

Fast Compression with a Static Model in High-Order Entropy

Luca Foschini* Roberto Grossi[†] Ankur Gupta[‡]
Jeffrey Scott Vitter[§]

Abstract

We report on a simple encoding format called `wzip` for decompressing block-sorting transforms, such as the Burrows-Wheeler Transform (BWT). Our compressor uses the simple notions of gamma encoding and RLE, organized with a wavelet tree, to achieve a slightly better compression ratio than `bzip2` in less time. In fact, our compression/decompression time is dependent on H_h , the h th order empirical entropy. This relationship of performance to the compressibility of data is a key new idea among compression algorithms. Another key contribution of our compressor is its simplicity. Our compressor can also operate as a full-text index with a small amount of data, while still preserving backward compatibility with just the compressor.

1 Introduction

Most text compression algorithms currently in use adapt their performance according to the statistics of the file seen so far. These *adaptive* methods have many advantages, mainly revolving around better compression ratios, as the algorithm infers the statistical model underlying the data. This process can be expensive computationally, and in truth, even a deterrent to use if the opportunity cost is high enough. On the other hand, purely static compression models perform extremely well with regard to compression/decompression time. What we offer in this paper is a static model that compresses to within 5% of results achieved by adaptive methods, using extremely simple techniques, with very fast encoding and decoding.

*Scuola Superiore Sant’Anna, Piazza Martiri della Libertà 33, 56127 Pisa, Italy (foschini@sssup.it). Support was provided in part by Scuola Superiore Sant’Anna.

[†]Dipartimento di Informatica, Università di Pisa, via Filippo Buonarroti 2, 56127 Pisa, Italy (grossi@di.unipi.it). Support was provided in part by the Italian MIUR project “ALINWEB: Algorithmics for Internet and the Web” and by the French EPST program “Algorithms for Modeling and Inference Problems in Molecular Biology”.

[‡]Center for Geometric and Biological Computing, Department of Computer Science, Duke University, Durham, NC 27708–0129 (agupta@cs.duke.edu). Work done as a visiting scholar at Purdue University. Support was provided in part by the National Science Foundation through grant CCR–9877133 and by the Army Research Office through grant DAAD19–03–1–0321.

[§]Department of Computer Sciences, Purdue University, West Lafayette, IN 47907–2066 (jsv@purdue.edu). Support was provided in part by the Army Research Office through grants DAAD19–01–1–0725 and DAAD19–03–1–0321 and by an IBM Research Award.

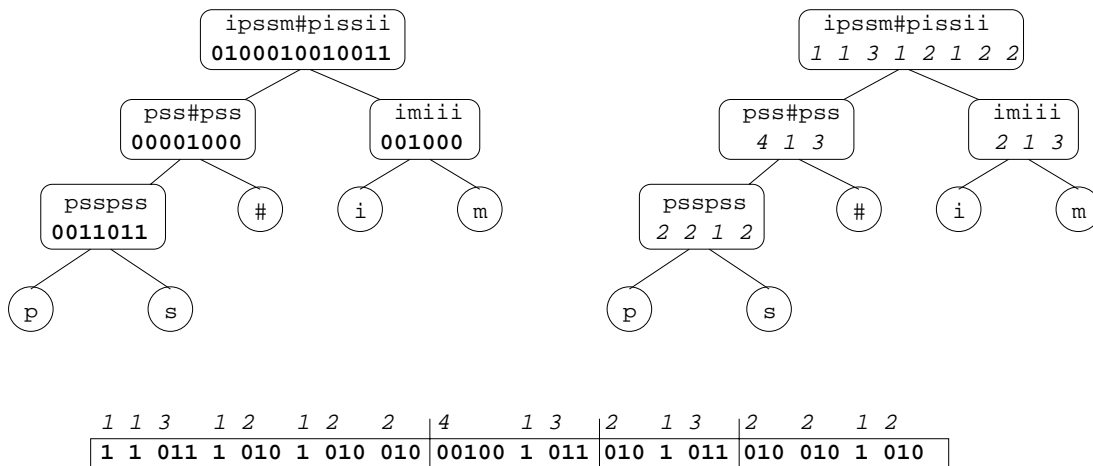


Figure 1: Left: an example wavelet tree. Right: an RLE encoding of the wavelet tree.

