Cache Friendly Burrows-Wheeler Inversion

Juha Kärkkäinen
Department of Computer Science
Gustaf Hällströmin katu 2 b
P.O. Box 68 FIN-00014
University of Helsinki
Helsinki, Finland
Email: juha.karkkainen@cs.helsinki.fi

Simon J. Puglisi
Department of Informatics
King's College London
The Strand
London WC2R 2LS
United Kingdom
Email: simon.puglisi@kcl.ac.uk

***Abstract*—The Burrows-Wheeler transform permutes the symbols of a string such that the permuted string can be compressed effectively with fast, simple techniques. Inversion of the transform is a bottleneck in practice. Inversion takes linear time, but, for each symbol decoded, folklore says that a random access into the transformed string (and so a CPU cache-miss) is necessary. In this paper we show how to mitigate cache misses and so speed inversion. Our main idea is to modify the standard inversion algorithm to detect and record repeated substrings in the original string as it is recovered. Subsequent occurrences of these repetitions are then copied in a cache friendly way from the already recovered portion of the string, short cutting a series of random accesses by the standard inversion algorithm. We show experimentally that this approach leads to faster runtimes in general, and can drastically reduce inversion time for highly repetitive data.**

***Keywords*-Burrows-Wheeler transform; BWT; suffix array; data compression; cache misses;**

I. Introduction

In the last decade the Burrows-Wheeler transform (BWT) [1] has been the focus of intense research, first as a tool for lossless data compression, and more recently as a tool for pattern matching and string processing. The transform alone provides no compression: it only permutes the symbols of a string in an invertible way. However, the transformed (permuted) string can be compressed effectively with fast, simple techniques [2]. The way to best exploit the structure of the transformed string for compression has been widely studied [3] and today compression performance offered by BWT compressors is state-of-the-art [4]. The focus of this paper is on the lightly studied inverse transform, which is the computational bottleneck at decompression time in practical BWT compression tools.

The basic algorithm for the inverse transform, already described in [1], is simple and runs in linear time, but its performance is hampered by an essentially random memory access pattern causing a large number of cache misses. Seward [5] gives an algorithm doing only n random accesses and conjectures that it "represents a realistic upper bound on inversion performance".

Our contribution: We introduce a new technique that can reduce the number of random accesses further by taking advantage of certain regularities that arise in the access pattern for typical texts. Our main idea is to modify the standard inversion algorithm to detect and record repeated substrings in the original string as it is recovered. Subsequent occurrences of these repetitions are then copied in a cache friendly way from the already inverted part of the original string, short-cutting a series of random accesses by the standard inversion algorithm. The resulting algorithm is faster than any previous algorithm – up to twice as fast for very repetitive texts.

Related work: Work on the inverse BWT has been sparse (at least relative to the work on the forward transform [6]). The known algorithms can be broadly classified by their space usage. The first study of the inverse transform by Seward [5], focussed mostly on fast, *large space* algorithms — requiring at least $n \log n$ bits. The algorithms we describe in this paper are also large space algorithms, and so we review Seward's work in more detail in Section III. At the other extreme, *small space* algorithms for BWT inversion are derived from the literature on compressed full-text indexes [7]. These algorithms require at most $n \log \sigma + o(n \log \sigma)$ bits, and while their practical performance continues to be improved [8], small space algorithms are typically much slower than their large space counterparts. Recently, Kärkkäinen and Puglisi [9] explored *medium space* algorithms, which lie in between small and large space algorithms. They show that a variety of different approaches lead to a smooth space-time trade-off between the two extremes.

Road Map: The paper proceeds as follows. In Section II we set notation and revisit basic concepts required to understand BWT inversion. Section III reviews the large-space algorithms of Seward [5] and provides some details of our modern implementations of his work. Section IV describes our new cache-friendly inversion algorithm, which we call `copy`. Section V details our experiments and the

This work was supported by Academy of Finland grant 118653 (ALGODAN) and by the Australian Research Council. Simon J. Puglisi is supported by a Newton Fellowship.

F								L
\$	K	A	L	A	L	A	V	A
A	\$	K	A	L	A	L	A	V
A	L	A	L	A	V	A	\$	K
A	L	A	V	A	\$	K	A	L
A	V	A	\$	K	A	L	A	L
K	A	L	A	L	A	V	A	\$
L	A	V	A	\$	K	A	L	A
L	A	L	A	V	A	\$	K	A
V	A	\$	K	A	L	A	L	A

Figure 1. Matrix M for text $S =$ KALALAVA\$

practical performance of the large-space algorithms, and is followed by some possible directions for future research.

