Hindawi
Journal of Sensors
Volume 2018, Article ID 6908760, 17 pages
https://doi.org/10.1155/2018/6908760

Research Article

Fast Algorithm of Truncated Burrows-Wheeler Transform Coding for Data Compression of Sensors

Qin Jiancheng[1], **Lu Yiqin**[1] **and Zhong Yu**[2,3]

[1]*School of Electronic and Information Engineering, South China University of Technology, Guangdong, China*
[2]*Zhaoqing Branch, China Telecom Co., Ltd., Guangdong, China*
[3]*School of Software, South China University of Technology, Guangdong, China*

Correspondence should be addressed to Lu Yiqin; eeyqlu@scut.edu.cn

Received 6 October 2017; Revised 3 February 2018; Accepted 19 February 2018; Published 16 April 2018

Academic Editor: Joon-Min Gil

Lots of sensors in the IoT (Internet of things) may generate massive data, which will challenge the limited sensor storage and network bandwidth. So the study of big data compression is very useful in the field of sensors. In practice, BWT (Burrows-Wheeler transform) can gain good compression results for some kinds of data, but the traditional BWT algorithms are neither concise nor fast enough for the hardware of sensors, which will limit the BWT block size in a very small and incompetent scale. To solve this problem, this paper presents a fast algorithm of truncated BWT named "CZ-BWT algorithm" and implements it in the shareware named "ComZip." CZ-BWT supports the BWT block up to 2 GB (or larger) and uses the bucket sort. It is very fast with the time complexity O(N) and fits the big data compression. The experiment results indicate that ComZip with the CZ-BWT filter is obviously faster than bzip2, and it can obtain better compression ratio than bzip2 and p7zip in some conditions. In addition, CZ-BWT is more concise than current BWT with SA (suffix array) sorts and fits the hardware BWT implementation of sensors.

1. Introduction

With the rapid expansion of IoT (Internet of things), lots of sensors are available in various fields, which may generate massive data. Meanwhile, the storage capacity of sensors and network bandwidth are limited, especially in a WSN (wireless sensor network). GBs or TBs of big data in IoT make enormous challenges to the sensors.

Data compression is a smart way to reduce the storage usage and speed up the network transportation. In addition, BWT (Burrows-Wheeler transform [1]) can gain good compression results for some kinds of data. For example, there are a lot of lightweight sensors in a zone of WSN to obtain the temperature data, and most of the data are similar. Thus, a practical way is using some high-performance nodes in this WSN to gather these data, use BWT to compress them and transmit them to the back end cloud platform.

BWT is also valuable in the field of bioinformatics. For example, the big genome data need compression and index, and BWT is an effective way [2, 3]. The DNA data are special and fit the BWT compression. Although we cannot simply compare the bioinformation software such as BWA (Burrows-Wheeler Aligner) and the universal compression software such as bzip2, analyzing their BWT algorithms is meaningful.

But a practical problem is the speed of BWT for sensors and big data. High compression speed is important because the sensors have to treat GBs of data or more, while the traditional BWT algorithms are neither concise nor fast enough, which will limit the BWT block size in a very small and incompetent scale. In our previous paper, we have discussed the traditional compression software bzip2 [4]. Its BWT block size is not more than 900 KB, which will limit the compression ratio. Although it is not large enough to deal with the big data, enlarging the BWT block will observably decelerate the compression. The primary reason is the computing consumption of the traditional BWT algorithms. Besides, the hardware performance and energy consumption of the sensors are limited, which makes it difficult to increase the BWT block size for the big data compression.

We have designed a combined parallel algorithm named "CZ algorithm" to compress and encrypt the big data efficiently and developed our compression software named "ComZip" [5]. Now we have made ComZip compatible to Linux platforms. As mentioned in the figures of [4, 5], ComZip has a BWT filter. This paper focuses on the BWT filter and proposes a fast algorithm of truncated BWT named "CZ-BWT algorithm" to compress the big data efficiently. CZ-BWT algorithm has the following features:

(1) It uses truncated BWT to simplify the algorithm and gain the good performance.

(2) It uses bucket sort to speed up the BWT encoding and decoding with time complexity O(N), so that the BWT block size can rise to 2 GB or more to fit the big data compression.

(3) It can simplify the hardware design of the BWT filter, so that the sensors may use hardware to accelerate the BWT compression.

We did some experiments on both platforms x86/64 and ARM (advanced RISC machines) to compare the efficiencies of data compression among ComZip with/without CZ-BWT, bzip2, and p7zip. The experiment results indicate that ComZip with CZ-BWT filter is obviously faster than bzip2, and it can obtain better compression ratio than bzip2, p7zip, and ComZip itself without CZ-BWT filter in some conditions. In addition, the algorithm analysis infers that CZ-BWT is more concise than current BWT with SA (suffix array) sorts and fits the hardware BWT implementation of sensors.

To make further experiments, we provide 2 versions of ComZip in the website: for Ubuntu Linux (x86/64 platform) and Raspbian (ARM platform). The researchers may download them from http://www.28x28.com/doc/cz_bwt.html.

The remainder of this paper is structured as follows:

Section 2 expresses the problems of BWT for sensors and big data compression. Section 3 introduces the algorithm of CZ-BWT encoding and decoding. Section 4 analyzes the complexities of CZ-BWT algorithm. The experiment results are given in Section 5. The conclusions and future work are given in Section 6.

2. Problems of BWT for Sensors and Big Data Compression

Numerous sensors in IoT can generate big data, but the bottlenecks of data transportation, storage, and computation in the networks of sensors need to be eliminated. Data compression meets this requirement. Figure 1 shows a typical scene in a WSN with both lightweight and heavy nodes, where BWT is feasible.

