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 ration than `bzip2` in less time. In fact, our compression/decompression time is dependent on H_h, the empirical hth order entropy. 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 have suffered (up to this point) from the opposite problem—inadequate compression with fixed encoding models for a large class of files. However, these static model approaches 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, with extremely simple techniques, yet very fast encoding/decoding. Two basic methods we use are those of RLE and gamma encoding, which we detail below.

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 $\lfloor \log \ell \rfloor$ in unary,

*Scuola Superiore Sant'Anna, Piazza Martiri della Libertà 33, 56127 Pisa (`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 (`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`). Support was provided in part by the ARO through MURI grant DAAH04–96–1–0013.

§Department of Computer Sciences, Purdue University, West Lafayette, IN 47907–2067 (`jsv@purdue.edu`). Support was provided in part by the Army Research Office (ARO) through grant DAAD19–01–1–0725 and by the National Science Foundation through research grant CCR–9877133.

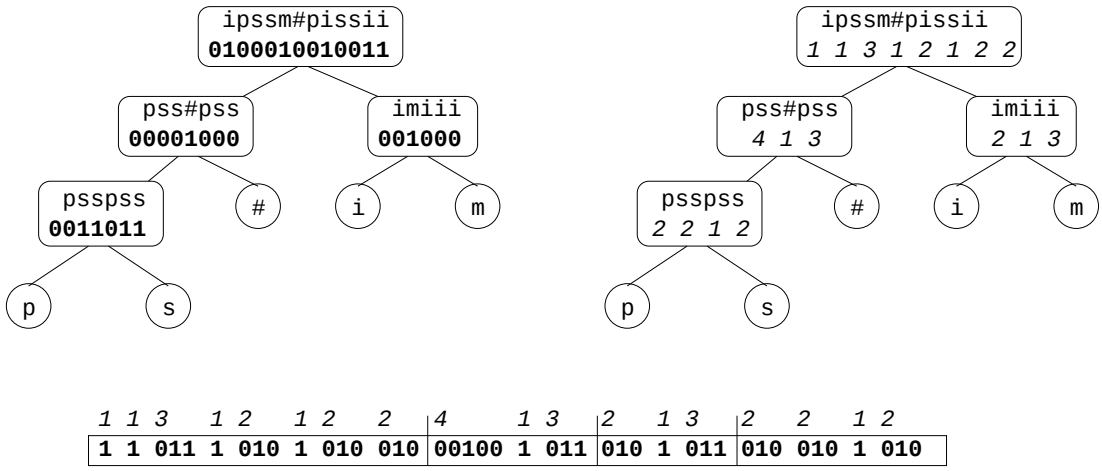

Figure 1: Example for `mississippi#`, whose transform is `ipssm#pissii`.

followed by the value of $\ell - 2^{\lfloor \log \ell \rfloor}$ encoded in binary, for a total of $2\lfloor \log \ell \rfloor$ bits. γ codes are optimal for a statistical model in which the distribution is $1/x^2$.

2 The Wavelet Tree and the Static Model

2.1 The Wavelet Tree

Grossi, Gupta, and Vitter [3] 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. Encoding the dictionaries in this way achieves the high-order entropy of the text, as per the discussion in Lemma 4.1 of [3]. Consider the example wavelet tree in Figure 1, built on the text `mississippi#`.

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 `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 runlengths previously described. The leading **0** in each node of the wavelet creates a unique association between the sequence of RLE values and the bitvector.

It has been shown both theoretically [3] and practically [4] 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 the 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

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: Bits per symbol of several codes for RLE

Value	Huffman	γ code	Value	Huffman	γ code	Value	Huffman	γ code
1	0	1	6	10100	00110	11	1100010	0001011
2	111	010	7	110101	00111	12	1010101	0001100
3	100	011	8	110000	0001000	13	11011100	0001101
4	1011	00100	9	1101111	0001001	14	11011000	0001110
5	11001	00101	10	1101001	0001010	15	11000111	0001111

Table 2:

γ codes with respect to the space required per symbol. However, as is discovered in [4], 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. There is clear empirical evidence that γ encoding is almost the best. In Section 2.3, we'll formalize this experimental finding more clearly by curve-fitting the distribution implied by γ onto the distribution of the runlengths.