II. Preliminaries

Let $S = S[0..n] = S[0]S[1]\dots S[n]$ be a string (or text) of $n+1$ symbols or characters. The first n symbols of S are drawn from an ordered alphabet, Σ, of $|\Sigma| = \sigma$ distinct symbols. The final character $S[n]$ is a special "end of string" character, \$, distinct from and lexicographically smaller than all the other characters in Σ.

For any $i \in 0..n$, the string $S[i..n]S[0..i-1]$ is a ***rotation*** of S. Let M be the $(n+1) \times (n+1)$ matrix whose rows are all the rotations of S in lexicographic order. Let F be the first and L the last column of M, both taken to be strings of length $n+1$. The string L is the ***Burrows–Wheeler transform*** of S. An example is given in Figure 1. Note that F and L (and all other columns in M) are permutations of S.

Inverse Transform.: We will not describe here, how L can be computed from S (the forward trasform), or how L can be compressed. We will also give the algorithms for the inverse transform, i.e., the reconstruction of S from L, without a proof or explanation — these can be found in any of the large number of descriptions of the Burrows–Wheeler transform.

For any string X, define the following functions:

$$\begin{aligned}
\text{char}_X(j) &= X[j] \\
\text{rank}^c_X(j) &= |\{i \mid i < j \text{ and } X[i] = c\}| \\
\text{rank}_X(j) &= \text{rank}^{X[j]}_X(j) \\
\text{select}_X(c,r) &= j : \text{char}_X(j) = c \text{ and } \text{rank}_X(j) = r
\end{aligned}$$

The notation char_X is used instead of $X[j]$ when X might be stored in a compressed form that does not support trivial character access.

Two abstract inversion algorithms are given in Figure 2. The algorithm on the left-hand side reconstructs S from first symbol to last symbol, and the one on the right reconstructs S in reverse order. It turns out that the operations used in the second (reverse order) algorithm are easier and faster to implement and all the algorithms that follow are variations of it. Note that the reverse order algorithm will output the original text left-to-right provided the original text is reversed prior to applying the forward BWT — an approach often adopted in practice.

Compact representation of F.: The string F contains the characters of S in sorted order and all copies of the same symbol are grouped together allowing a simple and very compact representation of F. For any symbol c, let $\text{start}_F(c)$ be the position of the first occurrence of c in F. The function start_F is clearly a full representation of F. In practice, operation start_F can be implemented as a lookup table of size $\sigma+1$. It can be easily computed by scanning L to count the number of occurrences of each symbol and then computing the cumulative sums of the occurrence counts.

Given this representation, select_F is easy to implement:

$$\text{select}_F(c,r) = \text{start}_F(c) + r.$$

char_F and rank_F can be implemented by a binary search in time $\mathcal{O}(\log \sigma)$. All the algorithms that follow use this representation of F.

III. Four Easy Pieces

Figure 3 shows four simple algorithms described by Seward [5]. All except `indexF` need $5n$ bytes of space and run in linear time.

Algorithm `bw94`: is from the seminal paper by Burrows and Wheeler [1]. Let us consider the abstract algorithm on the right in Figure 2. Three operations are repeatedly called in the main loop. We have already seen how to implement select_F, and $\text{char}_L[j]$ has the trivial implementation as $L[j]$. This leaves rank_L.

Computing the value of $\text{rank}_L(j)$ based on L alone takes $\mathcal{O}(j)$ time. However, all the values $\text{rank}_L(j)$, $j = 0..n$, can be easily computed in $\mathcal{O}(n)$ time in *sequential order* by scanning L while keeping account of the number of occurrence of each symbol. Algorithm `bw94` takes advantage of this by computing the rank-values in advance and storing them in an array R. The details can be found, for example, in [5].

Algorithm `basis`: is the similar to `bw94` but moves some of the work from the main loop to the preprocessing loop. This turns out to be a little faster in practice in our experiments while Seward's results were the opposite.

Algorithms `mergedRL` and `mergedTL`: are variations of `bw94` and `basis`, respectively. They are the same algorithms except the array L is merged with the other main array, R or T, to form a single array, RL or TL. The rationale for this is that, in the main loop, the sequence of accesses to L, R or T is essentially random with a high likelihood of a cache miss for each access. Merging two arrays can reduce the number of cache misses to a half, leading to a significant improvement in speed.

