Space Efficient Algorithms for the Burrows-Wheeler Backtransformation

Ulrich Lauther Tamás Lukovszki

Siemens AG, Corporate Technology
81730 Munich, Germany
{Ulrich.Lauther,Tamas.Lukovszki}@siemens.com

Abstract The Burrows-Wheeler transformation is used for effective data compression, e.g., in the well known program bzip2. Compression and decompression are done in a block-wise fashion; larger blocks usually result in better compression rates. With the currently used algorithms for decompression, $4n$ bytes of auxiliary memory for processing a block of n bytes are needed, $0 < n < 2^{32}$. This may pose a problem in embedded systems (e.g., mobile phones), where RAM is a scarce resource. In this paper we present algorithms that reduce the memory need without sacrificing speed too much.
The main results are: Assuming an input string of n characters, $0 < n < 2^{32}$, the reverse Burrows-Wheeler transformation can be done with 1.625 n bytes of auxiliary memory and $O(n)$ runtime, using just a few operations per input character. Alternatively, we can use n/t bytes and 256 t n operations. The theoretical results are backed up by experimental data showing the space-time tradeoff.

1 Introduction

The Burrows-Wheeler transformation (BWT) [6] is at the heart of modern, very effective data compression algorithms and programs, e.g., bzip2 [13]. BWT-based compressors usually work in a block-wise manner, i.e., the input is divided into blocks and compressed block by block. Larger block sizes tend to result in better compression results, thus bzip2 uses by default a block size of 900,000 bytes and in its low memory mode still 100,000 bytes. The standard algorithm for decompression (reverse BWT) needs auxiliary memory of 4 bytes per input character, assuming 4-byte computer words and thus $n < 2^{32}$. This may pose a problem in embedded systems (say, a mobile phone receiving a software patch over the air interface) where RAM is a scarce resource. In such a scenario, space requirements for compression ($8n$ bytes when a suffix array [10] is used to calculate the forward BWT) is not an issue, as compression is done on a full fledged host. In the target system, however, cutting down memory requirements may be essential.

1.1 The BWT Backtransformation

We will not go into details of the BW-transformation here, as it has been described in a number of papers [2,4,6,7,8,11] and tutorials [1,12] nor do we give a

proof of the reverse BWT algorithm. Instead, we give the bare essentials needed to understand the problem we solve in the following sections. The BWT (conceptually) builds a matrix whose rows contain n copies of the n character input string, row i rotated i steps. The n strings are then sorted lexicographically and the last column is saved as the result, together with the "primary index", i.e., the index of the row that contains - after sorting - the original string. The first column of the sorted matrix is also needed for the backtransformation, but it needs not to be saved, as it can be reconstructed by sorting the elements of the last column. (Actually, as we will see, the first column is also needed only conceptually.)

Figure 1 shows the first and last columns resulting from the input string "CARINA". The arrow indicates the primary index. Note that we have numbered the occurrences of each character in both columns, e.g., row 2 contains the occurrence 0 of character "A" in L, row 5 contains occurrence 1. We call these numbers the *rank* of the character within column L.

	F		L	rank	base
0	A	0	N	0	A : 0
1	A	1	C	0	C : 2
⇨ 2	C	0	A	0	I : 3
3	I	0	R	0	N : 4
4	N	0	I	0	R : 5
5	R	0	A	1	

Figure 1. First (F) and last (L) column for the input string "CARINA".

To reconstruct the input string, we start at the primary index in L and output the corresponding character, "A", whose rank is 0. We look for A_0 in column F, find it at position 0 and output "N". Proceeding in the same way, we get "I", "R", "A", and eventually "C", i.e., the input string in reverse order. The position in F for a character/rank pair can easily be found if we store for each character of the alphabet the position of its first occurrence in F; these values are called *base* in Figure 1.

This gives us a simple algorithm when the vectors *rank* and *base* are available:

```
int h = primary_index;
for (int i = 0; i < n; i++) {
  char c = L[h];
  output(c);
  h = base[c] + rank[h];
}
```

The *base*-vector and *rank* can easily be calculated with one pass over L and another pass over all characters of the alphabet. (We assume an alphabet of 256 symbols throughout this paper.)

```
for (int i = 0; i < 256; i++) base[i] = 0;
for (int i = 0; i < n; i++) {
  char c = L[i];
  rank[i] = base[c];
  base[c]++;
}
```

```
int total = 0;
for (int i = 0; i < 256; i++) {
  int h = base[i];
  base[i] = total;
  total  += h;
}
```

These algorithms need $O(n)$ space (n words for the *rank*-vector) and $O(n)$ time. Alternatively, we could do without precalculation of rank-values and calculate $rank[h]$ whenever we need it, by scanning L and counting occurrences of $L[h]$. This would give us $O(1)$ space and $O(n^2)$ time.

