effectively.

[2]In other words, the MTF recoding method used together with the MFF transform must be *memoryless*.
[3]For a more thorough treatment of semi-fixed coding (also called phased-in or truncated block coding), see for example the excellent book by pp. 80-85 of Salomon et al[32]

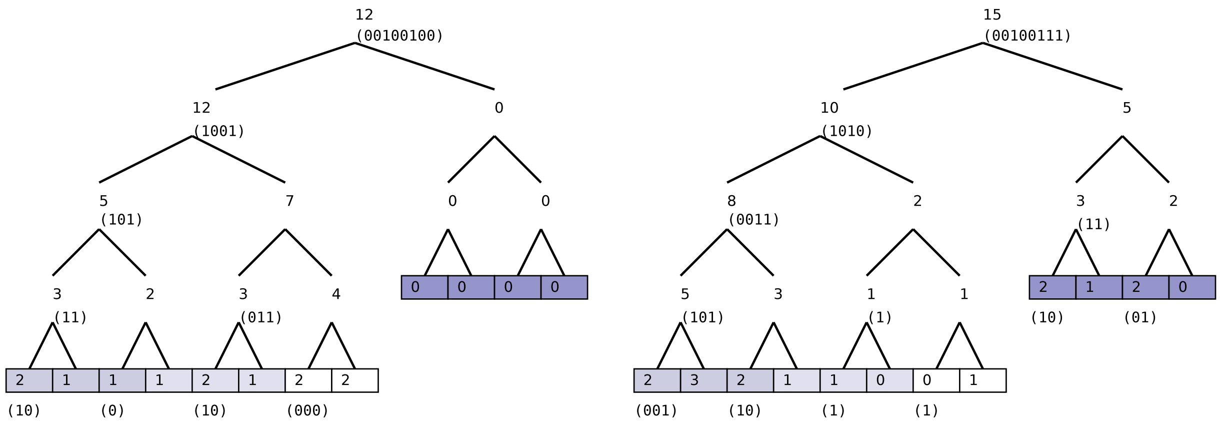

FIGURE 7 Interpolative coding of the sorted Move-to-Front (MTF) numbers (left) and run lengths (right) of Figure 6, with their semi-fixed length codes in parentheses. The final encoding of the MTF numbers is 00100100100110111011100010000 and the final encoding of the run lengths is 00100111101000111110111001001101 1. [Colour figure can be viewed at wileyonlinelibrary.com]

2.5.2 | Semi-fixed length coding

Semi-fixed length coding, called "truncated binary coding" by Golomb,[31] can be applied whenever the upper bound of the integer to be coded is not a power-of-two. Semi-fixed length codes are prefix codes with two codeword lengths: k and $k-1$ bits. In interpolative coding, the choice of the semi-fixed length codebook typically has a substantial effect on the compression rate. In our scheme, we make use of two codebooks: *center-short*, in which the shorter codes are assigned to the middle part of the possible range; and *center-long*, in which the shorter codes are assigned to the low and high end of the range. For example, to code an integer $x <= 5$, one could use the center-short codebook {000,001,10,11,010,011} or the center-long codebook {10,000,001,010,011,11}.

For the sorted run-lengths, we use center-long for the leaves and center-short for the internal nodes of the sum-up tree. The run-lengths belonging to the same sort bin can be modelled as identically distributed random variables. The SD of the sum of two random numbers from the same distribution is smaller than 2* single deviation. Or in other words: summing up values from the same distribution tends to pack the sums relatively closer to the mean of the sum, approaching the normal distribution in the limit. The center-short/center-long division takes advantage of this property.

It would seem plausible that the same argumentation would be true for the sorted MTF numbers. However, using the same codebook split for the MTF numbers results in a worse compression rate than just using *center-short* for all of them.

The choice of the semi-fixed length codebook is significant. For example, using the above approach for the world192.txt test file results in a compression rate of 1.3544, while the inverse codebook order for the run-lengths (ie, using center-short for the leaf level and center-long for the rest), and center-long for all MTF numbers, results in a compression rate of 1.4106. Automatic optimization of the codebook choice is here left for future work.

2.5.3 | Context-based sorting

As the BWT has shown, context-based sorting can be used to increase clusteredness of the input data. The same idea can be used to increase the effectiveness of interpolative coding. This was done in an earlier work,[16] where the authors used context-based sorting combined with interpolative coding to compress bi-level images, paralleling the compression rate of the JBIG standard, but being around 50% faster. In Section 2.3, we proposed using the run character as a sort context for the corresponding MTF number and run-length. We have argued earlier that this sorting significantly increases the clusteredness of the MTF number and run-length sequences [4]. Interpolative coding can efficiently take advantage of this extra clustering.

