Slashing the Time for BWT Inversion

Juha Kärkkäinen[1,*] Dominik Kempa[1,*] Simon J. Puglisi[2,†]

[1] Department of Computer Science, University of Helsinki, Finland
{juha.karkkainen,dominik.kempa}@cs.helsinki.fi

[2] Department of Informatics, King's College London, London, United Kingdom
simon.puglisi@kcl.ac.uk

Abstract

Inverting the Burrows-Wheeler transform (BWT) is a bottleneck in BWT-based decompressors. The state-of-the-art inversion algorithm runs in linear time but is slow in practice due to CPU-cache misses. For more than a decade these cache misses have been thought to be inherent to BWT inversion. We show how to reduce the number of cache misses by a factor of nearly two, and simultaneously the cost of cache misses by another factor of two, obtaining a consistent speed up by a factor of 2.3–4. We can do even better if the data is highly repetitive. We describe an algorithm that achieves an asymptotic reduction in cache misses in theory and is the fastest algorithm in practice for such data.

1 Introduction

The Burrows-Wheeler transform (BWT) [2] is a lossless, invertible transform of a string which makes the string easy to compress using textbook techniques [11]. Since its discovery in 1994 the BWT has been the subject of heavy research both for compression, and in the field of pattern matching, where it has been deeply linked with the suffix array [10], to produce fast compressed indexes [12].

Early work on the BWT focussed on how to best compress the transformed string (see [1]) and this work has matured to the point that BWT-based compressors are now state-of-the-art [4]. Another heavily researched aspect is how to perform the transform itself, also known as the suffix sorting problem [13, 6, 3].

An aspect of the BWT that has received relatively little attention is the efficient *inversion* of the transformed string, which is the final step in any BWT-based decompressor. The basic inversion algorithm [2] is simple (roughly 10 neat lines of C code) and works in linear time, but in practice is slow due to a non-local memory access pattern which causes at least one CPU-cache miss for each letter decoded. This

[*]Supported by Academy of Finland grant 118653 (ALGODAN).
[†]Supported by Newton Fellowship.

property of the basic algorithm was first elucidated by Seward [14], and in the same paper he gave an algorithm requiring at most n cache misses, which he conjectured "represent[ed] a realistic upper bound on inversion performance". For more than a decade these CPU-cache misses have been thought to be inherent to BWT inversion.

Our contribution. In this paper we describe ways to avoid cache misses and speed inversion. Our first technique is based on a super-alphabet, effectively decoding two characters at a time. A detailed theoretical analysis proves that it reduces the number of cache misses by nearly a factor of two, and extensive experiments show a corresponding speed up in practice. An improvement of similar size is obtained by starting the inversion simultaneously at multiple positions. This technique does not reduce the number of cache misses but does reduce the *cost* of cache misses by taking advantage of the out-of-order execution capabilities of modern CPUs. The combination of these techniques gives an algorithm that is consistently 2.3–4 times faster than Seward's algorithm in practice.

Our second technique takes advantage of repeated factors in the original string. Such repetitions manifest as *runs* of equal characters *in the inverted string* [15]. We recover repetition information from the runs and then, during inversion, we copy repeated factors in a cache-friendly way. For highly repetitive strings — those with a sublinear number of runs — the algorithm achieves an asymptotic reduction in cache misses, and can reach optimality in the cache-oblivious model. Experiments show that the algorithm is the fastest in practice for highly repetitive data.

Related work. Two previous works study fast inversion of the BWT. The first, due to Seward [14], highlights the presence of cache-misses in the original inversion algorithm of [2]. Seward describes a simple trick to reduce these cache-misses by half, but argues that any further reduction is unlikely. Recently, Kärkkäinen and Puglisi [8] did manage to reduce cache-misses further by a factor of up to two *for repetitive strings*. Their *copy* algorithm is the inspiration for some of the techniques described in this paper. Both [14] and [8], as well as this paper, study "large-space" algorithms, which invert fast, but use at least $n \log n$ bits of working memory. The literature on compressed indexing (see [12, 9]) contains several techniques for inverting in "small-space", at the cost of runtime. Finally, Kärkkäinen and Puglisi [7], describe several "medium-space" inversion algorithms which occupy the area of the space-time spectrum inbetween the two extremes.