The question, now, is: is there a data structure that needs significantly less than n words without increasing run time excessively?

In this paper we present efficient data structures and algorithms solving following problems:

Rank searching: The input must be preprocessed into a data structure, such that for a given index i, it supports a query for $rank(i)$. This query is referred to as rank-query.

Rank-position searching: The input must be preprocessed into a data structure, such that for a given character c and rank r, it supports a query for index i, such that $rank(i) = r$. This query is referred to as rank-position-query. (This allows traversing L and F in the direction opposite to that discussed so far, producing the input string in forward order).

1.2 Computation Model

As computation model we use a random access machine (RAM) (see e.g., in [3]). The RAM allows indirect addressing, i.e., accessing the value at a relative address, given by an integer number, in constant time. In this model it is also assumed that the length of the input n can be stored in a computer word. Additionally, we assume that the size $|A|$ of the alphabet A is a constant, and particularly, $|A| - 1$ can be stored in a byte. Furthermore, we assume that a bit shift operation in a computer word, word-wise *and* and *or* operations, converting a bit string stored in a computer word into an integer number and vice-versa and algebraic operations on integer numbers ('+', '-', '*', '/', 'mod', where '/' denotes the integer division with remainder) are possible in constant time.

1.3 Previous Results

In [14] Seward describes a slightly different method for the reverse BWT by handling the so-called transformation vector in a more explicit way. He presents several algorithms and experimental results for the reverse BWT and answering rank-queries (more precisely, queries "how many symbols x occur in column L up to position i?", without the requirement $L[i] = x$). A rigorous analysis of the algorithms is omitted in [14]. The algorithms described in [14], `basis` and `bw94`

need $5n$ bytes of memory storage and support a constant query time; algoritm `MergedTL` needs $4n$ bytes if n is limited to 2^{24} and supports a constant query time. The algorithm `indexF` needs $2.5n$ bytes if $n < 2^{20}$ and $O(\log |A|)$ query time. The algorithms `tree` and `treeopt` build 256 trees (one for each symbol) on sections of the vector L. They need $2n$ and $1.5n$ bytes, respectively, if $n < 2^{20}$ and support $O(\log(n/\Delta)+c_x\Delta)$ query time, where Δ is a given parameter depending on the allowed storage and c_x is a relatively big multiplicator which can depend on the queried symbol x.

1.4 Our Contributions

We present a data structure which supports answering a rank-query $Q(i)$ in $O(1)$ time using $n(\frac{\ell-1}{8} + \frac{w|A|}{2^\ell})$ bytes, where w denotes the length of a computer word in bytes, and $|A|$ is the size of the alphabet. If $|A| \leq 256$ and $w = 4$ (32 bit words), by setting $\ell \in \{12, 13\}$, we obtain a data structure of $\frac{13}{8}n$ or 1.625 bytes. For $w = 2$ we get a data structure of $\frac{25}{16}n$ or 1.5625 bytes. Thus, the space requirement is strictly less than that of the trivial data structure, which stores the rank for each position as an integer in a computer word and that of the methods in [14] with constant query time. The preprocessing needs $O(n)$ time and $O(|A|)$ working storage.

We also present data structures of n bytes, where we allow at most $L = 2^9$ sequential accesses to a data block of L bytes. Because of caching and hardware prefetching mechanism of todays processors, with this data structure we obtain a reasonable query time.

Furthermore, we present a data structure, which supports answering a rank-query $Q(i)$ in $O(t)$ time using t random accesses and $c \cdot t$ sequential accesses to the memory storage, where c is a constant, which can be chosen, such that the speed difference between non-local (random) accesses and sequential accesses is utilized optimally. The data structure needs $\frac{n(8+|A|\log ct)}{8ct} + \frac{n|A|w}{ct^2}$ bytes. For $t = \omega(1)$, this results in a sub-linear space data structure, e.g., for $t = \Theta(n^{1/d})$ we obtain a data structure of $\frac{1}{d}n^{1-1/d}|A|(1 + o(1))$ bytes. The preprocessing needs $O(n)$ time and $O(|A|)$ working storage.