Improving upon γ to encode these RLE values requires a significant amount of work with more complicated methods. For the purposes of illustration, consider the comparison of γ encoding to that of an optimal Huffman encoding for `bible.txt`, given in Table 2. Note that the γ code differs from Huffman by at most one bit. A similar skew appears in almost every file, and as such, this means that the majority of RLE values are encoded into codewords of roughly the same length by both Huffman and γ.

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 encoding/decoding time. We consider assigning a non-integral number of bits to each RLE in Section 2.3.

2.3 Statistical evidence that the distribution of RLE values fits 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` show in Figure 2. Note that this distribution is fitted by the function

$$cdf(x) = e^{\frac{-a}{x}} \qquad x \in \mathbf{N}^+, \tag{1}$$

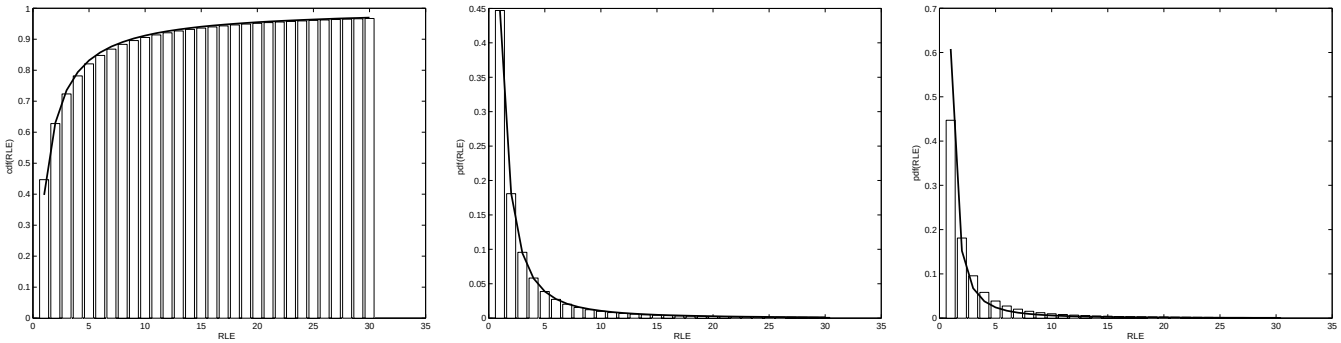

Figure 2: The x-axis shows the distinct RLE values for `bible.txt`, in increasing order. Left: The empirical cumulative distribution together with our fitting function *cdf*. Center: The empirical probability density function together with our fitting function *pdf*. Right: The empirical probability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

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 $a \in [0.5 \ldots 1.8]$ 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, after renormalization, we obtain the corresponding probability density function:

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

where the term $\sum_{i=1}^{\infty} \frac{ae^{\frac{-a}{i}}}{i^2}$ is the normalization factor. As one can see from Figure 2, the function (2) fits the empirical probability density of the RLE values computed for file `bible.txt` well, suggesting that the approximation of the *cdf* to a continuous function leads to a negligible error.[1] As one can see, $pdf(x) \sim \frac{1}{x^2}$ as x approaches infinity, i.e.,

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

Since the γ code is optimal for distributions proportional to $\frac{1}{x^2}$, 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] The results are not satisfying as compared to γ, as the range of RLE values to encode is unlimited and this is not the best situation to work with an arithmetic encoder.

[1] We 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.