[4]Usually, the gain of using context-based sorting is so clear that an interpolative-coding based compression method can be thought to consist of three instead of two necessary conceptual stages: context-based sorting, upper bound computation, and semi-fixed length coding

2.5.4 | Example

Figure 7 shows an example of interpolative coding, applied to the result of the stable sorting phase in Figure 6. The tree nodes ($2n - 1$ for n items) are indexed in breadth-first order. The sort bins are serialized in alphabetical order to n leaves of the tree. For example, the four zero leaves are from the sort bin of char "a." Following Teuhola's[15] approach, we then build a heap-like structure, where the parent of each node is computed as the sum of its children. The root node is encoded separately with the Elias delta code.[33] For all left children in the tree, we apply bounded, semi-fixed length coding, using the center-long codebook for leaves and center-short for internal nodes. Figure 7 shows the codewords below each node. The run-lengths of Figure 6are encoded analogously. Pseudo-code for interpolative encoding is presented in Algorithm 5.

In Teuhola 2009,[15] interpolative coding was applied to the plain MTF number sequence. In this paper, we propose using run-length coding and reversible sorting to increase the compressibility of the data. As a result, our compression rate for, for example, the Calgary corpus is now around 7% better than in Teuhola's article.

Algorithm 5. InterpolativeEncode

```
Input: Sequence of $n$ nonnegative integers S[0 … $n-1$]
Output: Binary code sequence B
 1: for $i$ from 0 to $n-1$ do
 2:     T[$n+i$] ← [$S$][$i$]
 3: end for
 4: for i from n − 1 downto 1 do
 5:     T[$i$] ← T[$2*i$] + T[$2*i+1$]
 6: end for
 7: B ← universal-encode(T[1])
 8: for i from 1 to n − 1 do
 9:     bound ← T[$i$]
10:     if bound >0 then
11:         B append semi-fixed-length-encode(T[$2*i$], bound)
12:     end if
13: end for
14: return <B>
```

2.6 | Structure of the output

The compressed byte stream consists of three top-level components: the BWT permutation index, the compressed header and the compressed data. A fixed-length 8-byte unsigned integer is used to store the BWT permutation index. The header section contains the final MTF list state produced by the MTF recoding and the global run character frequencies produced by the RLE stage. The main part of the compressed data consists of the sorted MTF numbers and run-lengths. All components, except the BWT permutation index, are compressed using interpolative coding. Since interpolative coding is non-statistical and nonparametric, the decoder can decompress each header component separately before starting the MFF process to recover the BWT output.

For most files, the size of the compressed header is less than 1% of the total compressed size. Thus, the overhead of our post-BWT stage seems reasonable, and this is demonstrated by the competitive compression rates for the small files in our test corpora, such as xargs.1 in the Canterbury corpus. Table 2 shows the sizes of the output components in selected

TABLE 2 Example output component sizes

Component	bib	geo	*E.coli*
1. BWT permutation index	8 bytes	8 bytes	8 bytes
2. Final MTF list state	93 bytes	283 bytes	18 bytes
3. Run character frequencies	105 bytes	2416 bytes	93 bytes
4. Sorted MTF numbers	18 077 bytes	425 634 bytes	6 845 772 bytes
5. Sorted run lengths	8776 bytes	32 877 bytes	4 180 426 bytes

BWT, Burrows-Wheeler transform; MTF, Move-to-Front.

files. The main factor affecting the size of the header is the size of the active alphabet. As can be seen from Table 2, the header is small for *E. coli* (which has an active alphabet of just four characters), and relatively large for geo (which has an active alphabet of 256 characters).

3 | VARIATIONS ON THE BASIC METHOD

As mentioned in the introduction, several *pre-transformation* steps have been suggested in the literature (eg, Kruse[6] and Awan[7]). These can be used to improve the compression rate of almost any Burrows-Wheeler-based compression method by making assumptions about the input data.