This WSN has lots of lightweight nodes to sense the situation and generate massive data. Since they have limited energy, storing capacity, and computing resources, they cannot keep the data or achieve the long distance transportation, while a few heavy nodes in the WSN can gather and compress the data and then transport them to the backend cloud platform. The cloud platform has plenty of resources to store, decompress, and analyze the data.

We have discussed the big data compression and encryption in the heavy nodes in a WSN in [5], but if the heavy nodes use BWT, we still have the following problems:

(1) How can the BWT block be enlarged without rapid decrease of the encoding/decoding speed?

(2) Can we design simplified hardware BWT filters for the sensors?

A larger BWT block can gain better compression ratio. In this paper and the previous [4, 5], we use the same definition of the compression ratio as follows:

$$R = 1 - \frac{D_{\text{zip}}}{D}. \quad (1)$$

D_{zip} and D are the volumes of the compressed and original data, respectively. If the original data are not compressed, $R = 0$. If the compressed data are larger than the original data, $R < 0$. Always $R < 1$.

Facing GBs or TBs of big data, a small block of 900 KB cannot show the power of BWT. But enlarging the block will cause the performance bottleneck. As the analyses in Section 4 reveal, BWT encoding speed depends on the string sorting algorithm, and traditional BWT encoding has the time complexity O(N^2lb$N$). N is the block size. If we change the block from 900 KB to 60 MB without any optimization, the encoding will become very slow. This is the first problem.

Although the hardware development improves the performance of the heavy sensors, it is still a challenge for the sensors to achieve fast BWT encoding/decoding. For example, ARM platforms have multicore CPUs with low energy consumption, and the current flash memory has enough capacity and good performance to support a large BWT block, but a practical BWT filter must be fast enough. This is the reason we consider making hardware BWT filters for the sensors.

The problem is that the complex traditional BWT algorithms bring difficulties to the hardware design. If a hardware BWT filter is very complex, its performance will be limited and its energy consumption will be high, and then it is unfit for the sensors.

To solve the problems, we need to review the main related works around sensors and big data compression.

In [5], we have discussed that current mathematic models and methods of lossless compression can be divided into 3 classes:

(1) The compression based on the probabilities and statistics

(2) The compression based on the dictionary indexes

(3) The compression based on the order and repeat of the symbols; BWT belongs to this class

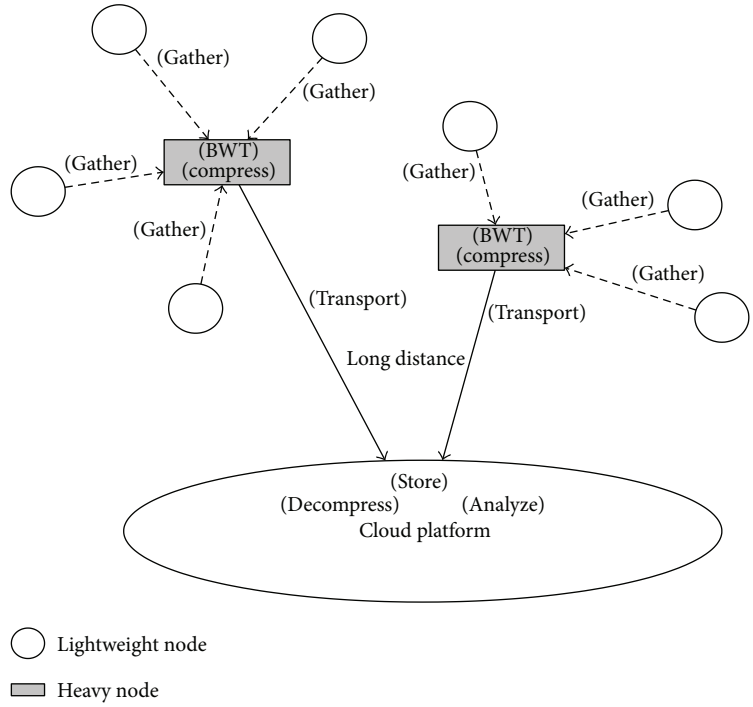

Figure 1: WSN with lightweight and heavy nodes.

Current popular compression softwares are comprehensive applications of the above basic classes, and they have different features, which determine their compression ratio and speed. Especially, to compress big data in the sensors, we have 2 requirements:

(1) Compression speed: fast enough. Since the hardware performance of a sensor is limited, the speed is very important. Too slow softwares such as PAQ and WinUDA are unfit for the big data.

(2) Compression ratio: high. A large data window with good algorithms can benefit the compression ratio. The softwares with too small data windows such as WinZip (512 KB), WinRAR (2 MB), gzip (32 KB), and bzip2 (900 KB block) are unfit for the big data.

In [4, 5], we have developed and updated the compression software ComZip, and in this paper, we developed its Linux version, so that it can run in some sensors such as ARM platforms. ComZip uses all the 3 compression classes:

(1) In class 1, ComZip uses the arithmetic coding [6] and PPM (partial prediction match) algorithm [7], which can gain pretty good compression ratio.

(2) In class 2, LZ77 algorithm [8] is used, which has the advantage of speed.

(3) In class 3, BWT is used, which is the focus in this paper.

To solve the problem of BWT encoding/decoding speed, a lot of algorithms have been developed. Current string sorting algorithms for BWT can reach the speed of linear time complex O(N), for example, some algorithms using SA (suffix array) [9] such as the 3 most popular linear-time algorithms: KS [10], KA [11], and SA-IS [12]. Among them, SA-IS is currently the best algorithm in the speed. Moreover, the further optimization of SA-IS algorithm is studied [13], and the first linear nonrecursive algorithm named GSACA is a new approach for the future [14].