Two basic methods we use are RLE and gamma encoding. Run-length encoding (RLE) simply represents each subsequence of identical symbols (a run) from an input sequence as the pair (l, s) , where l is the number of times that symbol s is repeated. For a binary string, there is no need to encode s , since its value will alternate between 0 and 1 . The length l is then encoded in some fashion. One such method is the γ code, which represents the length l in two parts: the first encodes $1 + \lfloor \log \ell \rfloor$ in unary, followed by the value of $\ell - 2^{\lfloor \log \ell \rfloor}$ encoded in binary, for a total of roughly $2\lfloor \log \ell \rfloor$ bits. γ codes are optimal for a statistical model in which the probability of a gap g is proportional to $1/g^2$.

2 The Wavelet Tree and the Static Model

2.1 The Wavelet Tree

Grossi, Gupta, and Vitter [6] introduce the *wavelet tree* for reducing the redundancy inherent in retaining separate dictionaries for each symbol appearing in the text. In order to remove redundancy among dictionaries, each successive dictionary only encodes those positions not already accounted for previously in other dictionaries. Encoding the dictionaries in this way achieves the high-order entropy of the text, as per the discussion in Lemma 4.1 of [6]. Consider the example wavelet tree in Figure 1, built on the text `mississippi#`, where `#` is an end-of-text symbol.

We implicitly consider each left branch to be associated with a 0 and each right branch to be associated with a 1 . Each internal node u is a dictionary $\text{dict}[u]$ with the elements in its left subtree stored as 0 , and the elements in its right subtree stored as 1 . For instance, consider the leftmost internal node, whose leaves are `p` and `s`. The dictionary (leaving aside the leading 0) indicates that a single `p` appears in the BWT string, followed by two `s`'s, and so on. The second tree indicates an RLE encoding of the dictionaries, and the bottom bitvector indicates its actual storage on disk in heap layout with a γ encoding of the run lengths previously described. The leading 0 in each node of the wavelet creates a unique association between the sequence of

file	γ	δ	γ +escape	arithm.	huffman	$a = 0.88$	adaptive a
E.coli	2.1780	2.5238	2.4763	2.7797	1.9932	2.1017	2.0758
asyoulik.txt	2.6304	2.9104	2.9129	2.7324	2.5946	2.5875	2.5873
bible.txt	1.6109	1.7677	1.7839	1.8190	1.5963	1.5901	1.5903
cp.html	2.6949	2.9554	2.9310	2.7170	2.6487	2.6465	2.6543
fields.c	2.4387	2.6145	2.5894	2.4645	2.3228	2.4186	2.4186
kennedy.xls	1.4269	1.6051	1.4718	1.6834	1.3521	1.3998	1.3968
random.txt	6.7949	7.9430	7.7460	6.1273	6.0004	6.5210	6.4187
sum	2.9500	3.2324	3.1803	2.9184	2.8765	2.8792	2.8698
world192.txt	1.4699	1.5890	1.6095	1.5815	1.4555	1.4540	1.4550
xargs.1	3.3820	3.7303	3.6564	3.3763	3.3068	3.3404	3.3404

Table 1: Several codes with RLE (in bits per symbol)

RLE values and the bitvector.

It has been shown both theoretically [6] and in practice [7] that the space occupancy of the wavelet tree does not change regardless of the shape of the tree. As such, we opt for a balanced tree whose nodes are stored according to a heap layout. Thus, the root occupies position 1, and the node in position i has its parent in position $\lfloor i/2 \rfloor$ (if $i > 1$) and its children (if any) in positions $2i$ and $2i + 1$, respectively.