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

We then employed 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 tends to yield good results on most files.) To this end, we took a free and fast arithmetic-like coder called range coder [6], employed in `szip`. We encode the RLE value r by assigning it an interval of length $cdf(r+1) - cdf(r) = pdf(r)$. (This appears to be faster than using the cumulative counts of the frequency of values already scanned, analogously to the 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 transformed our arithmetic compressor so that the parameter a could be changed adaptively during the execution, hoping for a better compression ratio. We needed a cue to infer a from the values already read, and decided to use an MLE (maximum likelihood estimation) algorithm (described below).

The main hurdle to simply using MLE is its assumption of independent trials. (In our terminology, this would have to mean that each of the runlengths is independently drawn from its pdf.) Though we could not find a satisfiable measure for this notion, we did measure the autocovariance (normalized) of the RLE values. This method is widely adopted in signal theory [1] as a good indicator on 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 of it.

With this observation in mind, we assume statistical independence of the RLE values in order to define the likelihood function

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

We want to find the value a where $l(x)$ reaches its maximum. Equivalently, we can find where the log of the function reaches its maximum (log-likelihood function):

$$L_x(a, x_1 \dots x_k) = \log l_x(a, x_1 \dots x_k) = -k \log \Big(\sum_{i=1}^{\infty} \frac{e^{\frac{-a}{i}}}{i^2} \Big) - 2 \sum_{i=1}^{\infty} \log(x_i) - a \sum_{i=1}^{\infty} \frac{1}{x_i}$$

The function is differentiable and we can take the derivative of it with respect to a, equating to zero. This yields

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

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 it is very difficult to compute even for a 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 $a \in [0.5 \dots 1.8]$ (which is the range of interest for us) with the second-degree polynomial:

$$a = 6.96 - 16.4912 \cdot H(x)^{-1} + 10.6186 \cdot H(x)^{-2} \tag{3}$$

In our code, 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 at Table 1, the improvement is negligible, ranging from 1% to 2% in the best cases. The best case is the file `random.txt` belonging to the Calgary corpus, for which the hypothesis of independence of the RLE values holds with high probability by its very construction.

3 Wzip: A Simple Tool for Fast Compression and 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. The last block encodes (n mod b) symbols if n is not a multiple of b. 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:

- 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 want to store such bitvector in the header, and then smaller bitvectors in the blocks that refer only to the symbols stored in the bitvector in the header).
- Note that the wavelet tree has α implicit leaves and $\alpha - 1$ internal nodes with dictionaries (see Figure 1). We store the dictionaries encoded with RLE+γ by taking them in heap numbering and by concatenating their encoding. (We may want to byte-align the encoding of each dictionary, but this is not necessary).

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 mod 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, up 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 `bwt` stream) of the file and compresses it efficiently using our wavelet tree techniques. Our method introduces a novel method of creating the wavelet tree, which relates the speed of compression to the compressibility of the input. This behavior is a key observation, and introduces a new consideration into the notion of compressibility—highly uniform data should be easier to handle, both in terms of space and time.

If we were to build the wavelet tree naively from the `bwt` stream, we would run multiple scans on `bwt` to set up the bitvector in each individual node, as shown in Figure 1. Then, we would compress the resulting dictionaries with RLE+γ. A single-scan method is made possible by placing one item at a time, in each of the internal nodes from its root-to-leaf path via an upward walk. Since, for each node, process could take up to $O(\log |\Sigma|)$ time, it requires $O(n \log |\Sigma|)$ in total. We describe a refinement of this construction method requiring 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.

Let c be the current symbol in `bwt` and let u be its corresponding leaf in the wavelet tree. (Recall that the numbering of internal nodes follows the heap layout.) 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 on an individual basis, but 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`. First 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 iff we are encoding a run of 1s.

Below, we describe the main loop of the compression. We do not specify the task of encoding the RLE values with gamma 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), either when we arrive from the left child of u and the current run in u is made up of 1s, or when we arrive from the right child of u and the current run in u is made up of 0s. We express this condition succinctly in line 5, where the flag `bit` indicates if the current run is of 1s or not. We complement the value of that flag, 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 arrival in 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 is true, since either the beginning or the end of a run in `bwt` 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 `bwt`, except possibly for the first or the last run in `bwt`). Now, the number of runs in the dictionaries is upper bounded by the sum of the logarithm of their runlengths, which is $O(nH_h)$ as shown in [4].