In his implementation[1], Seward places the restriction $n < 2^{24}$ which allows $R[j]$ and $L[j]$ to be packed into a single 32-

[1]Seward calls his algorithm `mergedTL` but it is `mergedRL` using our notation. Our implementation is a proper `mergedTL`.

```
1: construct F from L
2: j ← select_L($, 0)
3: for i ← 0 to n do
4:       S[i] ← c ← char_F(j)
5:       r ← rank_F(j)
6:       j ← select_L(c, r)
```

```
1: construct F from L
2: j ← select_L($, 0)
3: for i ← n downto 0 do
4:       S[i] ← c ← char_L(j)
5:       r ← rank_L(j)
6:       j ← select_F(c, r)
```

Figure 2. Two abstract algorithms for the inverse Burrows–Wheeler transform, reproduced from [9].

Algorithm `bw94`

```
1: Compute start_F from L
2: for j ← 0 to n do
3:       R[j] ← rank_L(j)
4: j ← select_L($, 0)
5: for i ← n downto 0 do
6:       S[i] ← c ← L[j]
7:       r ← R[j]
8:       j ← select_F(c, r)
```

Algorithm `basis`

```
1: Compute start_F from L
2: for j ← 0 to n do
3:       T[j] ← select_F(L[j], rank_L(j))
4: j ← select_L($, 0)
5: for i ← n downto 0 do
6:       S[i] ← L[j]
7:       j ← T[j]
```

Algorithm `mergeRL`

```
1: Compute start_F from L
2: for j ← 0 to n do
3:       RL[j] ← ⟨rank_L(j), L[j]⟩
4: j ← select_L($, 0)
5: for i ← n downto 0 do
6:       ⟨r, c⟩ ← RL[j]
7:       S[i] ← c
8:       j ← select_F(c, r)
```

Algorithm `indexF`

```
1: Compute start_F from L
2: for j ← 0 to n do
3:       T[j] ← select_F(L[j], rank_L(j))
4: j ← select_F($, 0)
5: for i ← n downto 0 do
6:       S[i] ← char_F(j)
7:       j ← T[j]
```

Figure 3. Four algorithms by Seward [5]

bit word. Because we are interested in larger n in this paper, we adopt the following variant of `mergedTL`. Allocate an array U of $n + \lceil n/4 \rceil$ words. $U[j]$ for $j \neq 4 \pmod 5$ stores a $T[j]$ value, while $U[j]$ for $j = 4 \pmod 5$ holds four symbols (bytes) from L. Although $L[j]$ and $T[j]$ are not directly adjacent in this arrangement, $L[j]$ is never more than 12 bytes (three words) away from $T[j]$: close enough to be in cache when it is required.

Algorithm `indexF`: is similar to `basis` but it replaces char_L in the main loop with char_F (which requires moving the starting point from $\text{select}_L(\$, 0)$ to $\text{select}_F(\$, 0)$). The time complexity becomes $\mathcal{O}(n \log \sigma)$ since char_F is implemented by a binary search over start_F, but it is fast in practice as start_F fits in the cache. The advantage is that L is no more needed in the main loop. Thus the space requirement can be reduced to $4n$ bytes.

IV. Faster Inversion

The fastest of the algorithms we have seen so far is `mergedTL`. Experiments by Seward [5] and by us indicate that its running time is dominated by the cache misses caused by the accesses to the array TL. Since the accesses are essentially random, each access is likely to cause a cache miss, and there does not seem to be a way to avoid at least one cache miss on most rounds of the main loop. Indeed, Seward argues that "the maximum speed of BWT decompression is unavoidably constrained by the rate at which cache misses are serviced" [5]. In this section, we show that there is a way to reduce the number of cache misses.

Let j be a position in L such that $L[j] = L[j+1]$. By definition of rank, we have that $\text{rank}_L(j+1) = \text{rank}_L(j) + 1$. Let $\text{next}(j)$ denote the position accessed after j in the main loop of the basic algorithms. Then $\text{next}(j) = \text{select}_F(L[j], \text{rank}_L(j)) = \text{start}_F(L[j]) + \text{rank}_L(j)$. Therefore, $\text{next}(j+1) = \text{next}(j) + 1$. Furthermore, if $L[\text{next}(j)] = L[\text{next}(j+1)]$, we will have that $\text{next}^2(j+1) = \text{next}^2(j) + 1$, and so on. In other words, two access paths sometimes stay together for a while; we call this a *chain*. The traversal of the two paths of a chain will reconstruct the same substring into two different places in the text. Thus a chain corresponds to a repetition in the text. Highly repetitive texts have many and/or long chains.