2 Preliminaries

A *string* $\mathrm{T}[0..n-1] = \mathrm{T}[0]\mathrm{T}[1]\ldots\mathrm{T}[n-1]$ is a finite sequence of *characters* from *alphabet* $\Sigma = \{0, 1, \ldots, \sigma - 1\}$. We assume that $\sigma = \mathcal{O}(n)$ and use σ_T to denote the number of distinct characters occurring in T. We assume that $\mathrm{T}[n-1] = 0$ and 0 does not appear elsewhere in T. We use `$` to denote 0 and letters for other symbols. A *prefix* of T is a string of the form $\mathrm{T}[0..i]$, for $i \in 0..n-1$. Analogously, $\mathrm{T}[i..n-1]$ is called a *suffix* of T. By T^R we denote the *reverse* of string T, i.e., $\mathrm{T}^R[i] = \mathrm{T}[n-1-i]$ for $i \in 0..n-1$.

We define the *rotation* of T as a string $\mathrm{T}^{(i)} = \mathrm{T}[i..n-1]\mathrm{T}[0..i-1]$, for $i \in 0..n-1$.

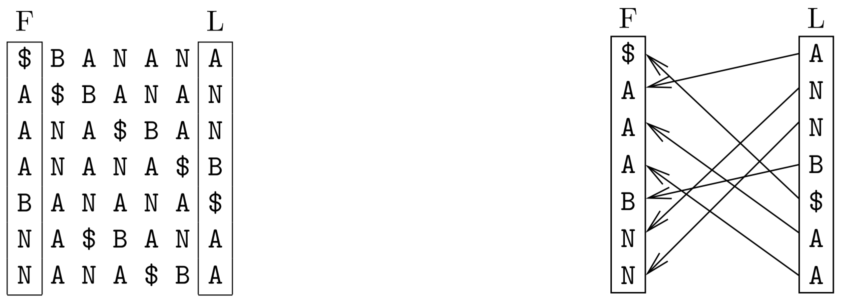

Figure 1: BWT matrix $\mathcal{M}$ and LF mapping for T = `BANANA$`

For $i < 0$ or $i \geq n$ we set $\mathrm{T}^{(i)} = \mathrm{T}^{(i \bmod n)}$. Let $\mathcal{M}$ be the $n \times n$ matrix, whose rows are all the rotations of T in lexicographical order. We denote the rows by $\mathcal{M}[i]$, $i \in 0..n-1$. Let F and L be the strings formed by characters from first and (respectively) last column of $\mathcal{M}$. The string L is the *Burrows-Wheeler transform* of T (BWT). We define the *LF mapping* $\mathrm{LF}[0..n-1]$ as the permutation satisfying $\mathcal{M}[\mathrm{LF}[i]] = \mathcal{M}[i]^{(-1)}$ for $i \in 0..n-1$. An example is shown in Figure 1.

BWT inversion, i.e., reconstructing T from L, is based on the following result.

Theorem 1 ([**2**]). *For each* $i \in 0..n-1$,

- $\mathrm{LF}[i] = \mathrm{C}[\mathrm{L}[i]] + rank_L(i)$, *where* $\mathrm{C}[k]$ *is the number of characters* $\{0, \ldots, k-1\}$ *occurring in* L *and* $rank_L(i)$ *is the number of times* $\mathrm{L}[i]$ *occurs in* $\mathrm{L}[0..i-1]$
- $\mathrm{T}^R[i] = \mathrm{L}[\mathrm{LF}^i[p_0]]$, *where* $\mathrm{L}[p_0] = \$$.