After this, we turn to the inverse problem, the problem of answering rank-position-queries. We present a data structure of $n(\frac{|A|(\ell+8w)}{2^\ell} + \ell)$ bits, which supports answering rank-position-queries in $O(\log(n/2^\ell))$ time. The preprocessing needs $O(n)$ time and $O(|A| + 2^\ell)$ working storage. For $\ell = 13$, we obtain a data structure of $14\frac{3}{8} \cdot n$ bits.

Finally, we present experimental results, that show that our algorithms perform quite well in practice. Thus, they give significant improvement for decompression in embedded devices for the mentioned scenarios.

1.5 Outline of the Paper

In Section 2 we describe various data structures supporting rank-queries. Section 3 is dedicated to data structures for rank-position-queries. In Section 4 we

present experimental results for the reverse BWT. Finally, in Section 5 we give conclusions and discuss open problems.

2 Algorithms for Rank-Queries

Before we describe the algorithm we need some definitions. For a string S of length n (i.e. of n symbols) and integer number i, $0 \leq i < n$ we denote by $S[i]$ the symbol of S at the position (or index) i, i.e., the index counts from 0. For a string S and integers $0 \leq i, j < n$, we denote by $S[i..j]$ the substring of S starting at index i and ending at j, if $i \leq j$, and the empty string if $i > j$. $S[i..i] = S[i]$ is the symbol at index i. The rank of symbol $S[i]$ at the position i is defined as $rank(i) := |\{j : 0 \leq j < i, S[j] = S[i]\}|$, i.e., the rank of the kth occurrence of a symbol is $k - 1$.

2.1 A Data Structure of $\frac{13}{8} \cdot n$ Bytes

We divide the string S into $n' = \lceil n/L \rceil$ blocks, each of $L = 2^\ell$ consecutive symbols (bytes). L will be determined later. (The last block may contain less than L symbols.) The jth block $B[j]$ of the string S, $0 \leq j < n'$, starts at the position $j \cdot L$, i.e. $B[j] = S[j \cdot L \,..\, \min\{n, (j+1)L\} - 1]$.

In our data structure, for each block $B[j]$, $0 \leq j < n'$ and each symbol $x \in A$, we store an integer value $b[j, x]$, which contains the number of occurrences of symbol x in blocks $B[0], ..., B[j]$, i.e. in $S[0..L(j+1) - 1]$. In the following we assume $b[-1, x] = 0$, $x \in A$, but there is no need to store these values explicitly. For storing the values $b[j, x]$, $0 \leq j < n'$, $x \in A$, we need $n'|A| = \lceil n/L \rceil |A|$ computer words, i.e. $\lceil n/L \rceil 8w|A|$ bits.

Additionally, for each index i, $0 \leq i < n$, we store the *reduced* rank $r[i]$ of the symbol $x = S[i]$, $r[i] = rank(i) - b[i/L - 1, x]$. Then a rank query $Q(i)$ obviously can be answered by reporting the value $b[i/L - 1, x] + r[i]$. Note that $0 \leq r[i] < L$, and thus, each $r[i]$ can be stored in ℓ bits. We can save an additional bit (or equivalently, double the block size), if we define the reduced rank $r[i]$ in a slightly different way:

$$r[i] = \begin{cases} rank(i) - b[i/L - 1, x] & \text{if} \quad i \bmod L < L/2, \\ b[i/L, x] - rank(i) - 1 & \text{otherwise.} \end{cases}$$

For storing the whole vector r, we need $n \cdot (\ell - 1)$ bits.

Storage requirement: The storage requirement for storing $r[i]$ and $b[j, x]$, $0 \leq i < n$, $0 \leq j < n'$, $x \in A$ is $n(\ell - 1 + \frac{8w|A|}{L}) = n(\ell - 1 + 8w|A|2^{-\ell})$ bits. We obtain the continuous minimum of this expression at the point, in which the derivative is 0. Thus, we need $1 + 8w|A|2^{-\ell}(-\ln 2) = 0$. After reorganizing this equality, we obtain

$$2^{-\ell} = \frac{1}{8w|A| \ln 2} \qquad \text{and thus}$$
$$\ell = \log(8w|A|) + \log \ln 2.$$

Since $|A| \leq 256$, for $w = 4$ (32 bit words), the continuous minimum is reached in $\ell \approx 3 + 2 + 8 - 0.159$. Setting $\ell = 12$ or $\ell = 13$ (and the block size $L = 4096$ or $L = 8192$, respectively), the size of the data structure becomes $\frac{13}{8}n$ bytes.