Another way to boost the effectiveness of Burrows-Wheeler compression, one that is often used in the literature, is to tweak the alphabet order used in the BWT itself to sort the input characters. The original paper by Burrows and Wheeler simply used the numerical values of each byte (or ASCII character) as the sort key. The effect of the sort order on compressibility of the BWT output was studied in depth by Chapin.[34] A smallish general improvement on text files can be attained by using the so-called *aeiou* order instead of plain ASCII order. In our implementation, we use the aeiou order by Balkenhol and Shtarkov.[30]

3.1 | Skipping maximal subtrees

The basic version of interpolative coding skips encoding of all-zero subtrees. Analogously, maximal subtrees can be skipped, provided that we know the maximum leaf. For binary files, the maximum MTF number is often close to 255, while for text files it is typically around 70 to 90. However, the DNA file E.coli in the Canterbury-large corpus has an alphabet of size 4, producing MTF numbers 0, 1 and 2 in our method. Here, skipping maximal subtrees, that is, subtrees full of the MTF number 2, gives an improvement of around 2% to the compression rate. For all our other test files, skipping maximum subtrees had no effect.

3.2 | Enlarged sort context

Our method uses the character *c* at the front of the MTF list (ie, the character that produced the next MTF number) as the sort context for the MTF number and the run-length. However, there is more contextual information available to both the encoder and decoder: we also know the character that follows *c* in the original run character sequence (this is the previously encoded or decoded character) and the character that precedes *c* (this is always at the second position of the MTF list, since each run character has distinct neighbors). This additional context information could be used to further improve the clustering effect of our sorting. The main correlation that could be exploited here are repeating patterns of alternating characters, such as “ababa,” that often occur in the run character sequence. These produce zeros in the MTF recoding. We could, for each character *c*, keep track of the character that most frequently alternates with *c*. Whether or not this character is currently both the predecessor and the successor of *c* could be used as a binary pseudo-context, according to which we sort the MTF number either to the head or tail section of *c*’s sort bin, hopefully clustering more of the zeros produced by *c* to the same side. This kind of pseudo-context technique has been efficiently used by Niemi et al.[16] Taking advantage of this idea seems promising, but, in the current work, we were only able to achieve a marginal improvement to the compression. Due to the fact that it requires statistics to be maintained, we left this improvement out from our final method.

4 | EXPERIMENTAL RESULTS

We evaluated our method on three standard data compression test datasets: the Calgary,[35] Canterbury,[36] and the Canterbury-large[37] corpora. As a reference method, we used the popular bzip2 compression tool (version 1.0.6) with the -9 command-line option for maximum compression.

The compression results are shown in Tables 3, 4, and 5. In the tables, *Sort+Inter* is the basic method described in this paper, consisting of the BWT, MTF, MFF, RLE, and interpolative coding. For comparison, we included compression

TABLE 3 Compression results for the Calgary corpus. Results are in bits per byte (bpc)

File	Entropy	Inter	Sort+Inter	Sort+Arith	bzip2
bib	1.8737	2.0191	**1.9451**	2.0193	1.9749
book1	2.4114	2.4922	**2.3377**	2.3706	2.4204
book2	2.0391	2.1119	**1.9910**	2.0551	2.0619
geo	3.8572	4.6163	4.5231	4.6562	**4.4469**
news	2.1461	2.5807	**2.4442**	2.5229	2.5159
obj1	2.7335	4.1325	**3.8824**	4.2321	4.0130
obj2	2.2296	2.5656	**2.4172**	2.5523	2.4776
paper1	2.3049	2.5324	**2.4342**	2.5492	2.4917
paper2	2.3118	2.4998	**2.3934**	2.4709	2.4371
pic	0.7146	0.8508	**0.7256**	0.7633	0.7756
progc	2.2646	2.5756	**2.4736**	2.5997	2.5334
progl	1.5924	1.7642	**1.6927**	1.7718	1.7395
progp	1.5410	1.7427	**1.6920**	1.7873	1.7351
trans	1.3661	1.5152	**1.4638**	1.5482	1.5282
average	2.1215	2.4285	**2.3154**	2.4213	2.3679

TABLE 4 Compression results for the Canterbury corpus. Results are in bits per byte (bpc)