Although current algorithms with SA are faster than the traditional BWT algorithms, it is not so easy to apply them directly to the sensors with the hardware and energy limits. Considering the large BWT block for the big data, the memory requirement of the SA construction is many times of the block. Meanwhile, if we try to design a hardware BWT filter for the sensors, we will meet the complexity of the algorithms such as the recursive computation in SA-IS [13]. GSACA is nonrecursive, but currently, it is slower than SA-IS, and its memory consumption is quite large [14], which are weaknesses for the limited computing resources of the sensors.

Parallel algorithms and the hardware design of BWT are also studied to improve the speed, including the parallel architecture [3, 15] and the practical hardware acceleration, for example, FPGA (field-programmable gate array) [16] and GPU (graphic processing unit) [17]. The advancement is that parallel algorithms benefit the hardware BWT performance, and the researchers tend to simplify the hardware design so that they can obtain higher speeds [3, 17], but the algorithms such as SA-IS are still complex for the sensors. Thus, finding faster and simpler BWT algorithms is useful.

In [3], a limited SA length k is brought into the string sorting, which can reduce the computation. We call this method "truncated BWT." We also use truncated BWT in this paper, but the limited length is different because we do

not use SA, and we use bucket sorting instead of traditional merging or comparing-based sorting.

3. CZ-BWT Encoding and Decoding

3.1. Concepts of CZ-BWT. The compression software ComZip uses the parallel pipeline named "CZ pipeline" and the truncated BWT named "CZ-BWT." We have introduced the framework of CZ encoding pipeline in [5], and the reverse framework is CZ decoding pipeline. Figure 2 is the same encoding framework, and the only difference is the alternative BWT filter in use. CZ-BWT is working in the BWT filter.

CZ-BWT combines the following methods to improve the performance and simplify the algorithm design:

(1) CZ-BWT uses truncated string sorting instead of SA sorting.

As shown in the first figure of [16], the principle of BWT is sorting the data to fit the compression. Sorting is the primary computation in BWT, which determines the performance. We use the same example as that in [16] to explain the truncated string sorting in CZ-BWT.

Figure 3 shows the matrices for BWT sorting. The block size $N = 8$. As shown in (a), the traditional BWT uses full string sorting, which needs comparing of entire strings, for example, Row 0 "XYZAACOL" and Row 1 "YZAACOLX." The sorting result of (a) is shown in (b). Column 0 "AACLOXYZ" is the sorted string, and Column 7 "ZAAOCLXY" is the BWT output string. As shown in (c), the SA sorting ought to compare the suffixes of the same string, e.g., Row 0 "XYZAACOL" and Row 1 "YZAACOL," but the SA algorithms have been optimized to avoid such slow comparison [9–14]. As shown in (d), the truncated string sorting only compares short strings with length $k < N$, for example, Row 0 "XYZ" and Row 1 "YZA" with $k = 3$.

Figure 3(c) shows the SA sorting. Because the common SA sorting result is not always the same as the initial BWT result [1] shown in (b), a special ending symbol "$" is attached to the string tail in order to bridge the gap of the different results. This special symbol has a smaller code (e.g., −1) than any 8b binary code (0 … 255), which means the SA sorting algorithm needs special treatments for the ending symbol besides the normal 8b charset.

Figure 3(e) shows the truncated string sorting result of (d). In this example, (b) and (e) are the same, but in practice, if 2 truncated strings are the same, for example, "ABC" compares to "ABC," the sorting result depends on their original positions. So the decoding algorithm of CZ-BWT is different from that of common BWT.

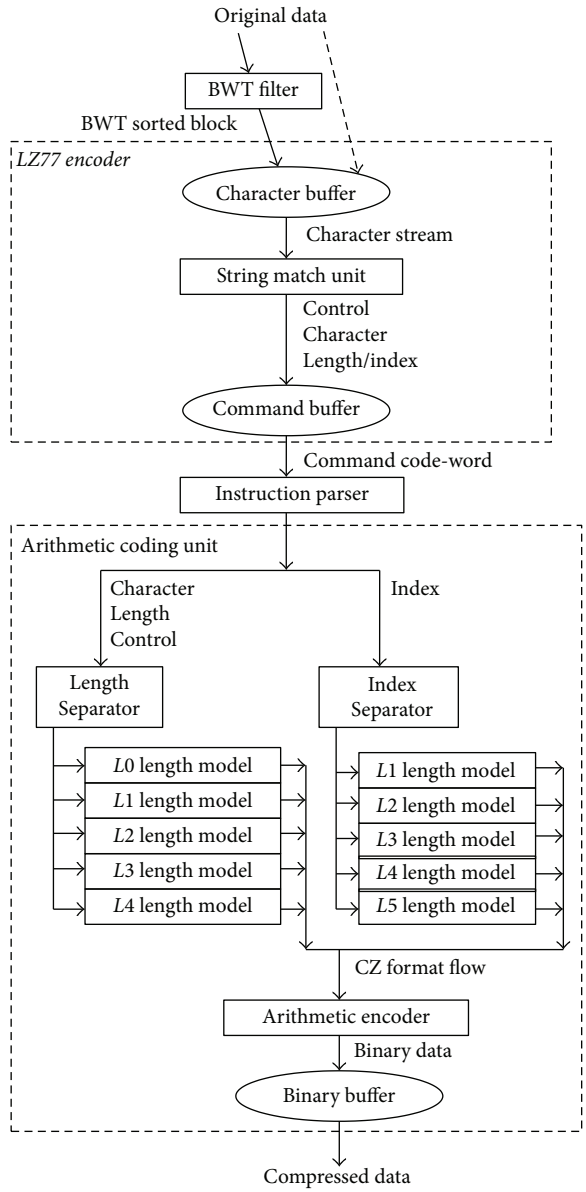

Figure 2: Framework of CZ encoding pipeline with BWT.