2.2 Empirical Distribution of RLE Values and γ Codes

A natural question arises as to the choice of the simplistic γ encoding, since, theoretically speaking, a number of other prefix codes (δ , ζ , and skewed Golomb, for instance) outperform γ codes. However, as discovered in [7], γ encoding seems extremely robust. Our recent experiments are summarized in Table 1, where we report the bits per symbol (*bps*) achieved in our experiments. Each row of Table 1 reports data for a file from Calgary or Canterbury corpora. The compression methods are applied to the sequence of RLE values obtained in the example of Figure 1, so the alphabet for each file is given by all distinct RLE values rather than the original set of symbols in the input string. The second column reports the bps required for the γ code; the third to the δ code; the fourth to the γ code in which any run length other than 1 is encoded using γ whereas a sequence of s 1s is encoded with the γ code for 1 followed by the γ code for s ; the fifth to Moffat’s arithmetic coder mentioned in Section 2.4; the sixth column refers to the Huffman code in which the cost of encoding the (large!) prefix tree is not counted (which explains its bps being smaller than that of the arithmetic code). The last two columns refer to the rangecoder mentioned in Section 2.4, where we employ either a fixed slack parameter $a = 0.88$, or we choose the best value of a adaptively. There is clear empirical evidence that γ encoding is almost the best. In Section 2.3, we will formalize this experimental finding more clearly by curve-fitting the distribution implied by γ onto the distribution of the run lengths.

Improving upon γ to encode these RLE values requires a significant amount of work with more complicated methods. As a matter of fact, this news is both encouraging and discouraging. It seems that there is no real hope to improve upon γ using prefix codes, since Huffman codes are optimal prefix codes. Further improvement then, in some sense, necessitates more complicated techniques (such as arithmetic coding) which have their own host of difficulties, most often a greatly increased en-

coding/decoding time.

2.3 Statistical Evidence Justifying the Static Model of γ Codes

In this section, we motivate our choice of γ encoding more formally, with statistical evidence suggesting that the underlying distribution of RLE values matches that which is optimally encoded by γ encoding. For instance, consider the empirical cumulative distribution of the RLE values for `bible.txt`, shown in Figure 2. Note that this distribution is fitted by the function

$$cdf(x) = e^{-a/x} \quad x \in \mathbf{N}^+, \quad (1)$$

where parameter $a \in \mathbf{R}^+$ is a constant depending on the data file (`bible.txt` in our case). For the Canterbury Corpus, we observed that $a \in [0.5, 1.8]$, depending on the file (e.g., $a = 0.9035$ for `bible.txt`). We can compute the derivative of cdf as if it were a continuous function and we obtain the probability density function

$$pdf(x) = \left(\frac{ae^{-a/x}}{x^2} \right) / \left(\sum_{i=1}^{\infty} \frac{ae^{-a/i}}{i^2} \right), \quad i, x \in \mathbf{N}^+, a \in \mathbf{R}^+ \quad (2)$$

where the term $\sum_{i=1}^{\infty} \frac{ae^{-a/i}}{i^2}$ is the normalization factor. As one can see from Figure 2, function (2) fits the empirical probability density of the RLE values computed for `bible.txt` extremely well, suggesting that approximating the cdf by a continuous function incurs negligible error.¹ Since $pdf(x) \sim \frac{1}{x^2}$ as x approaches infinity, we have

$$\lim_{x \rightarrow \infty} e^{-a/x} = 1 \Rightarrow \lim_{x \rightarrow \infty} \left(\frac{ae^{-a/x}}{x^2} \right) / \left(\sum_{i=1}^{\infty} \frac{ae^{-a/i}}{i^2} \right) \propto \frac{1}{x^2}$$

Since the γ code is optimal for distributions proportional to $1/x^2$, we finally have some reasonable motivation for the success of the γ code on an RLE stream of wavelet tree data. However, these results only indicate the measure of success on prefix codes; encodings which can assign fractional bits may yet yield significant improvement.