Computing the values $b[j, x]$ and $r[i]$: The values $b[j, x]$, $0 \leq j < n'$, $x \in A$, can be computed in $O(n)$ time using $O(|A|)$ space by scanning the input as follows. We maintain an array b_0 of $|A|$ integers, such that after processing the ith symbol of the input string S, $b_0[x]$ will contain the number of occurrences of symbol x in $S[0..i]$. At the beginning of the algorithm we initialize each element of b_0 to be 0. We can maintain the above invariant by incrementing the value of $b_0[x]$, when we read the symbol x. After processing the symbol at a position i with $i \equiv -1 \bmod L$, i.e. after processing the last symbol in a block, we copy the array b_0 into $b[i/L]$.

The values of $r[i]$, $0 \leq i < n$, are computed in a second pass, when all the b-values are known. Here, we maintain an array r_0 of $|A|$ integers, such that just before processing the ith symbol of the input string S, $r_0[x]$ will contain the number of occurrences of symbol x in $S[0..i-1]$. Thus, we can set $r[i] = r_0[x] - b[i/L - 1, x]$ or $r[i] = b[i/L] - r_0[x] - 1$, respectively. At the beginning of the algorithm we initialize each element of r_0 to be 0. We can maintain the above invariant by incrementing the value of $r_0[x]$, after reading the symbol x.

Clearly, the above algorithm needs $O(n)$ time and $O(|A|)$ working storage (for the arrays b_0 and r_0).

Answering a query $Q(i)$: If we have the correct values for $r[i]$ and $b[j, x]$, $0 \leq i < n$, $0 \leq j < n'$, $x \in A$, then a query $Q(i)$, $0 \leq i < n$ can be answered easily by determining the symbol $x = S[i]$ in the string S and combining the reduced rank $r[i]$ with the appropriate b-value:

$$rank(i) = \begin{cases} r[i] + b[i/L, x] & \text{if} \quad i \bmod L < L/2, \\ b[i/L + 1, x] - r[i] - 1 & \text{otherwise.} \end{cases}$$

This sum can be computed using at most 2 memory accesses and a constant number of unit time operations on computer words. Summarizing the results of this section we obtain the following.

Theorem 1. *Let S be a string of length n. S can be preprocessed into a data structure which supports answering a rank query $Q(i)$ in $O(1)$ time. The data structure uses $n(\ell - 1 + 8w|A|/2^\ell)$ bits, where w is the number of bytes in a computer word. For $|A| \leq 256$, $w = 4$, and $\ell \in \{12, 13\}$, the size of the data structure is $\frac{13}{8}n$ bytes. The preprocessing needs $O(n)$ time and $O(|A|)$ working storage.*

Remark: If the maximum number of occurrences of any symbol in the input is smaller than the largest integer that can be stored in a computer word, i.e. $n < 2^p$ and $p < 8w$, then we can store the values of $b[j, x]$ using p bits instead of a complete word. Then the size of the data structure is $n(\ell - 1 + \frac{p|A|}{2^\ell})$ bits. For instance, for $p = 16$ we obtain a data structure of $n\frac{25}{16}$ bytes, and for $p = 24$ one of $n\frac{51}{32}$ bytes.

Utilizing processor caching – data structure of $\leq n$ bytes: The processors in modern computers use caching, hardware prefetching, and pipelining techniques, which results in significantly higher processing speed, if the algorithm accesses consecutive computer words than in the case of non-local accesses (referred to as random accesses). In case of processor caching, when we access a computer word, a complete block of the memory will be moved into the processor cache. For instance, in Intel Pentium 4 processors, the size of such a block (the so-called L2 cache line size) is 128 bytes (see, e.g. [9]). Using this feature, we also obtain a fast answering time, if we get rid of storing the values of $r[i]$, but instead of this, we compute $r[i]$ during the query by scanning the block (of L bytes) containing the string index i. More precisely, it is enough to scan the half of the block: the lower half in increasing order of indices, if $i \bmod L < L/2$, and the upper half in decreasing order of indices, otherwise. In that way we obtain a data structure of size $n|A|w/L$ bytes.