(2) CZ-BWT sorts simple integers instead of strings, and it reverses the character sorting sequence indeed.

Figure 4 shows different types of data comparisons. Truncated string comparing, for example, "XYZ" and "ACO" in (a) can be changed into simple integer comparing if we regard "XYZ" as a 24b integer. A 64b integer can substitute a truncated string with length $k < 9$, but in most of the platforms, for example, x86/64 and ARM, the LSB (least significant byte) is in the front, so the sorting sequence of the characters is reversed. As shown in (b), "Z" is the MSB (most significant byte) of the integer "XYZ," so its actual sorting result will be the same as reverse string sorting in (c).

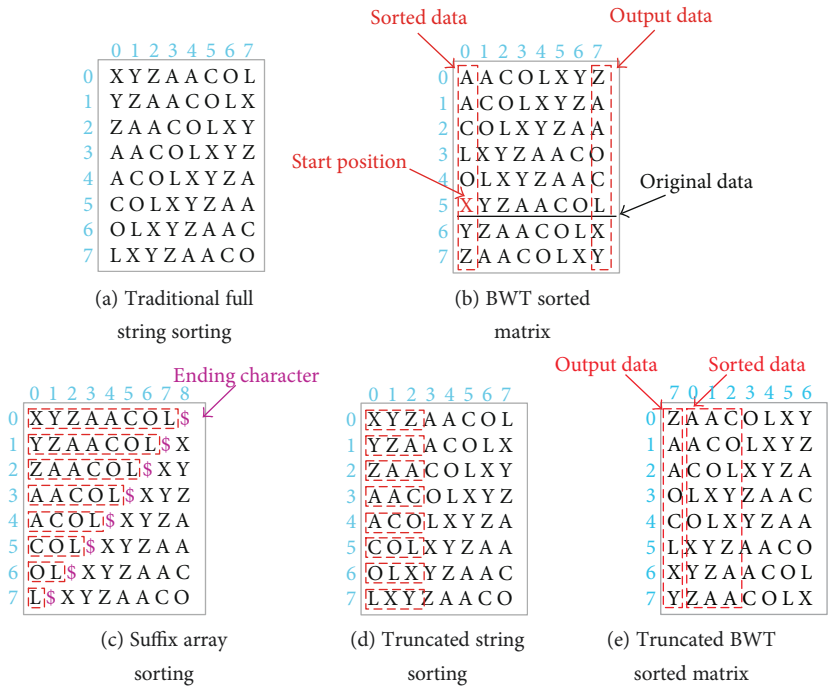

Figure 3: Matrix for BWT sorting.

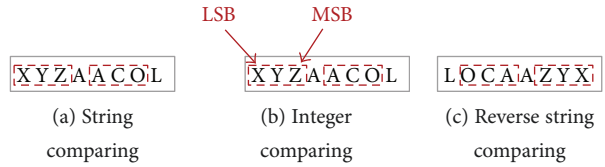

Figure 4: Data comparison for sorting.

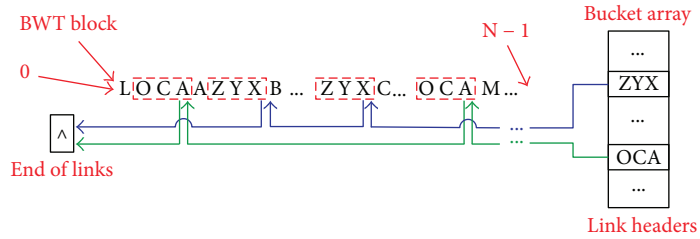

Figure 5: Phase 1 of CZ-BWT encoding ($k = 3$).

(3) CZ-BWT uses bucket sorting instead of traditional merging or comparing based sorting.

Since the sting sorting is changed into integer sorting, CZ-BWT can use bucket sorting. When we use truncated strings with $k = 3$, 256^3 buckets are needed. If the memory is sufficient, $k = 4$ is feasible and 256^4 buckets are needed. Both encoding and decoding in CZ-BWT use bucket sorting.

3.2. CZ-BWT Encoding. We use another example with BWT block length $N > 1$ KB. There are 2 phases in CZ-BWT encoding:

Phase 1 (building the bucket sorting links). We assume the BWT block data is a "cycle" string $s[0 \dots N-1]$, which has the following feature:

$$s[i+N] = s[i] \quad (i = -2, -1, 0, 1, \dots). \tag{2}$$

And $s[i-2 \dots i]$ is a 24b integer ($i = 0, 1, \dots, N-1$). Then we build the bucket sorting links on s. Figure 5 shows the example of 2 links: "ZYX" and "OCA." The bucket array has 256^3 link headers, and all links have the same end: null

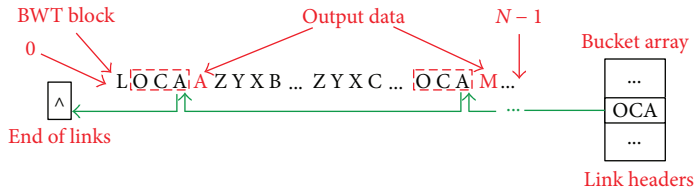

Figure 6: Phase 2 of CZ-BWT encoding ($k = 3$).