2.4 Arithmetic Coding of RLE Values

We performed various tests with Moffat's implementation of arithmetic coding,² but the results were not satisfying compared with γ . To resolve this problem, we employ the statistical model of function cdf to tailor an arithmetic coder to perform well on RLE values. Recall that both pdf and cdf depend on the knowledge of the parameter a in formula (1), which in turn depends on the file being encoded. (We ran experiments with a fixed $a = 0.88$, which also yielded good results on most files.) To this end, we take a free and fast arithmetic-like coder called range coder [9], employed in `gzip`. We encode the RLE value r by assigning it an interval of length $cdf(r+1) - cdf(r) =$

¹We employed the MATLAB function called `LSQCurvefit`, which finds the best fitting in terms of the least square error between the function and the raw data to be approximated.

²The code (written in Java at <http://mg4j.dsi.unimi.it>) is inspired by the arithmetic coder of J. Carpinelli, R. M. Neal, W. Salamonsen and L. Stuiver, which is in turn based on [5].

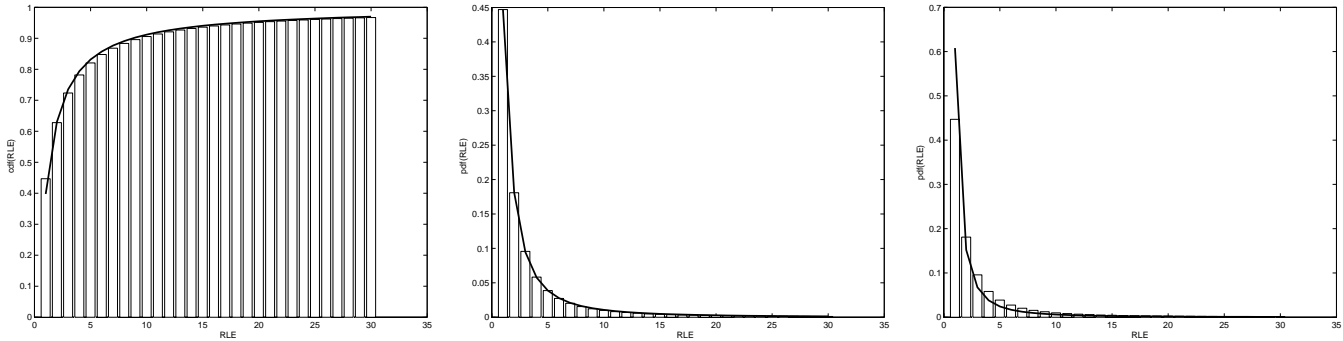


Figure 2: The x axis shows the distinct RLE values for `asdf bible.txt`, in increasing order. Left: The empirical cumulative distribution together with our fitting function cdf from (1). Center: The empirical probability density function together with our fitting function pdf from (2). Right: The empirical probability density function together with the fitting function $\frac{6}{\pi^2 \cdot x^2}$, where $\frac{6}{\pi^2} = \sum_{i=1}^{\infty} \frac{1}{i^2}$ is the normalizing factor.

$pdf(r)$. (This encoding appears to be faster than using the cumulative counts of the frequency of values already scanned, like other well-known arithmetic coders.) With this kind of compressor, we improve the compression rate from 1% to 5% with respect to γ encoding (see Table 1). We then transform our arithmetic compressor so that the parameter a could be changed adaptively during execution, hoping for a better compression ratio. We need a cue to infer a from the values already read, so we use a maximum likelihood estimation (MLE) algorithm (described below).

The main hurdle to simply using MLE is its assumption of independent trials. Though we could not find a satisfiable measure for this notion, we measure the auto-covariance (normalized) of the RLE values. This method is widely adopted in signal theory [11] as a good indicator of independence of a sequence of values, though it does not necessarily imply independence. In our case, the correlation between consecutive RLE values is very low for the files in Canterbury corpus, which again, though it does not imply independence in the strict sense, is a strong indication nonetheless. With this observation in mind, we assume statistical independence of the RLE values in order to define the likelihood function

$$l_x(a, x_1, \dots, x_k) = \prod_{i=1}^k pdf(x_i) = \left(\prod_{i=1}^k \frac{ae^{-a/x_i}}{x_i^2} \right) \left(\sum_{i=1}^{\infty} \frac{ae^{-a/i}}{i^2} \right)^{-k}.$$