Theorem 2. *Let S be a string of length n. S can be preprocessed into a data structure $D(S)$, which supports answering a rank query $Q(i)$ by performing 1 random access to $D(S)$ (and to S) and at most $L/2$ sequential accesses to S. The data structure uses $n|A|w/L$ bytes, where w is the length of a computer word in bytes. The preprocessing needs $O(n)$ time and $O(|A|)$ working storage.*

For $|A| \leq 256$, $w = 4$ and $L = 2^{10}$, the size of the data structure is n bytes. If $n < 2^p$, $p < 8w$, then we get a data structure of n bytes by using a block size of $L = p|A|/8$ bytes.

2.2 A Sub-linear Space Data Structure

In this Section we describe a sub-linear space data structure for supporting rank-queries in strings in $O(t)$ time for $t = \omega(1)$.

Similarly to the data structure described in Section 2.1, we divide the string S into $n^* = \lceil \frac{n}{c \cdot t} \rceil$ blocks, each of size $L = c \cdot t$ bytes, where c is a constant. (The constant c can be determined for instance, such that the effect of processor caching and pipelining is being exploited optimally).

We store the values $b^*[j, x]$, where $b^*[j, x]$ is the number of occurrence of symbol $x \in A$ in the jth block. The value of $b^*[j, x]$ is stored as a bit-string of $\lfloor \log ct \rfloor + 1$ bits. Note that $0 \leq b^*[j, x] \leq L$. Let $b'[j, x] = b^*[j, x] \bmod L$. Then $0 \leq b'[j, x] \leq L$, and thus, each $b'[j, x]$ can be stored in $\log ct \rfloor + 1$ bits. Furthermore, the value of $b^*[j, x]$ can be reconstructed from the value of $b'[j, x]$ in constant time: if $b'[j, x] = 0$ and an arbitrary symbol (say the first symbol) of block $B[j]$ is equal to x, then $b^*[j, x] = L$, otherwise $b^*[j, x] = b'[j, x]$. For this test, we store in $c[j]$ the first symbol in $B[j]$, for each block $B[j]$. Storing $b^*[j, x]$ and $c[j]$, for $0 \leq j < \lceil \frac{n}{c \cdot t} \rceil$ and $x \in A$ needs $\frac{n \cdot (8 + |A| \log cn)}{c \cdot t}$ bits, i.e. $\frac{n \cdot (8 + |A| \log cn)}{8 \cdot c \cdot t}$ bytes.

Additionally to the linear space data structure, the blocks are organized in $\hat{n} = \lceil \frac{n}{ct^2} \rceil$ super-blocks, such that each super-block contains t consecutive blocks. We compute the values of $\hat{b}[k, x]$, for $0 \leq k < \lceil \frac{n}{c \cdot t^2} \rceil$ and $x \in A$, such that $\hat{b}[k, x]$

contains the number of occurrences of symbol x in the super-blocks $0, ..., k$. These values are stored as integers. Storing all values needs $\frac{n \cdot |A|}{c \cdot t^2}$ computer words.

The values of $b^*[j, x]$ and $\hat{b}[k, x]$, $0 \leq j < n^*$, $0 \leq k < \hat{n}$, $x \in A$ can be computed in $O(n)$ time using $O(|A|)$ space by scanning the input string S in a similar way as in Section 2.1.

Answering a query: Using this data structure, a query $Q(i)$ can be answered as follows. Let $\hat{B}$ be the super-block containing the query position i, i.e. $\hat{B}$ is the super-block with index $k = i/(c \cdot t^2)$. Let B be the block containing the position i, i.e. B is the block with index $j^* = i/(c \cdot t)$. Let $x = S[i]$. The query $Q(i)$ can be answered by summing $\hat{b}[k-1, x]$ and the values of $b^*[j, x]$, for each index j of a block in the super-block $\hat{B}$, such that $j < j^*$, i.e. $(i/(c \cdot t^2)) \cdot t \leq j < j^*$. Then we scan the block B and compute the rank $r[i]$ of $S[i]$ in block B during the query time (by scanning the half of the block, as described in the previous section). Then

$$rank(i) = r[i] + \hat{b}[i/(c \cdot t^2) - 1, x] + \sum_{j=(i/(c \cdot t^2)) \cdot t}^{i/(c \cdot t)-1} b^*[j, x].$$

Since we have one random access to the value $\hat{b}[k-1, x]$, at most t random accesses to the values of $b^*[j, x]$, and $c \cdot t/2$ sequential accesses to the input string, we can perform a rank-query in $O(t)$ time.