Cache Oblivious Model. We analyze the cache complexity of our algorithms under the Cache Oblivious model [5], where we have a fully associative cache of M words, cache lines of B words and an optimal replacement policy. The parameters M and B are not known to the algorithms. The analysis is based on identifying the following kinds of memory access sequences:

Sequential accesses: A sequence of accesses to k consecutive memory words in a sequential order causes at most $1 + \lceil (k-1)/B \rceil$ cache misses and consumes one cache line.

Small data structure accesses: A sequence of accesses to a data structure of size $K < M$ words causes at most $1 + \lceil (K-1)/B \rceil$ cache misses and consumes at most $1 + \lceil (K-1)/B \rceil$ cache lines.

Multiple access sequences may be interleaved as long as the total number of cache lines consumed by them at any point in time is less than M/B.

3 Constant Factor Speedup

In this section, we take Seward's fastest algorithm MTL (`mergedTL` in [14]) as a starting point. It is based directly on Theorem 1 and is shown in Figure 2. We first

```
MTL(L)
1: LF ← COMPUTELF(L)
2: $p$ ← locate(L, \$); $l \leftarrow 0$
3: while $l < n$ do
4:     $T^R[l] \leftarrow L[p]$
5:     $p \leftarrow LF[p]$; $l \leftarrow l+1$
6: return $T^R$

COMPUTECOUNTS(L)
1: for $i \leftarrow 0$ to $\sigma$ do
2:     $C[i] \leftarrow 0$
3: for $i \leftarrow 0$ to $n-1$ do
4:     $C[L[i]+1] \leftarrow C[L[i]+1]+1$
5: for $i \leftarrow 1$ to $\sigma-1$ do
6:     $C[i] \leftarrow C[i]+C[i-1]$
7: return C

COMPUTELF(L)
1: C ← COMPUTECOUNTS(L)
2: for $i \leftarrow 0$ to $n-1$ do
3:     $LF[i] \leftarrow C[L[i]]$
4:     $C[L[i]] \leftarrow C[L[i]]+1$
5: return LF
```

```
MTL-SA(L)
1: LF ← COMPUTELF(L)
2: LL ← COMPUTELL(L, LF)
3: $LF^2$ ← COMPUTELF$^2$(LF)
4: $p$ ← locate(L, \$); $l \leftarrow 0$
5: while $l+1 < n$ do
6:     $T^R[l..l+1] \leftarrow LL[2p..2p+1]$
7:     $p \leftarrow LF^2[p]$; $l \leftarrow l+2$
8: if $l = n-1$ then
9:     $T^R[l] \leftarrow LL[2p]$
10: return $T^R$

COMPUTELL(L, LF)
1: for $i \leftarrow 0$ to $n-1$ do
2:     $LL[2i] \leftarrow L[i]$
3:     $LL[2i+1] \leftarrow L[LF[i]]$
4: return LL

COMPUTELF$^2$(LF)
1: for $i \leftarrow 0$ to $n-1$ do
2:     $LF^2[i] \leftarrow LF[LF[i]]$
3: return $LF^2$
```

Figure 2: Basic and super-alphabet BWT inversion.

show how to nearly half the number of cache misses, and then how to reduce the cost of cache misses by another factor of about two.

Reducing number of cache misses. To half the number of cache misses, we employ a technique called *super-alphabet*: We process two characters at a time in the main loop. We precompute an array LL of size $2n$, where $LL[2i] = L[i]$ and $LL[2i+1] = L[LF[i]]$. Note that $L[LF[i]]$ is the successor of $L[i]$ in T^R. We also need the array LF^2, where $LF^2[i] = LF[LF[i]]$. The algorithm, which we call MTL-SA, is shown in Figure 2.

A key detail omitted in the pseudocode of MTL is that the arrays L and LF are stored interleaved so that $L[p]$ and $LF[p]$ are next to each other and the accesses to them on lines 4 and 5 cause only one cache miss instead of two. Similar interleaving is applied to the arrays LL and LF^2 in MTL-SA.