Our new algorithm `copy` is a modification of `mergedTL` and does what `mergedTL` does most of the time. However, it will watch for chains by comparing $L[j]$ and $L[j+1]$ at each step. When the algorithm finds a chain longer than one, say starting from j and $j+1$ and ending at k and $k+1$, it

Table I
DATA SETS USED FOR EMPIRICAL TESTS. FOR EACH TYPE OF DATA (DNA, XML, ENGLISH, PROT, SOURCE) A 50MB, 100MB AND 200MB FILE WAS USED. THE STATISTICS ARE FOR THE 50MB FILE, BUT ARE INDICATIVE FOR THE LARGER FILES TOO.

Data set name	Sizes (Mb)	$\|\Sigma\|$	H_0	mean LCP
XML	50,100,200	97	5.23	44
DNA	50,100,200	4	1.98	31
ENGLISH	50,100,200	239	4.53	2,221
SOURCE	50,100,200	230	5.54	168
PROT	50,100,200	27	4.20	166

performs the following procedure to store the chain. Let i be the text position to where $L[j]$ was copied and let ℓ be the length of the chain. The values k, i and ℓ are written to $T[j]$, $T[j+1]$ and $T[k]$, respectively, (the original values are not needed any more), and the position $j+1$ is marked. When the algorithm later arrives to position $j+1$ and finds it marked, the algorithm recovers k, i and ℓ by accessing $T[j]$, $T[j+1]$ and $T[k]$, and jumps directly to $k+1$ thus avoiding the random accesses that normally would occur between $j+1$ and $k+1$. The corresponding symbols are found in $S[i..i+\ell-1]$ and copied to the current text position.

`copy` runs in linear time and needs $6n$ bytes of space; the additional n bytes is needed for the text S, which needs to be kept in memory due to the copying.

As described, `copy` only follows one chain at a time. If a triple (or longer run) of symbols appears in the BWT, there is an opportunity to follow three (or more) chains simultaneously. Intuitively, following multiple chains simultaneously seems likely to be beneficial, but makes implementation more difficult. We will explore this idea further in the full paper.

V. EXPERIMENTAL RESULTS

For testing we used the files listed in Table I [2]. All tests were conducted on a 3.0 GHz Intel Xeon CPU with 4Gb main memory and 1024K L2 Cache. The machine had no other significant CPU tasks running. The operating system was Fedora Linux running kernel 2.6.9. The compiler was g++ (gcc version 3.4.4) executed with the -O3 option. Times given are the minima of three runs and were recorded with the standard C `getrusage` function.

Experiments measured the time to invert the BWT. Following previous experimental methodology [5], [9] we did not count the time to read the transformed text in from file and the inverted text was written to an array, held in memory. The algorithms and their space requirements are summarized in Table II. The runtimes are shown in Table III.

To avoid clutter we measured only large space inversion algorithms. However, the implementation of `mtl` is identical to that used in [9], allowing a comparison to the more space-efficient approaches of that paper to be made.

[2]Available from http://pizzachili.dcc.uchile.cl/.

Table II
ALGORITHMS AND THEIR SPACE REQUIREMENTS. THE SPACE REQUIREMENTS DO NOT INCLUDE THE OUTPUT ARRAY, EXCEPT FOR THE `copy` ALGORITHM, WHICH MAKES USE OF IT DURING INVERSION. SPACE FOR `wtree` METHOD INCLUDES $n \log \sigma$ WORKING SPACE USED TO HOLD L DURING TREE CONSTRUCTION.

Alg.	Space	Description
`basis`	$5n$ bytes	Seward's basic inversion algorithm [5]
`bw94`	$5n$ bytes	Original algorithm from [1]
`mtl`	$5n$ bytes	Large n version of `mergedTL` in [5]
`indexF`	$4n$ bytes	Large n version of `indexF` in [5]
`copy`	$6n$ bytes	Detect and copy repetitions in the output string

Table III
RUNTIMES (IN SECONDS) FOR THE VARIOUS INVERSION ALGORITHMS.

Dataset	basis	bw94	mtl	indexF	copy
XML-50	9.42	10.49	8.61	9.78	6.94
XML-100	21.34	23.02	18.59	21.43	14.84
XML-200	47.01	49.63	39.24	46.32	31.08
DNA-50	10.80	11.83	9.91	9.67	9.66
DNA-100	25.52	26.30	22.30	22.44	21.92
DNA-200	57.46	58.95	48.48	48.82	46.92
ENGLISH-50	10.55	11.50	9.60	11.22	7.43
ENGLISH-100	24.42	25.67	21.43	24.83	17.67
ENGLISH-200	55.11	57.43	46.24	53.64	38.43
PROT-50	9.78	10.65	8.78	9.59	8.40
PROT-100	23.58	24.74	20.65	22.63	18.87
PROT-200	51.19	54.69	42.96	46.78	38.37
SOURCE-50	7.50	8.41	6.67	8.99	6.07
SOURCE-100	16.57	18.29	14.43	19.10	12.85
SOURCE-200	37.23	40.20	31.44	41.03	27.84