Summarizing the description, we obtain:

Theorem 3. *Let S be a string of length n. S can be preprocessed into a data structure which supports answering a rank-query $Q(i)$ in $O(t)$ time. The data structure uses $n(\frac{|A|(8+\log ct)}{ct} + \frac{8w|A|}{ct^2})$ bits. For $t = \omega(1)$, it uses $(\frac{n \cdot (8+|A| \log ct)}{ct})(1+o(1))$ bits. The preprocessing needs $O(n)$ time and $O(|A|)$ working storage.*

Note, that for a block size of $L = ct = 2^9$ bytes and $t \geq \frac{16}{7}w$, we obtain a data structure of n bytes. With other words, we can guarantee a data structure of n bytes using smaller blocks than in Section 2.1 to the cost of t random storage accesses.

Corollary 1. *Let S be a string of length n. S can be preprocessed into a data structure of n bytes, which for $t \geq \frac{16}{7}w$, supports answering a rank-query $Q(i)$ in $O(t)$ time using t random accesses and $ct/2$ sequential accesses. The preprocessing needs $O(n)$ time and $O(|A|)$ working storage.*

If we allow in Theorem 3, for instance, a query time $O(n^{1/d})$, then we can store the value of $b^*[j, x]$ in $\frac{\log n}{d}$ bits and the whole matrix b^* in $\frac{1}{d}n^{1-1/d}|A|$ computer words.

Corollary 2. *Let S be a string of length n. S can be preprocessed into a data structure which supports answering a rank-query $Q(i)$ in $O(n^{1/d})$ time. The data structure uses $\frac{1}{d}n^{1-1/d}|A|(1+o(1))$ computer words. The preprocessing needs $O(n)$ time and $O(|A|)$ working storage.*

3 An Algorithm for Rank-Position-Queries

In this Section we consider the inverse problem of answering rank-queries, the problem of answering rank-position-queries. A rank-position-query $Q^*(x,k)$, $x \in A$, $r \in \mathbb{N}_0$, reports the index $0 \leq i < n$ in the string S, such that $S[i] = x$ and $rank(i) = k$, if such an index exist, and "no index" otherwise. We show, how to preprocess S into a data structure of $n(\frac{|A|w}{L} + \frac{\ell}{8})$ bytes, which supports answering rank-position-queries in $O(\log(n/L))$ time.

We divide the string S into $n' = \lceil n/L \rceil$ blocks, each containing $L = 2^\ell$ consecutive symbols. The rank-position-query $Q^*(x,k)$ will work in two steps:

1. Find the block $B[j]$, which contains the index of the kth occurrence of x in S, and determine k_0 the overall number of occurrences of x in the blocks $B[0], ..., B[j-1]$.
2. Find the relative index i' of the $k'(:= k - k_0)$th occurrence of x within $B[j]$, if i' exists, and return with index $i = i' + jL$, and return with "no index" otherwise.

Data structure for Step 1: For each block $B[j]$, $0 \leq j < n'$ and each symbol $x \in A$, we store an integer value $b[j,x]$, which contains the overall number of occurrences of symbol x in blocks $B[0], ..., B[j-1]$, i.e. in $S[0 \,..\, jL-1]$. For storing the values $b[j,x]$, $0 \leq j < n'$, $x \in A$, we need $n'|A| = \lceil n/L \rceil |A|$ computer words, i.e. $\lceil n/L \rceil 8w|A|$ bits. The values of all $b[j,x]$ can be computed in $O(n)$ time using $O(|A|)$ working storage.

Let j be the largest number, such that $b[j,x] < k$. Then $B[j]$ is the block, which contains the index of the kth occurrence of x, if S contains at least k occurrences of x, and $B[j]$ is the last block otherwise. We set $k_0 = b[j,x]$. Using this data structure, Step 1 can be performed in $O(\log(n/L))$ time by logarithmic search for determining j.