File	Entropy	Inter	Sort+Inter	Sort+Arith	bzip2
alice29.txt	2.2091	2.3259	**2.2033**	2.2732	2.2724
asyoulik.txt	2.4753	2.6038	**2.4901**	2.5474	2.5287
cp.html	2.1412	2.5299	**2.4322**	2.5818	2.4790
fields.c	1.6353	2.1953	**2.1381**	2.3419	2.1804
grammar.lsp	1.7559	2.7165	**2.5885**	3.1303	2.7583
kennedy.xls	1.1099	1.5478	1.5049	1.1452	**1.0121**
lcet10.txt	1.9934	2.0632	**1.9526**	2.0070	2.0190
plrabn12.txt	2.4255	2.4915	**2.3686**	2.4050	2.4169
ptt5	0.7147	0.8508	**0.7256**	0.7633	0.7756
sum	2.0087	2.7571	**2.5841**	2.7967	2.7006
text.html	2.2506	3.1743	**3.0699**	3.4730	3.1793
xargs.1	2.3533	3.3009	**3.1644**	3.6489	3.3347
average	1.9227	2.3797	**2.2682**	2.4261	2.3047

TABLE 5 Compression results for the Canterbury-large corpus. Results are in bits per byte (bpc)

File	Entropy	Inter	Sort+Inter	Sort+Arith	bzip2
bible.txt	1.6030	1.6246	**1.5285**	1.6224	1.6714
E.coli	1.9836	2.4059	2.3212	**1.9643**	2.1575
world192.txt	1.3837	1.4309	**1.3544**	1.4203	1.5835
average	1.6567	1.8204	1.7348	**1.6690**	1.8041

results for straight interpolative coding of the run lengths and MTF numbers without sorting (*Inter*). To measure the efficiency of our final entropy coders, we computed the order-1 entropy of the MFF transform result. The entropy value is the sum of the entropies of each sort bin.

4.1 | Alternative: Arithmetic entropy coder

Most Burrows-Wheeler-based compression methods use statistical modeling, combined with arithmetic coding as their final stage. To compare the performance of interpolative coding against arithmetic coding in our posttransformation method, we implemented the *Sort+Arith* method, where interpolative coding has been replaced with an order-1 arithmetic coder. In this method, arithmetic coding is applied to the sorted MTF numbers and run-lengths, using the sort character as an order-1 context. The model used is based on the structured model of Figure 5 in Fenwick's article,[12] with seven levels. The first level encodes symbols 0 and 1. When encountering a larger symbol, the first-level coder emits one of six escape codes, the id of the lower-level coder responsible for encoding that symbol. Symbols are preinstalled (ie, given an initial probability $1/n$, where n is the total count of the symbols in the context) in every coder, except in the last one, which encodes symbols 64 to 255 and uses dynamic symbol installation, that is, a symbol is installed only after encountering it the first time. Symbol probabilities are updated dynamically as the encoding proceeds. As can be seen, the results for the arithmetic final coder are substantially worse compared to the interpolative coder.

In our experiments, we noticed that MTF recoding of run characters produces a distribution closely following to Zipf's law,[24] so that the frequency of the MTF number having rank i is close to n/i, where n is the highest frequency (rank 1). This differs from Fenwick's observation[25] claiming that the MTF output rather follows the function n/i^2. We conjecture that applying run-length encoding before MTF recoding recovers the basic Zipf property among the run characters. On the other hand, for run-lengths a better approximation is generally n/i^2. For both cases, the fit with Zipf is quite good for the most frequent items, while getting progressively worse for bigger ranks. We tried to use this prior information about the input distributions to bootstrap the adaptive arithmetic coding model, but without substantial improvement to the results.

4.2 | Compression results

On the Calgary corpus, our interpolative coding based method has the best compression result on all files except geo. We note that we did not apply the common trick of reversing this file. The interpolative coding based method is 2.2% better than bzip2 on average. Using an arithmetic instead of interpolative coding produces much worse results. Obviously, the structured arithmetic coding model is not well-tuned to handle the sorted data and the order-1 context.

For the Canterbury corpus, interpolative coding again has the best result for all files except one: kennedy.xls (1.5049 bpc), where it is soundly beaten by bzip2 (1.0121 bpc) and *MFF+Arith* (1.1452 bpc). The kennedy.xls file has the strange property that performing the BW-transform on it twice effectively halves the compressed size. However, we were unable to find an effective heuristic, that could be used to decide whether a double-BWT is needed for a file or not. Thus, we list the result for the single-BWT case only. On the average, *Sort+Inter* is around 1.6% better than bzip2 with the Canterbury corpus.

For the Canterbury-large corpus, *MFF+Inter* again has the best results on most files, being on average 3.9% better than bzip2. However, *Sort+Inter* has exceptionally poor performance on one specific file (*E. coli*), bringing down the average result. We note that for *E. coli*, MFF combined with arithmetic instead of interpolative coding gives a result of 1.9643 bpc. This is close to the best Burrows-Wheeler-based compression results for this file. For example, in the benchmarks of Abel,[18] the best result for *E. coli* is 1.954, achieved by the relatively slow WFC06 method.