We want to find the value a where l_x reaches its maximum. Equivalently, we can find the maximum of its log (a log-likelihood function) $L_x(a, x_1, \dots, x_k) = \log l_x(a, x_1, \dots, x_k)$. Since L_x is differentiable, we take its derivative with respect to a , solving for maxima by equating to 0. This yields

$$-\frac{\partial}{\partial a} \log \left(\sum_{i=1}^{\infty} \frac{e^{-a/i}}{i^2} \right) = \frac{1}{k} \sum_{i=1}^k \frac{1}{x_i} = H(x)^{-1},$$

where $H(x)$ is the Harmonic mean of the sequence x . By denoting the left hand term by $f(a)$, we have that $a = f^{-1}(H(x)^{-1})$.

Unfortunately, $f(\cdot)$ is not an analytical function and is very difficult to compute, even for fixed a . For instance, for $a = 0$ we have $f(a) = \frac{\zeta(3)}{\zeta(2)} = 0.7307629$, where $\zeta(\cdot)$ is the Riemann Z function. We therefore apply numerical methods to approximate the function for $a \in [0.5, 1.8]$ (which is the range of interest for us) with the second-degree polynomial: $a = 6.96 - 16.4912 \cdot H(x)^{-1} + 10.6186 \cdot H(x)^{-2}$.

We update the value of a using the above formula, restarting from 0 to compute the inverse of the Harmonic mean. Surprisingly, all this work leads to a small improvement with respect to the non-adaptive version (in which $a = 0.88$). Looking again at Table 1, the improvement is negligible, ranging from 1% to 2% at best. The best case is the file `random.txt` (in the Calgary corpus), for which the hypothesis of independence of RLE values holds with high probability by its very construction.

3 Wzip: A Simple Tool for Fast Compression/Decompression

3.1 The wzip Encoding Format

The lesson learned in Section 2 suggests that the wavelet tree, coupled with RLE and γ encoding, is a simple but effective mean for compressing the output of block-sorting transforms. In this section, we propose our compression format, `wzip`. The header contains three basic pieces of information: the text length n , the block size b , and the alphabet size Σ . The body of the encoding is then $\lceil n/b \rceil$ blocks, each block encoding b contiguous text symbols (except possibly the last block). With reference to Figure 1, recall the nodes of the wavelet tree are stored in heap ordering. We break this stream into blocks and encode it. The format for a given block is given below:

- A (possibly compressed) bitvector of $|\Sigma|$ bits that stores the symbols actually occurring in the block. Let $\alpha \leq |\Sigma|$ be the number of symbols present. (For large Σ , we may store the bitvector in the header, with smaller bitvectors in the blocks that refer only to the symbols stored in the bitvector in the header).
- The dictionaries encoded with RLE+ γ , taking them in heap numbering and concatenating their encoding. Note that the wavelet tree has α implicit leaves and $\alpha - 1$ internal nodes with dictionaries (see Figure 1).

Note that we do not need to store the length of each encoding, as it is already implicitly encoded as follows. When processing, the end of the encoding for the dictionary in the root of the wavelet tree ends when the sum of the encoded RLEs equals n (or $n \bmod b$ for the last block if not length b). At this point, we also know the total number of `0`s and `1`s, plus the (dummy) leading `0`. The former must be the sum of the RLE values in the next dictionary (in the left child), and the latter the sum of the one after that (in the right child). We can go on recursively this way, down to the implicit leaves, from which we can even infer the frequency of the occurrences of each symbol in the block.

3.2 Compression with `bwt2wzip`

In this section, we describe our compression method `bwt2wzip`, which takes as input the Burrows-Wheeler transformation (hereafter the `bwt` stream) of the file and compresses it efficiently using our wavelet tree techniques. Our approach introduces a

novel method of creating the wavelet tree in just $O(n + \min(n, nH_h) \times \log |\Sigma|)$ time, which is also faster in practice, as the entropy factor can significantly lower the time required. This behavior relates the speed of compression to the compressibility of the input. Thus, we introduce a new consideration into the notion of compressibility—highly uniform data should be easier to handle, both in terms of space and time.