pointer. We define the structure of the links and their headers as follows:

$$\text{header}[m] = \begin{cases} \max(i)(s[i-k+1 \dots i] = m) \\ \text{null}(s[i-k+1 \dots i] \neq m) \end{cases} \quad \left(i = 0, 1, \dots, N-1; m = 0, 1, \dots, 256^k - 1\right), \tag{3}$$

$$\text{link}[i] = \begin{cases} \max(j)(s[i-k+1 \dots i] = s[j-k+1 \dots j]) \\ \text{null}(s[i-k+1 \dots i] \neq s[j-k+1 \dots j]) \end{cases} \quad (i = 0, 1, \dots, N-1; j = 0, 1, \dots, i-1). \tag{4}$$

Phase 2 (outputting the sorted data). We follow each link to output the data. Figure 6 shows the example of the link "OCA," which will output the characters "MA," referring to Figure 3(e). And finally we output the start position of the block for CZ-BWT decoding.

Algorithm 1 shows the CZ-BWT encoding algorithm.

3.3. CZ-BWT Decoding. We use the same example as shown in Figure 3, but the string is reversed into "LOCAAZYX" because of the MSB/LSB in the integer sorting. Figure 7(a) shows the full decoding matrix of CZ-BWT, which corresponds to Figure 3(e). And we ought to pay attention to the column numbers of this matrix: Row 6 is the reversed string "YZAACOLX," and Row 5 is the original data, reversed string "XYZAACOL."

Recovering the whole decoding matrix is not necessary. Because CZ-BWT uses truncated data sorting, its decoding is different from that of the general BWT. As shown in Figure 7, there are 4 phases in CZ-BWT decoding:

Phase 1 (building the second column of the matrix). As shown in Figure 7(b), this phase is the 8b integer bucket sorting. The second column is Column 7, and Column 0 stores the input data, which are the output data of CZ-BWT encoding. The bucket array has 2^8 counters, so that we can scan Column 0 once and write the sorted data to Column 7.

Phase 2 (building the third column of the matrix). As shown in Figure 7(c), this phase is the 16b integer bucket sorting. The bucket array has 2^{16} counters, so that we can scan Column [0,7] once and write the sorted data to Column [7,6]. Because the amount of the 16b integers is related to the previous 8b integers, Column 7 will be the same as that in phase 1. Thus, we can write Column 6 only.

Phase 3 (building the forth column of the matrix). As shown in Figure 7(d), this phase is the 24b integer bucket sorting. The bucket array has 2^{24} counters, and we can scan Column [0,7,6] once and write the sorted data to Column [7,6,5]. But this time, we need not write Column 5, because the link headers in phase 4 have the same 24b sorting effect already. Hiding the writing back operation can simplify this algorithm and improve the decoding speed.

Phase 4 (outputting along the bucket sorting links). As shown in Figure 7(e), this phase is the outputting of the decoded block. We can easily change the bucket array from data counters into link headers by accumulating the counter values, because Column [7,6,5] is sorted, and the current link header position adds that the current counter value is the next link header. For example, in Figures 7(d) and 7(e), we focus on Column [7,6,5] and notice

$$\text{header}[\text{"LXY"}] + \text{counter}[\text{"LXY"}] = 3 + 1 = 4 = \text{header}[\text{"OLX"}]. \tag{5}$$

According to the "cycle" string $s[0 \dots N-1]$ as shown in (2), we define the data counters in Figure 7 as follows:

$$\text{counter}[m] = \sum_{i=0}^{N-1} \begin{cases} 1(s[i-k+1 \dots i] = m) \\ 0(s[i-k+1 \dots i] \neq m) \end{cases} \quad \left(m = 0, 1, \dots, 256^k - 1\right). \tag{6}$$

Then we can get the link headers in Figure 7(e) from the data counters in Figure 7(d) as follows:

$$\text{header}[m] = \sum_{i=0}^{m} \text{counter}[i] \quad \left(m = 0, 1, \dots, 256^k - 1\right). \tag{7}$$

```
function encode(s) {      /* s is the BWT block data string (original data) */
    link = array(0 … N − 1);      /* N is the length of string s */
    bucket = array(0…256^3−1);      /* bucket stores the link headers of (3) */
    for j = 0…256^3−1 do {bucket[j] = null;}      /* Initialize the link headers */
    for i = 0 … N − 1 do {j = s[i − 2 … i];   link[i] = bucket[j];   bucket[j] = i;}      /* Phase 1: build links of (4) */
    count = 0;      /* count traces the start position of the block */
    for j = 0…256^3−1 do {      /* Phase 2: output data */
      i = bucket[j];
      while i is not null do {
        output(s[i + 1]);   i = link[i];   count = count+ 1;      /* output a character of s */
        if i = N − 1 do {start = count;}      /* start stores the start position */
      }
    }
    output(start);      /* finally output the start position */
}
```

Algorithm 1: (CZ-BWT encoding ($k = 3$)).