4.3 | Speed results

The computational complexity of our post-BWT method is relatively low. Especially the interpolative final coder, being nonstatistical and cache-friendly, is often faster than the arithmetic coders employed by most post-BWT methods in the literature.

We compared the encoding speed of our post-BWT stage against that of the highly optimized bzip2 tool, which uses a Huffman final coder. Focusing on the comparative speed of the post-BWT stage algorithms, we left the BWT itself and all

TABLE 6 Average duration of the post-Burrows-Wheeler transform encoding stage for the test corpora. Results are in average seconds per file

Corpus	Inter	Sort+Inter	bzip2
Calgary	**0.0455**	0.0574	0.0555
Canterbury	**0.0369**	0.0442	0.0439
Canterbury-large	**0.1598**	0.1975	0.1772

memory allocations out of the measurements. Each file was compressed one hundred times in a row; the presented result is the sum of the average running times for each file in the corpus. The benchmarking was performed on an eight-core Intel Core i7-4800MQ CPU running at 2.90GHz. The performance of our method and bzip2's is highly dependent on the CPU cache size. Our test machine had L1 data and instruction caches of 32 KB each, an L2 cache of 256 KB and an L3 cache of 8 MB. Table 6 shows that the *Sort+Inter* method offers competitive speed compared to bzip2, being almost on-par with bzip2 for the smaller Calgary and Canterbury corpora, and requiring around 11% more time than bzip2 for the Canterbury-large corpus. The stripped-down version without sorting (*Inter*) is faster, but has a worse compression rate. In our methods, decoding takes approximately the same time as encoding. One should be careful from drawing too far-reaching consequences from these results, since the benchmarks are to a substantial degree affected by the amount of optimization (especially cache optimization) performed by the programmer. Our implementation is a straightforward one and we did not spend any extra effort on code optimization. We believe that our method could be made significantly faster by employing cache-optimization techniques and by parallel processing.

5 | CONCLUSIONS AND FURTHER WORK

In this paper, we have described a BWT-based compression method that utilizes an interpolative final coder. We introduced a clustering scheme which reversibly sorts the MTF numbers and run-lengths based on the character that produced them. For declustering we presented a novel transform called MFF. We have shown that the applied sorting significantly increases the clusteredness of the MTF number and run-length sequences. Combining the sorting with a final interpolative coder allows us to build a completely non-statistical lossless compression method with competitive speed and compression rate.

Further work on this topic is needed, for example, to find an effective way of using the extended sort context described in Section 3, and how to effectively combine the MFF transform with an arithmetic final coder. While our final compression results are quite close to the entropies of the MFF transform results, there is clearly still room for improvement in the final coding methods, especially for the arithmetic coder. As for interpolative coding, choosing the best codebook in each case is an interesting research topic.

ACKNOWLEDGEMENTS

The authors are grateful to the anonymous reviewers for their valuable comments and suggestions. The first author would like to thank his employers Verimatrix and Huawei Technologies for their flexibility.

ORCID

Arto Niemi https://orcid.org/0000-0003-3118-4511

REFERENCES

1. Burrows M, Wheeler DJ. *A Block-Sorting Lossless Data Compression Algorithm*. Palo Alto, CA: 124: Digital Systems Research Center; 1994.
2. Seward J.. bzip2 https://web.archive.org/web/20180801004107/http://www.bzip.org/.
3. Deorowicz S. Context exhumation after the Burrows-Wheeler transform. *Inf Process Lett*. 2005;95:313-320.
4. Fenwick P. Burrows-Wheeler compression. In: Sayood K, ed. *Lossless Compression Handbook*. San Diego, CA: Academic Press; 2003.
5. Adjeroh D, Bell T, Mukherjee A. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching*. New York, NY: Springer; 2008.
6. Kruse H, Mukherjee A. *Preprocessing Text to Improve Compression Ratios*. Los Alamitos, California: IEEE Computer Society Press; 1997:556.
7. Awan FS, Mukherjee A. *LIPT: A Lossless Text Transformation to Improve Compression*. London, UK: IEEE; 2001.
8. Wirth AI, Moffat A. *Can we do without Ranks in Burrows Wheeler Transform Compression?* . Los Alamitos, CA: IEEE Computer Society Press; 2001:419-428.