Theorem 2. *Algorithm* MTL *runs in* $\mathcal{O}(n)$ *time and causes at most* $n + \mathcal{O}(\frac{n}{B})$ *cache misses provided that* $\sigma + \mathcal{O}(B) \leq M$. *Algorithm* MTL-SA *runs in* $\mathcal{O}(n)$ *time and causes at most* $\lceil n/2 \rceil + \mathcal{O}(\sigma_T + \frac{n}{B})$ *cache misses provided that* $\max(\sigma, \sigma_T B) + \mathcal{O}(B) \leq M$.

Proof. The linear time complexity of both algorithms is trivial.

COMPUTECOUNTS and COMPUTELF make sequential accesses to L and LF causing $\mathcal{O}(n/B)$ cache misses, and non-sequential accesses to C causing $\mathcal{O}(\sigma/B)$ cache misses provided that $\sigma \leq M - \mathcal{O}(B)$.

The same analysis applies to ComputeLL and ComputeLF^2 except for the accesses that use LF[i] as an address. It is easy to see from ComputeLF that LF contains σ_T interleaved increasing sequences. Thus the number of cache misses caused by using LF[i] as an address is $\mathcal{O}(\sigma_T + n/B)$ provided that the cache is large enough. The number of cache lines consumed is $\sigma_T + \mathcal{O}(1)$.

In the main algorithms, the only non-sequential accesses are those using p as an address. Due to the interleaving of the arrays, they cause at most n cache misses in Mtl and at most $\lceil n/2 \rceil$ cache misses in Mtl-sa. $\square$

Reducing cost of cache misses. For a further speedup, we divide the text into p parts: $\mathrm{T}[0 \ldots \frac{n}{p} - 1]$, $\mathrm{T}[\frac{n}{p} \ldots \frac{2n}{p} - 1]$, $\ldots$, $\mathrm{T}[\frac{(p-1)n}{p} \ldots n - 1]$ and reconstruct each part separately.[1] The reconstructions are independent of each other and could be performed in parallel. However, we use just one thread of execution that switches between the parts and still obtain a speedup by a factor of about two using $p = 8$. The explanation is *out-of-order execution*: While one instruction has to wait for a cache miss, the CPU can execute subsequent instructions that are independent of the waiting instruction.

Reducing space. Our implementation of ComputeLL and ComputeLF^2 differs from the one given in Figure 2. The purpose is to reduce space by avoiding the storage of the LF array. ComputeLL computes LF[i] on demand the same way it is done in ComputeLF. ComputeLF^2 is the same as ComputeLF but with L replaced by LL and C replaced by CC, which stores the counts for pairs of characters. This increases the preprocessing cache complexity to $\mathcal{O}(\sigma_T + (n + \sigma^2)/B)$ cache misses using $\max\{\sigma_T + \sigma/B, \sigma^2/B\} + \mathcal{O}(1)$ cache lines.

4 Asymptotic Speedup for Repetitive Data

In this section, we describe an algorithm called Precopy that takes advantage of repetitions in the data.

A maximal sequence of consecutive equal characters in L is called a *run*. It is well known that repetitions in T manifest as runs in L [15]. Kärkkäinen and Puglisi [8] used runs to speed up BWT inversion based on the following property.

Lemma 3 ([8]). *For any* $i \in 1..n-1$ *such that* $\mathrm{L}[i] = \mathrm{L}[i-1]$, $\mathrm{LF}[i] = \mathrm{LF}[i-1] + 1$.

Here we generalize the concept of runs by defining an *s-run* as a maximal range $b..e$ such that all rows in $\mathcal{M}[b..e]$ have a common suffix of length s. Thus runs in L are 1-runs by our definition. An s-run of length k means that the same string of length s occurs at least k times in T. Precopy takes advantage of this by copying an earlier occurrence when possible and thus avoiding the cache misses of normal decoding.