Let c be the current symbol in the `bwt` stream, and let u be its corresponding leaf in the wavelet tree. While traversing the upward path in the wavelet tree to the root, we have to decide whether the run of bits in the current node should be extended or switched (from `0` to `1` or vice versa). However, we do not perform this task for each symbol. Instead, we exploit the runs of equal symbols c , say r_c in number, in the input to avoid multiple passes. We then extend the runs by r_c units at a time. Given any internal node in the tree, the set of values stored there are produced in increasing order, without explicitly creating the corresponding bitvector.

To make things more concrete, we use the following auxiliary information to compress the input string `bwt`. Notice that the leaves of the wavelet tree do not need to be explicitly represented; given a symbol $c \in \Sigma$, it suffices to know its leaf number `leaf[c]`. We also allocate enough space for the dictionaries `dict[u]` of the internal nodes u . We keep a flag `bit[u]`, which is `1` if and only if we are encoding a run of `1`s.

Below, we describe the main loop of the compression. We do not specify the task of encoding the RLE values with γ codes, as it is a standard computation performed on the dictionaries `dict[u]` of the internal nodes u .

```

1 while ( bwt != end ) {
2   for ( c = *bwt, r_c = 1; bwt != end && c == *(++bwt); r_c++ ) ;
3   u = leaf[c];
4   while ( u > 1 ) {
5     if ( (u & 0x1) != bit[u >>= 1] ) {
6       bit[u] = 1 - bit[u]; *(++dict[u]) = 0; }
7     *(dict[u]) += r_c;
8   }
9 }

```

We scan the input symbol c from the current position in `bwt` to determine r_c , the length of the run of c (line 2). We determine the heap number of the (virtual) leaf u associated with c (line 3) and start an upward traversal (lines 4–7). Here, we close the run in the current node u and start a new run (with cumulative run length equal to zero) in the following two cases: 1) we arrive from the left child of u and the current run in u is made up of `1`s, or 2) we arrive from the right child of u and the current run in u is made up of `0`s. We express this condition succinctly in line 5, where the flag `bit` indicates if the current run is of `1`s. We complement its value and prepare for the next entry in the current dictionary (line 6). We then extend the current run length by r_c (line 7). We exit the loop at the root (when $u = 1$ in line 4).

The time required to perform these actions over the whole `bwt` input stream is $O(n)$ to scan the `bwt` stream, and $O(n_r \times \log |\Sigma|)$, since we will require n_r traversals of $O(\log |\Sigma|)$ length in the wavelet tree. It turns out that $n_r = O(\min(n, nH_h))$, which proves our bound. Since $n_r \leq n$ trivially, we focus on showing that $n_r = O(nH_h)$, thus capturing precisely the high-order entropy of the text. Note that n_r is asymptotically

upper-bounded by the number of runs in the dictionaries of the internal nodes in the wavelet tree. This bound holds, since either the beginning or the end of a run in the **bwt** stream must correspond to the beginning or the end (or vice versa) of at least one distinct run in a dictionary. (Otherwise, we could extend the run also in the **bwt** stream, except possibly for the first or the last run). Now, the number of runs in the dictionaries is upper-bounded by the sum of the logarithm of their run lengths, which can be shown to be $O(nH_h)$ as in [7].

3.3 Decompression with **wzip2bwt**

Decompression is a fairly straightforward task once the encoding has been done, though some care must be taken when decomposing sets of runs. The decompression algorithm first performs a downward traversal to identify the symbol c to decompress. It then performs an upward traversal, analogous to that in **bwt2wzip**, except that it decrements the RLE values by r_c , producing in output r_c instances of c . However, the value of r_c is not necessarily the last RLE value examined along this path; rather it is the minimum among them. The reason stems from the fact that the runs in the dictionaries in the internal nodes (except for the root) may correspond to a union of runs that were disjoint in the input string **bwt**. Fortunately, the minimum value among those in an upward traversal from a leaf refers to an individual run in the **bwt** stream, and it is the value r_c .