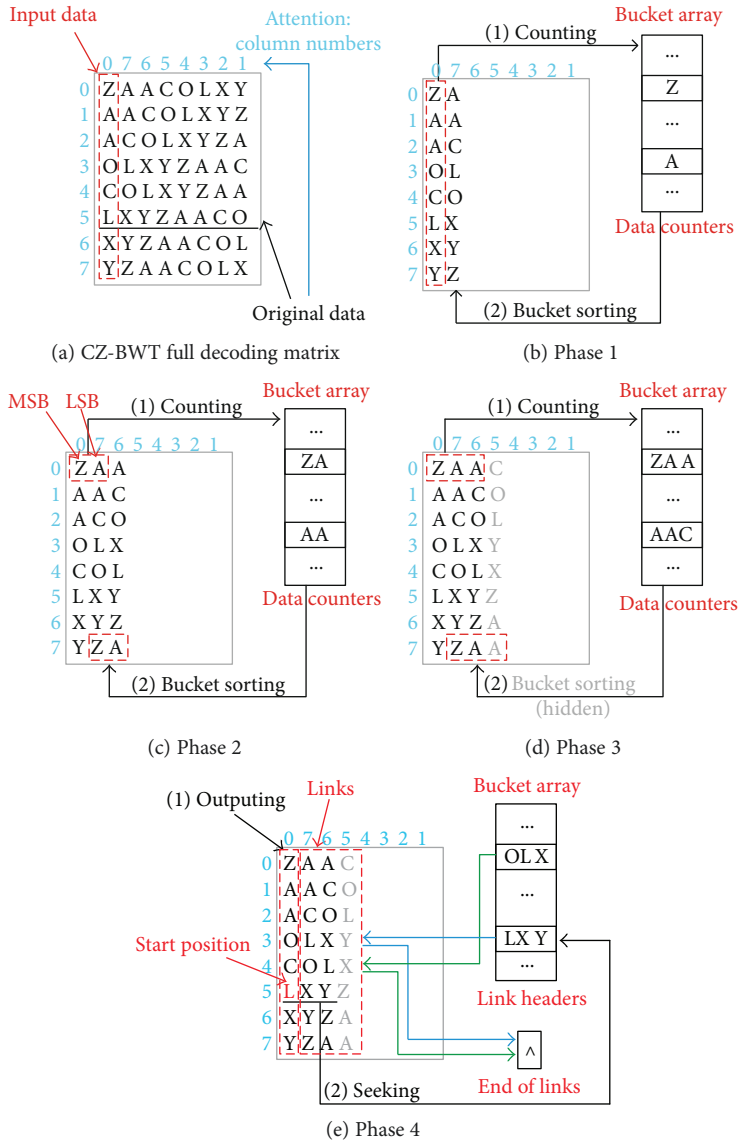

Figure 7: CZ-BWT decoding ($k = 3$).

```
function decode(s) {      /* s is the BWT block data string (BWT encoded data) */
    link = array(0 … N − 1);      /* link stores the Column [0,7,6,5] of the decoding matrix. N is the length of string s. */
    bucket_A = array(0…256^3−1);      /* bucket_A stores the data counters */
    for j = 0…256^3−1 do {bucket_A[j] = 0;}      /* Initialize the data counters */
    bucket_B = array(0…256^2−1);      /* bucket_B stores the data counters */
    for j = 0…256^2−1 do {bucket_B[j] = 0;}      /* Initialize the data counters */
    for i = 0 … N − 1 do {j = s[i];   link[i] = j;   bucket_A[j] = bucket_A[j] +1;}      /* Phase 1: count Column 0 */
    p = 0;      /* p traces the current position of link */
    for i = 0…256−1 do {
      j = bucket_A[i];   bucket_A[i] = 0;      /* Initialize the data counters */
      while j > 0 do {
        m = (link[p] < <8) | i;      /* m stores Column [0,7] */
        link[p] = m;   p = p + 1;      /* Phase 1: sort Column 7 */
        bucket_B[m] = bucket_B[m] +1;   j = j − 1;      /* Phase 2: count Column [0,7] */
      }
    }
    p = 0;      /* reset p */
    for i = 0…256^2−1 do {
      j = bucket_B[i];
      while j > 0 do {
        m = (link[p] < <8) | i;      /* m stores Column [0,7,6] */
        link[p] = m;   p = p + 1;      /* Phase 2: sort Column 6 */
        bucket_A[m] = bucket_A[m] +1;   j = j − 1;      /* Phase 3: count Column [0,7,6] */
      }
    }
    p = 0;      /* reset p */
    m = 0;      /* m traces the link headers */
    for i = 0…256^3−1 do {      /* Phase 4: calculate link headers */
      m = m + bucket_A[i];   bucket_A[i] = m;      /* bucket_A stores the link headers of (7) */
    }
    input(start);      /* input the start position of the block */
    j = start;      /* j traces the position */
    for i = 0 … N − 1 do {      /* Phase 4: output decoded data */
      p = link[j];      /* link keeps Column [0,7,6] after phase 2 */
      s[i] = (p > > 16) & 255;      /* s stores the decoded block string in Column 0 */
      j = bucket_A[p] −1;      /* seek the next position j with Column [0,7,6]: fetch & decrease */
      bucket_A[p] = j;      /* update the current link header of (8) */
    }
    output(s[0 … N − 1]);      /* finally output the block string */
}
```

Algorithm 2: (CZ-BWT decoding ($k = 3$)).

Here is a trick for the algorithm optimization. The exact value of a header ought to be 1 smaller than that in (7). For example, according to (7), header$[''\text{LXY}'']=4$, while in (5), header$[''\text{LXY}'']=3$ exactly. Now we explain this trick:

There are 256^3 links in Column [7,6,5] in Figure 7(e), and their 256^3 headers are dynamic. In this phase, the headers are calculated with (7) at first, and then each output character of string s will cause that a corresponding header switches to the next link node position:

$$\text{header}[m] = \text{header}[m] - 1. \tag{8}$$

When a counter gets a value larger than 1, for example, in Figures 5 and 6counter$[''\text{ACO}'']=2$ (the string is reversed), the dynamic header is useful to determine which is the current link node position. The next position of the same link is easily calculated with (8) because Column [7,6,5] in Figure 7(e) are already sorted.