[1]We assume that the L positions corresponding to the end positions of the parts are stored in the compressed file.

Computing s-runs. The *lcs array* $\mathrm{LCS} = \mathrm{LCS}[0..n-1]$ is an array defined by $\mathcal{M}$. Let $\mathrm{lcs}(x, y)$ denote the length of the longest common suffix of strings x and y. For every $j \in 1..n-1$,

$$\mathrm{LCS}[j] = \mathrm{lcs}(\mathcal{M}[j-1], \mathcal{M}[j]).$$

We define $\mathrm{LCS}[0] = \mathrm{LCS}[n] = 0$ so we can concisely characterize maximal s-runs in terms of the LCS array.

Observation 4. *The range $b..e$ is an s-run if and only if $\mathrm{LCS}[b] < s$, $\mathrm{LCS}[e+1] < s$ and for every $k \in b+1..e$, $\mathrm{LCS}[k] \geq s$.*

Define $Q_s = \{i \in 0..n-1 \mid \mathrm{LCS}[i] = s\}$ and $R_s = \cup_{j \in 0..s-1} Q_j$. Then R_s is exactly the set of starting positions of s-runs.

A key to fast computation of s-runs is the following property, which is a direct consequence of the definitions of the LCS array and the LF mapping, and Lemma 3.

Lemma 5. *For any $i \in 0..n-1$,*

$$\mathrm{LCS}[i] = \begin{cases} 0 & \text{if } i = 0 \text{ or } \mathrm{L}[i] \neq \mathrm{L}[i-1] \\ \mathrm{LCS}[\mathrm{LF}[i]] + 1 & \text{otherwise} \end{cases}$$

Thus the set Q_s can be efficiently computed, given the values of Q_{s-1}, as presented below (the proof can be derived from Lemma 5).

Corollary 6. *For any $s > 0$, $Q_s = \{\mathrm{LF}^{-1}[i] \mid i \in Q_{s-1} \text{ and } \mathrm{LF}^{-1}[i] \notin Q_0\}$.*

Let $r_s = |R_s|$ be the number of s-runs and let $r = r_1 = |Q_0|$. From the above we see that $|Q_{i+1}| \leq |Q_i|$, which implies $r_s \leq |Q_0|s = rs$.

Theorem 7. *The set R_s can be computed in $\mathcal{O}(n)$ time causing $\mathcal{O}(r_s + \frac{n}{B})$ cache misses provided that $\sigma + \sigma_T B + \mathcal{O}(B) \leq M$.*

Proof. The values of LF^{-1} can be computed by changing the line 3 of ComputeLF to $\mathrm{LF}^{-1}[\mathrm{C}[\mathrm{L}[i]]] \leftarrow i$. An analysis similar to Theorem 2 shows that the cache complexity is $\mathcal{O}(\sigma_T + (n+\sigma)/B)$ cache misses using $\sigma/B + \sigma_T + \mathcal{O}(1)$ cache lines. Note that $\sigma_T \leq r \leq r_s$. The set Q_0 can be computed by one pass over the input causing $\mathcal{O}(n/B)$ cache misses. Computing Q_j for $j > 0$ takes $\mathcal{O}(|Q_{j-1}|)$ time and cache misses. The set R_s, represented as a bit vector of size n, can be computed in $\mathcal{O}(n)$ time causing $\mathcal{O}(n/B + r_s)$ cache misses. □