In order to facilitate this process, we use the auxiliary information in **bwt2wzip**, with the addition of **symbol** and **alphabetsize**. The latter denotes the actual number of symbols in the **bwt** stream, and they are numbered from 0 to **alphabetsize** - 1. To recover the original value, we remap them using array **symbol**. We now comment on our main loop for decoding.

```

1 while( r_c = *(dict[u=1]) ) {
2   while ( (u = (u << 1) | bit[u]) < alphabetsize )
3     if ( *(dict[u]) < r_c ) r_c = *(dict[u]);
4   c = u - alphabetsize;
5   while ( u > 1 )
6     if ( !(*(dict[u >>= 1]) -= r_c) ) {
7       bit[u] = 1 - bit[u]; ++dict[u]; }
8   for( c = symbol[c]; r_c--; *(bwt++) = c ) ;
9 }

```

We start with the RLE value in the dictionary of the root ($u = 1$). We perform the downward traversal (lines 2–3), guided by the current run of **1**s or **0**s, looking at the flag **bit**[u] to branch either to the left (**bit**[u] = **0**) or the right (**bit**[u] = **1**) in the heap layout. We also keep the minimum RLE value in r_c , as previously mentioned. We then find the rank of the symbol to decode. Lines 4 and 8 are the analogue of line 2 in **bwt2wzip**, except that we output symbol c after remapping it, with **symbol** in the current position indicated by the **bwt** stream. The upward traversal is similar to lines 4–7 in **bwt2wzip**, except that we decrease the RLE values in the dictionaries (lines 5–7). The time required to decompress follows the same argument as for compressing.

<i>filename</i>	bwt2wzip					wzip2bwt				
	ATH	AXP	PIII	PIV	XEO	ATH	AXP	PIII	PIV	XEO
ap5.txt	4.811	2.822	2.244	4.878	5.250	6.736	4.200	3.438	6.232	6.500
bible.txt	4.093	2.688	2.162	3.473	4.370	5.302	3.656	2.910	4.746	5.037
world95.txt	3.077	2.375	1.946	2.705	3.800	3.744	3.167	2.698	3.750	4.450
calgary	4.465	3.481	2.566	4.162	5.565	6.256	5.148	3.939	5.643	6.826
canterbury	4.419	3.091	2.324	3.255	5.625	5.839	4.318	3.522	4.614	6.625

Table 2: Running times normalized with that of a simple copy routine.

3.4 Performance and experiments

In this section, we discuss our experimental setup and detail our results. We used several platforms to test our algorithms: ATH = Athlon AMD 1GHz 512MB Linux, gcc version 3.3.2 (Debian); AXP = AMD Athlon XP 1.8GHz 512MB Linux, gcc version 3.2.2 20030222 (Red Hat Linux 3.2.2-5); PIII = Intel Pentium III 1GHz 512MB Windows XP, gcc version 3.2 (mingw special 20020817-1); PIV = Pentium IV 2GHz 1GB Windows XP, gcc version 3.2 (mingw special 20020817-1), XEO = Intel Xeon 2GHz 2GB Linux, gcc version 3.3.1 20030626 (Debian prerelease). We drew our data from the Canterbury and Calgary corpuses. The first three rows of Table 2 are files from those corpora; the last two rows are the concatenation of all the files.

We compare our performance with a simple routine that copies the input `bwt` stream into another array. We normalize our routines with respect to this simple copy operation. (We don’t use `scan`, as the compiler often cheats and doesn’t actually generate code to `scan` if nothing happens. In these cases, “`scan`” is extremely fast, but misleading with regard to our experimental results.) `bwt2wzip` (compression) is just 2–6 times slower than a simple copy operation. `wzip2bwt` (decompression) is 3–7 times slower than the same. The difference in performance depends mainly on the architecture of the processor rather than the input file. (Consult Table 2 for proof of this fact, with bold figures for the minimum and the maximum.) The computation of RLE takes roughly 30% of the total time in `bwt2wzip` and 40% in `wzip2bwt`.