VI. DISCUSSION

`copy` is consistently the fastest algorithm, indicating that the extra complexity of identifying chains pays off. To test the idea that `copy` benefits from repetitions in the text, we created a highly repetitive 100Mb file by concatenating two copies of ENGLISH-50. `mtl` required 24.72 seconds, while `copy` took just 13.58 seconds: the entire second half of the string being copied from the first half.

We found `indexF` to be substantially more competitive than Seward did in his experiments. Particularly on small alphabets (DNA, PROT), the binary searches over the symbol count array are relatively cheap, and `indexF` approaches (and once beats) the speed of `mtl`, while using 1 byte per symbol less.

VII. FUTURE DIRECTIONS

As already mentioned in Section IV, following multiple chains (corresponding to runs of the same symbol in L) at the same time would seem beneficial, and we are currently pursuing such an implementation.

We believe that there are further possibilities to improve BWT inversion algorithms, through new techniques, through improvement of the techniques described here, but particularly by combining different techniques in the same

algorithm, and we will continue our investigations in this direction.

An interesting avenue for future work is parallel inversion. The idea is to reconstruct the text S at several places simultaneously. The parallelism can be explicit, with threads, enabling BWT inversion algorithms to benefit from modern, multicore CPU architectures; or implicit, by programming to the out-of-order execution of the CPU, which has proved successful for string sorting and LCP array construction [10], [11].

Finally, we note that the manner in which algorithm `copy` copies substrings from the already decoded parts of the string to form new output is reminiscent of the famous LZ77 dictionary compression algorithm. It would be interesting if this apparent relationship between BWT and LZ77 could be formalized — doing so may reveal a deep connection between these two fundamental compression and string processing tools.

REFERENCES

[1] M. Burrows and D. J. Wheeler, "A block sorting lossless data compression algorithm," Digital Equipment Corporation, Palo Alto, California, Tech. Rep. 124, 1994.

[2] G. Manzini, "An analysis of the Burrows-Wheeler transform," *Journal of the ACM*, vol. 48, no. 3, pp. 407–430, May 2001.

[3] D. Adjeroh, T. Bell, and A. Mukherjee, *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, July 2008.

[4] P. Ferragina and G. Manzini, "On compressing the textual web," in *Proc. 3rd ACM International Conference on Web Search and Data Mining (WSDM '10)*. New York, NY, USA: ACM, 2010, pp. 391–400.

[5] J. Seward, "Space-time tradeoffs in the inverse B-W transform," in *Proc. IEEE Data Compression Conference*, J. Storer and M. Cohn, Eds. Los Alamitos: IEEE Computer Society, 2001, pp. 439–448.

[6] S. J. Puglisi, W. F. Smyth, and A. Turpin, "A taxonomy of suffix array construction algorithms," *ACM Computing Surveys*, vol. 39, no. 2, pp. 1–31, 2007.

[7] G. Navarro and V. Mäkinen, "Compressed full-text indexes," *ACM Computing Surveys*, vol. 39, no. 1, 2007.

[8] J. Kärkkäinen and S. J. Puglisi, "Fixed block compression boosting in FM-indexes," 2011. [Online]. Available: http://arxiv.org/abs/1104.3810

[9] ——, "Medium-space algorithms for inverse BWT," in *Proc. 18th European Symposium on Algorithms (ESA '10)*, ser. LNCS, M. de Berg and U. Meyer, Eds., vol. 6346. Berlin Heidelberg: Springer-Verlag, 2010, pp. 451–462.

[10] J. Kärkkäinen and T. Rantala, "Engineering radix sort for strings," in *Proc. 15th Symposium on String Processing and Information Retrieval (SPIRE '08)*, ser. LNCS, A. Amir, A. Moffat, and A. Turpin, Eds., vol. 5280. Berlin: Springer-Verlag, 2008, pp. 3–14.

[11] J. Kärkkäinen, G. Manzini, and S. J. Puglisi, "Permuted longest-common-prefix array," in *Proc. 20th Symposium on Combinatorial Pattern Matching (CPM '09)*, ser. LNCS, G. Kucherov and E. Ukkonen, Eds., vol. 5577. Berlin Heidelberg: Springer-Verlag, 2009, pp. 181–192.