Inverting BWT. The Precopy algorithm for inverting BWT is listed in Figure 3. Like the other algorithms, it maintains the invariant $p = \mathrm{LF}^l[p_0]$, where $\mathrm{L}[p_0] = \$$, but additionally it stores the value $\mathrm{LF}^l[p_0]$ into $\mathrm{pow}_{\mathrm{LF}}[l]$. When it visits an s-run for the first time (lines 14–17), it operates as Mtl but additionally records the output position l in an array LT_s. On a later visit to the same s-run (lines 18–26), it copies the next s characters $\mathrm{T}^R[l..l+s-1]$ from the position recorded at the first visit. In addition, the algorithm needs to fill $\mathrm{pow}_{\mathrm{LF}}[l..l+s-1]$ and to perform the operation $p \leftarrow \mathrm{LF}^s[p]$. This is done using the following generalization of Lemma 3.

```
PRECOPY(L, $s$)
1:  LF $\leftarrow$ COMPUTELF(L)
2:  $R_s \leftarrow$ COMPUTER$_s$(L, $s$)
3:  for $i \leftarrow 0$ to $|R_s| - 1$ do
4:      visited[$i$] $\leftarrow$ false
5:  runid[0] $\leftarrow 0$
6:  for $i \leftarrow 1$ to $n - 1$ do
7:      runid[$i$] $\leftarrow$ runid[$i - 1$]
8:      if $i \in R_s$ then
9:          runid[$i$] $\leftarrow$ runid[$i$] + 1
10: $p \leftarrow$ locate(L, \$)
11: $l \leftarrow 0$
12: while $l < n$ do
13:     $i \leftarrow$ runid[$p$]
14:     if visited[$i$] = false then
15:         LT$_s$[$i$] $\leftarrow l$; visited[$i$] $\leftarrow$ true
16:         T$^R$[$l$] $\leftarrow$ L[$p$]; pow$_{\text{LF}}$[$l$] $\leftarrow p$
17:         $p \leftarrow$ LF[$p$]; $l \leftarrow l + 1$
18:     else
19:         $l_f \leftarrow$ LT$_s$[$i$]
20:         $p_f \leftarrow$ pow$_{\text{LF}}$[$l_f$]
21:         $s' \leftarrow \min(n - l, s)$
22:         for $k \leftarrow 0$ to $s' - 1$ do
23:             T$^R$[$l + k$] $\leftarrow$ T$^R$[$l_f + k$]
24:             pow$_{\text{LF}}$[$l + k$] $\leftarrow$ pow$_{\text{LF}}$[$l_f + k$] + $(p - p_f)$
25:         $p \leftarrow$ pow$_{\text{LF}}$[$l_f + s'$] + $(p - p_f)$
26:         $l \leftarrow l + s'$
27: return T$^R$
```

Figure 3: PRECOPY algorithm. We assume that COMPUTER$_s$ was implemented as explained in Theorem 7.

Lemma 8. *For any s-run $b..e$, and any $i, j \in b..e$, $k \in 0..s$, $\text{LF}^k[j] = \text{LF}^k[i] + (j - i)$.*

Theorem 9. PRECOPY *runs in $\mathcal{O}(n)$ time and causes $\mathcal{O}(\frac{n}{s} + r_s + \frac{n}{B})$ cache misses provided that $\sigma + \sigma_T B + \mathcal{O}(B) \leq M$.*

Proof. The proof of linear time complexity is trivial.

COMPUTELF was analyzed in Theorem 2 and COMPUTER$_s$ in Theorem 7. The instructions in lines 3–11 are sequential accesses causing $\mathcal{O}(n/B)$ cache misses. The main loop executes lines 15–17 only when the s-run is visited for the first time, causing $\mathcal{O}(r_s)$ cache misses in total. The other case (lines 19–26) increases l by s, and hence will happen at most $\lceil \frac{n}{s} \rceil$ times. Each execution performs $\mathcal{O}(s/B)$ cache misses, causing $\mathcal{O}(\frac{n}{B} + \frac{n}{s})$ cache misses overall. □

Setting $s = \sqrt{\frac{n}{r}}$, we have $\frac{n}{s} = \mathcal{O}(\sqrt{rn})$ and $r_s \leq rs = \mathcal{O}(\sqrt{rn})$.