With regard to fine tuning performance in the code for `bwt2wzip` and `wzip2bwt`, each time we access an entry pointed by `dict[u]`, we may initiate a cache miss. Also, we need to pre-allocate more space than needed to accommodate all the dictionaries (whose final size is known at the end of the compression, which is too late). We can alleviate this problem by synchronizing the access to the decoded RLE values. In particular, we can provide the same access pattern during the execution of the compression `bwt2wzip`. Indeed, during the computation, `wzip2bwt` accesses new RLE values in some nodes along a path in the wavelet only during the upward traversal (line 7). Some care must be taken at initialization to maintain this information.

Consequently, the RLE values are scrambled among the dictionaries and follow the access pattern of `wzip2bwt`. We no longer keep a pointer in `dict[u]`, instead, we temporary store the current RLE value for `u`. As a result, except for `dict[u]`, `bit[u]` and `symbol`, the access to the other structures is sequential, which enables us to exploit the many levels of cache. Moreover, we do not need to allocate temporary storage for keeping all the RLE values that we will encode. We can produce each

RLE value and encode it on the fly. A drawback of this approach is that we lose compatibility with the text indexing functionalities mentioned in Section 1.

4 Remarks and Further Work

In this paper, we develop the simple notions of RLE and gamma encoding to achieve competitive compression ratios and extremely fast compression and decompression time. Our code does not require any additional parameters beyond the text size, alphabet size, and block size, and is tailored to work for large alphabets, e.g., Unicode, UTF/16. Our method performs integer bit assignments and does not resort to costly computation of fractional bits, as does an arithmetic coding technique. A simple copy operation is only 2–6 times faster than our compression, and only 3–7 times faster than our decompression. As a matter of fact, our encoding algorithm is so fast that the true bottleneck is the encoding and decoding of γ ! The main bottleneck remains the fast computation of the BWT.

Compared with `gzip` and `bzip2`, our compression ratio is good. We need further experiments to evaluate how much we can speed up a `bzip`-like compressor based on the computation of the BWT which is then encoded with our wavelet method. We expect that `gzip` is still faster while we need extensive experiments to compare with PPM-type algorithms. Note that data in <http://www.maximumcompression.com> shows that our method does not achieve the best ratio on the market. On the other hand, our code is open source and easy to implement, as it uses introductory material on standard compression techniques. Our wavelet encoding could be somehow related to inversion coding [1], and more prefix codes have to be compared with γ (e.g., those in [2, 3, 4, 8]). We will investigate these refinements in the full version of our paper.

References

- [1] Z. Arnavut. Move-to-front and inversion coding. DCC 2000.
- [2] S. Deorowicz. Second step algorithms in the Burrows-Wheeler compression algorithm. *Software-Practice and Experience*, 2002, 32:99–111.
- [3] P. Fenwick. Punctured Elias codes for variable-length coding of the integers. TR 137, ISSN 1173-3500, The University of Auckland, NZ, 1996.
- [4] P. Fenwick. Burrows-Wheeler compression with variable-length integer codes. *Software-Practice and Experience*, 2002, 32:1307–1316.
- [5] A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic Coding Revisited, DCC 1995.
- [6] R. Grossi, A. Gupta, J. S. Vitter. High-Order Entropy-Compressed Text Indexes. SODA 2003.
- [7] R. Grossi, A. Gupta, J. S. Vitter. When Indexing Equals Compression: Experiments with Compressing Suffix Arrays and Applications. SODA 2004.
- [8] P. G. Howard. Interleaving entropy codes. *Sequences 1997*, Positano, Italy.
- [9] M. Schindler. <http://www.compressconsult.com/rangecoder>.
- [10] The Canterbury Corpus, <http://corpus.canterbury.ac.nz>.
- [11] <http://ccrma-www.stanford.edu/~jos/mdft/Autocorrelation.html>.