9. Ferragina P, Giancarlo R, Manzini G, Sciortino M. Boosting textual compression in optimal linear time. *J ACM*. 2005;52:688-713.
10. Ferragina P, Giancarlo R, Manzini G. *The Engineering of a Compression Boosting Library: Theory vs Practice in BWT Compression*. Berlin, Germany: Springer; 2006:756-767.
11. Balkenhol B, Kurtz S. Universal data compression based on the burrows and wheeler transformation: theory and practice. *IEEE Trans Inf Theory*. 2000;49:1043–1053.
12. Fenwick PM. Burrows-Wheeler compression: principles and reflections. *Theoret Comput Sci*. 2007;387:200-219.
13. Moffat A, Stuiver L. Binary interpolative coding for effective index compression. *Inf Retriev*. 2000;3:25-47.
14. Petri M, Moffat A. Compact inverted index storage using general-purpose compression libraries. *Softw Pract Exp*. 2018;48:974-982.
15. Teuhola J. Tournament coding of integer sequences. *Comput J*. 2009;52:368-377.
16. Niemi A, Teuhola J. *Interpolative Coding as Alternative to Arithmetic Coding in Bi-Level Image Compression*. Hamburg, Germany: IEEE; 2015.
17. Žalik B., Mongus D., Lukač N., Žalik K. R.. Efficient chain code compression with interpolative coding. *Inf Sci*. 2018;439-440:39:49.
18. Abel J. Post BWT stages of the Burrows-Wheeler compression algorithm. *Softw Pract Exp*. 2010;40:751-777.
19. Kärkkäinen J, Sanders P. *Simple Linear Work Suffix Array Construction*. Berlin, Germany: Springer; 2003:943-955.
20. Abel J. *A Fast and Efficient Post-BWT Stage for the Burrows-Wheeler Compression Algorithm*. Snowbird, Utah: IEEE; 2005:449.
21. Fenwick P. Universal Codes. In: Sayood K, ed. *Lossless Compression Handbook*. San Diego, CA: Academic Press; 2003.
22. Abel J. Incremental frequency count - a post BWT-stage for the Burrows-Wheeler compression algorithm. *Softw Pract Exp*. 2006;37:247-265.
23. Bentley JL, Sleator DD, Tarjan RE. A locally adaptive data compression scheme. *Commun ACM*. 1986;29:320-330.
24. Zipf GK. *Human Behavior and the Principle of Least Effort*. Reading, MA: Addison-Wesley; 1949.
25. Fenwick PM. The Burrows-Wheeler transform for block sorting text compression: principles and improvements. *Comput J*. 1996;39:731-740.
26. Knuth DE. *The Art of Computer Programming*. *Sorting and Searching*. Vol 3. 2nd ed. Reading, MA: Addison-Wesley; 1998.
27. Arnavut Z. Move-to-front and inversion coding. *Proc Data Compress Conf*. 2000;33:193-202.
28. Deorowicz S. Second step algorithms in the Burrows-Wheeler compression algorithm. *Softw Pract Exp*. 2002;32:99-111.
29. Fenwick PM. Burrows-wheeler compression with variable length integer codes. *Softw Pract Exp*. 2002;32:1307-1316.
30. Balkenhol B, Shtarkov Y. *One Attempt of a Compression Algorithm using the BWT. SFB343: Discrete Structures in Mathematics*. Bielefeld, Germany: University of Bielefeld, Faculty of Mathematics; 1999.
31. Golomb SW. Run-length encodings. *IEEE Trans Inf Theory*. 1966;12:399-401.
32. Salomon D, Motta G. *Handbook of Data Compression*. London, UK: Springer-Verlag; 2010.
33. Elias P. Universal codeword sets and representations of the integers. *IEEE Trans Inf Theory*. 1975;21:194-203.
34. Chapin B., Tate SR. *Higher Compression from the Burrows Wheeler Transform by Modified Sorting*. Denton, Tx: University of North Texas 1998.
35. The Calgary Corpus http://corpus.canterbury.ac.nz/descriptions/#calgary.
36. The Canterbury Corpus http://corpus.canterbury.ac.nz/descriptions/#cantrbry.
37. The Large Corpus http://corpus.canterbury.ac.nz/descriptions/#large.

How to cite this article: Niemi A, Teuhola J. Burrows-Wheeler post-transformation with effective clustering and interpolative coding. *Softw Pract Exper*. 2020;50:1858–1874. https://doi.org/10.1002/spe.2873