Corollary 10. PRECOPY *with $s = \sqrt{\frac{n}{r}}$ causes $\mathcal{O}(\sqrt{rn} + \frac{n}{B})$ cache misses.*

Note that if $r = o(n)$, then $\sqrt{rn} = o(n)$. Thus we achieve an asymptotic reduction in cache misses for highly repetitive strings. If $r = \mathcal{O}(n/B^2)$, the number of cache misses is $\mathcal{O}(n/B)$, which is optimal.

Implementations. To save space, the information about an s-run $b..e$, that is stored in R_s, runid, visited, and LT$_s$ in the pseudocode, is encoded within the LF[$b..e$] in the actual implementation of PRECOPY. The encoding uses the fact that the LF values within the s-run form an increasing sequence (Lemma 3) and do not all need to be stored explicitly. Another implementation PRECOPY-LCS computes and stores information about some LCS values larger than s in the same space, and can sometimes copy more than s characters at a time. We omit further details due to lack of space, but Theorem 9 and Corollary 10 hold for these implementations too.

Name	σ_T	n/r	$n/2^{20}$	Source	Description
dna	16	1.59	100	S	Human genome
english	239	2.93	100	S	Gutenberg Project
dblp.xml	97	6.84	100	S	DBLP bibliography
dna.001.1	5	61.10	100	R/PR	100 × 1MB dna
english.001.2	106	72.99	100	R/PR	100 × 1MB english
dblp.xml.0001.1	89	436.23	100	R/PR	100 × 1MB dblp.xml
kernel	160	92.79	246	R/R	36 × Linux Kernel sources
world_leaders	89	80.51	44	R/R	84 × CIA World Leaders
influenza	15	51.28	147	R/R	78041 × virus genome
rs.13	2	2889963	206	R/A	Run-Rich String Sequence

Table 1: Files used in the experiments. The files are from the Pizza & Chili standard corpus[2] (S) and the repetitive corpus[3] (R). The repetitive corpus contains artificially generated sequences (A), files with several variants of the same data (R), and files created from standard corpus files by concatenating 100 copies of a 1MB prefix and mutating them randomly (PR). The value n/r, the average length of a run in the BWT, is a good measure of repetitiveness for our purposes.

Algorithm	Space (bytes)	Description
mtl	$5n + n$	`mergedTL` from [14]
mtl-sa	$6n + n$	mtl with super-alphabet technique
mtl-8	$5n + n$	mtl starting from 8 positions simultaneously
mtl-sa-8	$6n + n$	combination of the two preceding versions
copy	$6n$	detect and copy repetitions in output, from [7]
precopy	$10n$	find and use s-runs for $s = \sqrt{n/r}$
precopy-lcs	$10n$	utilize LCS values in precopy

Table 2: Algorithms and their memory requirements. We assume that an integer/pointer requires 4 bytes and character requires 1 byte for storage. The "$+n$" means that the memory requirement could be reduced by n bytes by reading the input from disk and/or writing the output to disk.

5 Experiments

We performed experiments with the algorithms listed in Table 2 using the files listed in Table 1. All tests were conducted on a 2.66GHz Intel Core2 Duo CPU with 4GB main memory and 4096K L2 Cache. The machine had no other significant CPU tasks running. The operating system was Linux (Ubuntu 10.04) running kernel 2.6.38. The compiler was g++ (gcc version 4.4.3) executed with the -O3 option. The times given are the minima of three runs and were recorded with the C `getrusage` function.