The trick can save the algorithm operations: Since each time we have to fetch a header value to locate the position, and decrease the value with (8) for the future fetch, we may simply merge the "fetch" and "decrease" operations. So long as the initial value of each header in (7) is 1 larger than the exact value, we can use the "fetch & decrease" operation each time.

Algorithm 2 shows the CZ-BWT decoding algorithm. We use 2 bucket arrays to mix the phases and save the time of data accessing.

3.4. Example of CZ-BWT Decoding. To explain how Algorithm 2 works, we provide another example of CZ-BWT decoding in detail.

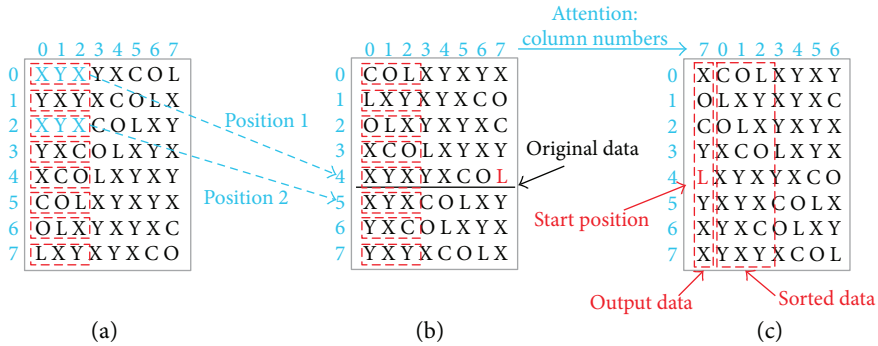

Figure 8: CZ-BWT encoding example: (a) truncated string sorting ($k = 3$); (b) BWT sorted matrix; (c) BWT sorted matrix output.

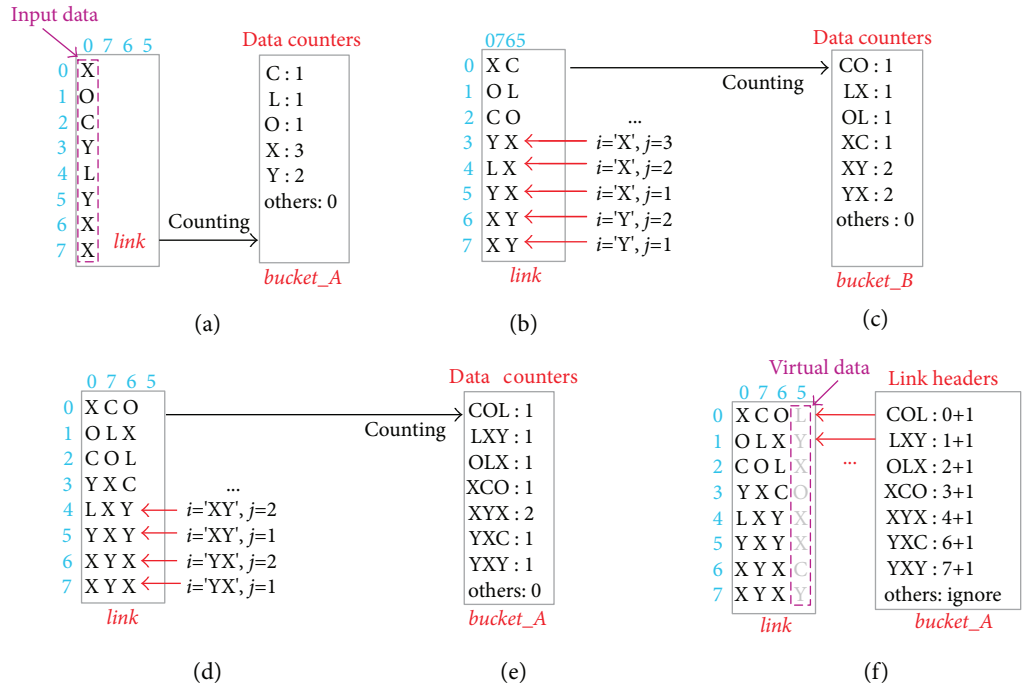

Figure 9: CZ-BWT decoding example: (a) Phase 1: count column 0; (b) Phase 1: sort column 7; (c) Phase 2: count column [0,7]; (d) Phase 2: sort column 6; (e) Phase 3: count column [0,7,6]; (f) Phase 4: calculate link headers.

In this example, we use the reversed string "XYXYXCOL" as the original data, which has two matches of the characters "XYX." Figure 8 shows the CZ-BWT encoding of Algorithm 1. We can find the relationship between Figures 3(d) and 3(e) and Figures 8(a) and 8(c). And in Figure 8(a), characters "XYX" in positions 1 and 2 are sorted. As a result, Figure 8(b) shows the unchanged "XYX" position sequence. The similar situation is in Figure 6.

Figures 9 and 10 show decoding phases 1 to 4 according to Figure 7. As the phases and key operations are described in Algorithm 2, we can see the data changes from Figures 9 and 10, so that we can follow the process of decoding the reversed string "XOCYLYXX."

In Figure 9(a), the input data "XOCYLYXX" are counted and then sorted in Figure 9(b). This is a typical bucket sorting with 256 counters. And the bucket sorting proceeds again in Figures 9(c) and 9(d), with 256^2 counters. The array *link* stores the sorting results.