Data structure for Step 2: For each block $B[j]$, $0 \leq j < n'$ and each symbol $x \in A$, we store a sorted list $p[j,x]$ of relative positions of the occurrences of symbol x in $B[j]$, i.e. if $B[j][i'] = x$, $0 \leq i' < L$, then $p[j,x]$ contains an element for i'. The relative index i' of the k'th occurrence of x in $B[j]$ is the $k'th$ element of the list $p[j,x]$. Note, that the overall number of list elements for a block $B[j]$ is L and each relative position can be stored in ℓ bits. Therefore, we can store all lists for $B[j]$ in an array $a[j]$ of L elements, where each element of $a[j]$ consists of ℓ bits. Additionally, for each $0 \leq j < n'$ and $x \in A$, we store in $s[j,x]$ the start index of $p[j,x]$ in $a[j]$. Since $0 \leq s[j,x] < L$, $s[j,x]$ can be stored in ℓ bits. Therefore, the storage requirement of storing $a[j]$ and $s[j,x]$, $0 \leq j < n'$, $x \in A$, is $n\ell + n\ell|A|/L$ bits. These values can be computed in $O(n)$ time using $O(L + |A|)$ working storage. (First we scan $B[j]$ and build linked lists for $p[j,x]$: for $0 \leq i' < n'$, if $B[j][i'] = x$, then we append a list element for i' to $p[j,x]$. Then we compute $a[j]$ and $s[j,x]$ for each $x \in A$ from the linked lists.) Let $k' = k - k_0$. Then the index i' of the k'th occurrence of symbol x in $B[j]$ can be computed in $O(1)$ time: $i' = a[j][s[j,x] + k' - 1]$, if $s[j,x] + k' < s[j,x+1]$, where $x+1$ is the symbol following x in the alphabet A. Otherwise, we return "no index".

Summarizing the description of this Section we obtain the following.

Theorem 4. *Let S be a string of length n and $L = 2^\ell$. S can be preprocessed into a data structure which supports answering a rank-position-query $Q^*(x,k)$ in $O(\log(n/L))$ time. The data structure uses $n(\frac{8w|A|}{L} + \ell + \frac{\ell|A|}{L})$ bits, where w is the number of bytes in a computer word. For $|A| \leq 256$, $w = 4$, and $\ell = 12$, the size of the data structure is $14\frac{3}{4} \cdot n$ bits, and for $\ell = 13$ it is $14\frac{3}{8} \cdot n$ bits. The preprocessing needs $O(n)$ time and $O(|A| + L)$ working storage.*

Remark: If we do not store the values of $p[j,x]$, but instead of this, we compute the relative index of the k'th occurrence of x for $Q^*(x,k)$ during the query time, we can obtain a sub-linear space data structure at the cost of longer query times.

4 Experimental Results

As each rank value is used exactly once in the reverse BWT, the space and runtime requirements depend solely on the size of the input, not on its content - as long as we ignore caching effects. Therefore, we give a first comparison of algorithms based on only one file. We used a binary file with 2542412 characters as input. Experiments were carried out on a 1 GHz Pentium III running Linux kernel 2.4.18 and using g++-3.4.1 for compilations. When implementing the $\frac{13}{8}n$ data structure we choose the variant with $L = 2^{13}$, where the rank values are stored in 12 bits (1.5 bytes). That allows reasonably fast access without too much bit fiddling. Our BWT-based compressor does no division into chunks for compression; the whole file is handled in one piece. The following table shows space and runtime requirements for various algorithms. The reported values refer just to the reverse BWT step, not to complete decompression. The row "4 byte rank value" contains the results for the straightforward data structure, where the rank value is maintained as a 4 byte integer for each position. The other rows show results for the algorithms discussed.

Algorithm	Space [byte/input char]	Runtime [sec/Mbyte]
4 byte rank values	4	0.371
12 bit rank fields, 8192 fields per block	1.625	0.561
no rank fields, 1024 characters per block	1	3.477
no rank fields, 2048 characters per block	0.5	6.504

Table 1. Space and time requirement of the reverse BWT algorithms

We can see that the increase in run time when we avoid fully precalculated rank-values is moderate (about 50%). Here we remark that in our implementation the reverse BWT with 4 byte rank values takes about 60% of the of the total decompression time. Thus, the increase in total decompression time is about 30%. Even without any rank-fields and one block of counts for every 1024

input characters (resulting in 256 search steps on average for each rank computation) the increase in time for the reverse BWT is less than ten fold. In an embedded system where decompressed data are written to slow flash memory, writing to flash might dominate the decompression time.

To further consolidate results, we give run times for the three methods for the files of the Calgary Corpus [15], which is a standard test suite and collection of reference results for compression algorithms (see e.g., in [5]). As runtimes were too low to be reliably measured for some of the files, each file was run ten times. Table 2 summarizes the running times.