The runtimes are reported in Table 3. They show the effectiveness of the super-alphabet technique and multiple starting points. Each one, alone, gives a significant time reduction, super-alphabet: 38–47 %, multiple starting points: 31–60 %. The

[2] `http://pizzachili.dcc.uchile.cl/texts.html`

[3] `http://pizzachili.dcc.uchile.cl/repcorpus.html`

Testfile	mtl	mtl-sa	mtl-8	mtl-sa-8	copy	precopy	precopy-lcs
dna	125.98	71.71	62.84	39.76	136.37	170.89	183.48
english	117.30	67.90	61.89	39.10	105.76	148.96	160.78
dblp.xml	91.83	56.17	56.74	36.33	80.58	81.72	83.35
dna.001.1	128.93	73.24	51.30	32.23	93.84	43.01	33.28
english.001.2	125.98	71.71	54.07	34.14	95.74	49.87	36.43
dblp.xml.0001.1	124.93	71.04	50.64	32.32	89.55	27.94	22.12
kernel	124.21	72.95	68.07	43.61	86.17	39.58	25.70
world_leaders	112.20	64.72	50.24	32.36	81.33	40.87	30.87
influenza	97.28	59.88	66.92	41.72	80.87	50.32	41.66
rs.13	156.49	83.46	69.71	49.04	102.97	15.96	14.57

Table 3: Times for inverting the BWT. The times are nanoseconds per character and do not include any reading from or writing to disk.

Testfile	copy	precopy	precopy-lcs
dna	7.45	11.01	11.01
english	24.73	26.98	26.98
dblp.xml	28.56	51.05	57.76
dna.001.1	36.05	84.65	94.86
english.001.2	33.10	84.99	94.40
dblp.xml.0001.1	36.68	94.10	98.22
kernel	33.12	87.19	97.57
world_leaders	34.60	85.44	95.15
influenza	34.63	81.20	91.60
rs.13	42.91	99.91	99.97

Table 4: The percentage of data that is sequentially copied (i.e., was processed "as fast as possible", causing $\mathcal{O}(1)$ cache misses per every B symbols decoded) for repetitive-input-driven algorithms.

combination of both achieves 57–75 % (68 % on average) time reduction. The smallest time reduction was observed on the testfiles that mtl processed exceptionally fast (dblp.xml and influenza).

On repetitive data, the algorithms designed for repetitive input easily beat mtl and hence clearly take advantage of the text structure. The precopy-lcs is the fastest, achieving a time reduction of 57–91 % (75 % on average) against mtl and clearly beating even mtl-sa-8 on the most repetitive files. The advantage of the precopy algorithms over the copy algorithm (the previous champion on repetitive input) is largely due to the amount of data that is sequentially copied from the already generated part of the output. The copy algorithm can never copy more than a half of the data. The details are outlined in Table 4.

References

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

[2] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[3] P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. In *Proc. 9th Latin American Theoretical Informatics Symposium*, volume 6034 of *LNCS*, pages 697–710. Springer, 2010.

[4] P. Ferragina and G. Manzini. On compressing the textual web. In *Proc. 3rd ACM International Conference on Web Search and Data Mining*, pages 391–400. ACM, 2010.

[5] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[6] J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theoretical Comput. Sci.*, 387(3):249–257, 2007.

[7] J. Kärkkäinen and S. J. Puglisi. Medium-space algorithms for inverse BWT. In *Proc. 18th European Symposium on Algorithms*, volume 6346 of *LNCS*, pages 451–462. Springer-Verlag, 2010.

[8] J. Kärkkäinen and S. J. Puglisi. Cache-friendly Burrows-Wheeler inversion. In *Proc. 1st International Conference on Data Compression, Communication and Processing*, pages 38–42, 2011.

[9] J. Kärkkäinen and S. J. Puglisi. Fixed block compression boosting in FM-indexes. In *Proc. 18th Symposium on String Processing and Information Retrieval*, volume 7024 of *LNCS*, pages 174–184. Springer-Verlag, 2011.

[10] U. Manber and G. W. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[11] G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.

[12] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

[13] S. J. Puglisi, W. F. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2):1–31, 2007.

[14] J. Seward. Space-time tradeoffs in the inverse B-W transform. In *Proc. IEEE Data Compression Conference*, pages 439–448. IEEE Computer Society, 2001.

[15] J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proc. 15th Symposium on String Processing and Information Retrieval*, volume 5280 of *LNCS*, pages 164–175. Springer, 2008.