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. Then it 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 for this is due to 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 `bwt`, 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 `bwt`, and they are numbered from 0 to `alphabetsize` - 1. To recover the original value, we remap them using array `symbol`. We are now ready to comment on our main loop for decoding. Again, we do not describe how to decode the RLE values with gamma code as it is a standard task.

```
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 ($u = 1$). When this value is 0, we have completed the task of decoding all the symbols. We perform the downward traversal (lines 2–3) guided by the current run of **1**s or **0**s, by looking at the flag `bit`$[u]$ to branch either to the left (`bit`$[u] = \mathbf{0}$) or to the right (`bit`$[u] = \mathbf{1}$) 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 be decoded. Lines 4 and 8 are the counterpart of line 2 in `bwt2wzip`, except that we output symbol c after remapping it with `symbol` in the current position indicated by `bwt`. The upward traversal is like that of lines 4–7 in `bwt2wzip`, except that we decrease the RLE values in the dictionaries (lines 5–7).

The time required for decompression follows the same argument as that for compressing.

	bwt2wzip					wzip2bwt				
filename	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**
cantrbry	4.419	3.091	2.324	3.255	**5.625**	5.839	4.318	3.522	4.614	6.625

Table 3:

3.4 Performance and experiments

In this section, we discuss our experimental setup and detail our results.

We used a number of 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 3 are individual files from those corpora, and the last two rows are the concatenation of all the files.

The performance is compared with that of simple routine that copies the input `bwt` into another array. We normalize our routines with the 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, and misleading with regard to our experimental results.) `bwt2wzip`(compression) is just between 2 and 6 times slower than a simple copy operation. `wzip2bwt`(decompression) is between 3 and 7 times slower than the same. The difference in performance depends mainly on the architecture of the processor on the platform, rather than the input file. (Consult Table 3 for corroboration 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 perfomance, in the code for `bwt2wzip` and `wzip2bwt`, each time we access an entry pointed by `dict`$[u]$, we may give rise to a cache miss. Also, we need to pre-allocate more space than needed to accomodate 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. Consider the first time that `wzip2bwt` accesses each decoded RLE value. By running an example, the reader may be convinced that we can provide the same access pattern during the execution of `bwt2wzip`. Indeed, during the computation, `wzip2bwt` accesses the 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 several 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 soon, on

the fly. A drawback of this approach is that we lose compatibility with the text indexing functionalities mentioned in Section 1.

4 Conclusions

In this paper, we developed the simple notions of RLE and gamma encoding to achieve competitive compression ratios and extremely fast time to compress/decompress. Our code does not require any additional parameters beyond the text size, alphabet size, and block size. Our method 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 simply byte-wise copy is only 2–6 times faster than our compression, and only 3–7 times faster than our decompression. As a matter of fact, our compression algorithm is so fast that the true bottleneck is the encoding/decoding of γ!

Compared to `gzip` and `bzip`, our compression ratio is good. However, data in `http://www.maximumcompression.com` shows that it does not achieve the best ratio on the market. On the other hand, the code is open source and easy to implement as it uses introductory material on standard compression techniques.

References

[1] http://ccrma-www.stanford.edu/ jos/mdft/Autocorrelation.html

[2] A. Moffat, R. M. Neal and I. H. Witten Arithmetic Coding Revisited, DCC95.

[3] R. Grossi, A. Gupta, J. S. Vitter. High-order entropy-compressed text indexes. SODA 2003.

[4] R. Grossi, A. Gupta, J. S. Vitter. When Indexing Equals Compression: Experiments with Compressing Suffix Arrays and Applications SODA 2004.

[5] The Canterbury Corpus, http://corpus.canterbury.ac.nz.

[6] www.compressconsult.com/rangecoder/