File	size [bytes]	$4n$ byte data structure	$\frac{13}{8}n$ byte data structure	n byte data structure
paper5	11954	0.175	0.351	2.632
paper4	13286	0.158	0.316	2.683
obj1	21504	0.195	0.341	2.536
paper6	38105	0.196	0.358	2.642
progc	39611	0.185	0.371	2.594
paper3	46526	0.180	0.361	2.682
progp	49379	0.191	0.361	2.718
paper1	53161	0.178	0.355	2.702
progl	71646	0.205	0.381	2.795
paper2	82199	0.255	0.344	2.768
trans	93695	0.201	0.392	2.652
geo	102400	0.287	0.410	2.601
bib	111261	0.283	0.415	2.790
obj2	246814	0.259	0.442	2.885
news	377109	0.350	0.551	3.128
pic	513216	0.259	0.323	3.735
book2	610856	0.388	0.580	3.459
book1	768771	0.430	0.649	3.796

Table 2. Runtime of the reverse BWT algorithms in [sec/Mbyte] with the files of the Calgary Corpus

Table 2 shows that the normalized running times increase with increasing file sizes. This effect can be explained as the result of caching. Since the order of indices in consecutive rank-queries can be an arbitrary permutation of $[0, ..., n-1]$, the number of page faults in the L1- and L2-caches becomes higher for bigger inputs.

5 Conclusions

We showed in this paper how the memory requirement of the reverse Burrows-Wheeler transformation can be reduced without decreasing the speed too much. This transformation is used e.g. in the well known program bzip2. Decreasing

the memory requirement for decompression may be essential in some embedded devices (e.g., mobile phones), where RAM is a scarce resource.

We showed that the reverse BWT can be done with 1.625 n bytes of auxiliary memory and $O(n)$ runtime. Alternatively, we can use n/t bytes and 256 t n operations. We also presented several time-space tradeoffs for the variants of our solution. These results are based on our new data structures for answering rank-queries and rank-position-queries. The theoretical results are backed up by experimental data showing that our algorithms work quite well in practice.

The question, if the space requirement of the data structures for rank-queries and rank-position-queries can be further reduced in our computational model, is still open. Improvements on the presented upper bounds have a practical impact. The problems of establishing lower bounds and improved time-space tradeoffs are open, as well.

References

1. J. Abel. Grundlagen des Burrows-Wheeler-Kompressionsalgorithmus (in german). *Informatik - Forschung und Entwicklung*, 2003. http://www.data-compression.info/JuergenAbel/Preprints/Preprint_Grundlagen_BWCA.pdf.
2. Z. Arnavut. Generalization of the BWT transformation and inversion ranks. In *Proc. IEEE Data Compression Conference (DCC '02)*, page 447, 2002.
3. M.J. Atallah, editor. *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.
4. B. Balkenhol and S. Kurtz. Universal data compression based on the Burrows-Wheeler transformation: Theory and practice. *IEEE Trans. on Computers*, 23(10):1043–1053, 2000.
5. T.C. Bell, J.G. Cleary, and I.H. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ, 1990.
6. M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. *Tech. report 124, Digital Equipment Corp.*, 1994. http://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html.
7. P. Fenwick. Block sorting text compression – final report. Technical report, Department of Computer Science, The University of Auckland, 1996. ftp://ftp.cs.auckland.ac.nz/pub/staff/peter-f/TechRep130.ps.
8. P. Ferragina and G. Manzini. Compression boosting in optimal linear time using the Burrows-Wheeler transform. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*, pages 655–663, 2004.
9. Tom's hadware guide. http://www.tomshardware.com/cpu/20001120/p4-01.html.
10. U. Manber and E. Meyers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22:935–948, 1993.
11. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
12. M. Nelson. Data compression with the Burrows-Wheeler transform. *Dr. Dobb's Journal*, 9, 1996.
13. J. Seward. Bzip2 manual, http://www.bzip.org/1.0.3/bzip2-manual-1.0.3.html.
14. J. Seward. Space-time tradeoffs in the inverse B-W transform. In *Proc. IEEE Data Compression Conference (DCC '01)*, pages 439–448, 2001.
15. I.H. Witten and T.C. Bell. The Calgary Text Compression Corpus. available via anonymous ftp at: ftp.cpcs.ucalgary.ca/pub/projects/text.compression.corpus.