In Figure 9(e), the data are counted with 256^3 counters, but the sorting is hidden in the calculation of link headers in Figure 9(f). Thus, the *link* does not store Column 5 indeed.

Figure 10 shows the data output process in phase 4. The practical output operation in Algorithm 2 is using a string s to store the output characters. Figures 10(a) and 10(b) give the example of outputting $s[0]$ and $s[1]$.

The start position is Row 4 in Figure 10(a). Each time, the algorithm outputs a character by the following steps:

(1) Outputting the character of Column 0 in the *link*. In Figure 10(a), $s[0] = "L."$

(2) Fetching and decreasing the link header value with Column [0,7,6] in the *link*. In Figure 10(a), header$["LXY"] = 1$.

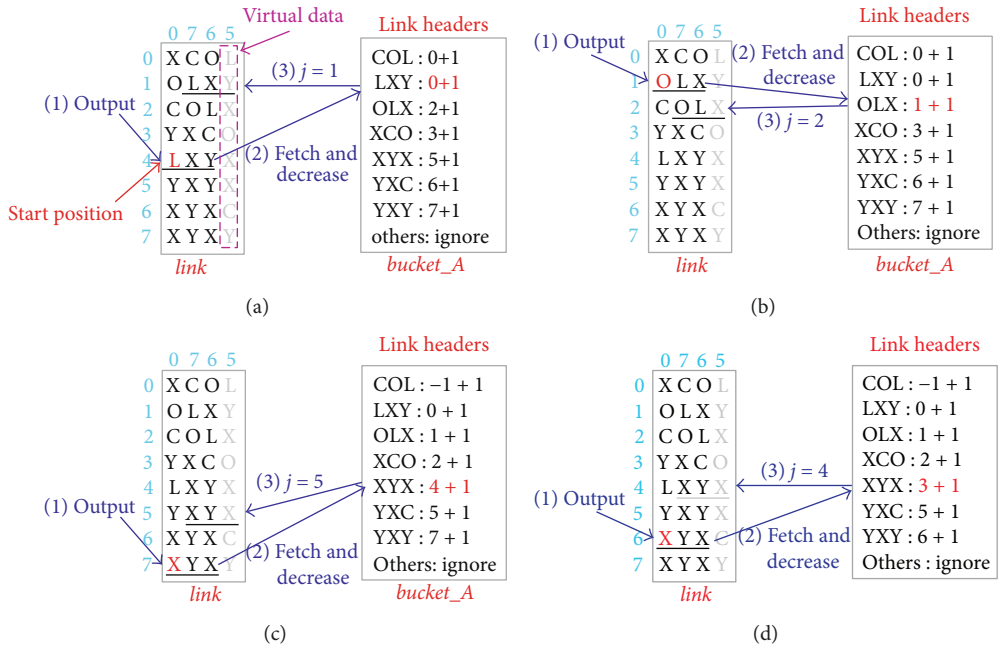

Figure 10: CZ-BWT decoding example: output. (a) Phase 4: output character $s[0]$; (b) phase 4: output character $s[1]$; (c) phase 4: output character $s[5]$; (d) phase 4: output character $s[7]$ (end). Finally, reversed $s = "LOCXYXYX."$

(3) Keeping the header value as the next position to output a character. $j = 1$.

In Figure 10(b), the steps are executed again:

(1) $s[1] = "O"$.

(2) header$["OLX"] = 2$.

(3) $j = 2$.

The steps go on repeating until we gain the full-length $s = "LOCXYXYX,"$ which are reversed. As mentioned in Section 3.3, CZ-BWT decoding outputs reversed data. As CZ-BWT encoding also outputs reversed data through the backward links in Figures 5 and 6, this decoding algorithm can reverse the data again and finally gain the original data.

As shown in Figures 10(c) and 10(d), the decoding algorithm can maintain the correct values of the dynamic headers, for example, header["XYX"], which keeps the proper order of the character outputs.

4. Analyses of the CZ-BWT Algorithm

Quite a few recent advancements of BWT algorithms are driven by the rapid development of genome information technologies [2, 3, 13, 18], and there are many DNA softwares using BWT, including DNA compression, alignment, sequencing, and indexing. Due to the difference between the DNA and common data charsets, we cannot proceed direct experiments to compare a DNA software such as BWA (Burrows-Wheeler aligner) with a universal compression software such as ComZip or bzip2, but we can analyze their BWT algorithms to investigate their advantages and shortcomings.

4.1. Time Complexities. We may study the BWT encoding and decoding algorithms by analyzing their time and space complexities in the worst cases. First, it is known that the traditional BWT encoding algorithm has the time complexity $O(N^2 \text{lb} N)$. N is the block size. The analyses are as follows:

According to the principle of BWT compression [1], the key computation of BWT encoding is the string sorting, which determines the encoding speed. And the string sorting consists of 2 algorithms:

(1) The comparison of 2 strings: The length of each string is equal to the BWT block size N, so this string comparison has the time complexity $O(N)$.

(2) The sorting of data elements: In BWT encoding, a data element is a string, and the amount of the strings is equal to the block size N. Some traditional sorting algorithms such as quick sort and heap sort have the time complexity $O(N \text{lb} N)$, and it has been proven that $O(N \text{lb} N)$ is the fastest level in all comparison-based algorithms.

From the above 2 algorithms, we find that the fastest traditional BWT encoding has the time complexity $O(N^2 \text{lb} N)$. It is not fast enough. As mentioned in Section 2, the current well-known fastest string sorting algorithm is SA-IS, which has the time complexity $O(N)$. So we compare CZ-BWT encoding which is used in ComZip and the SA-IS encoding which is used in BWA.

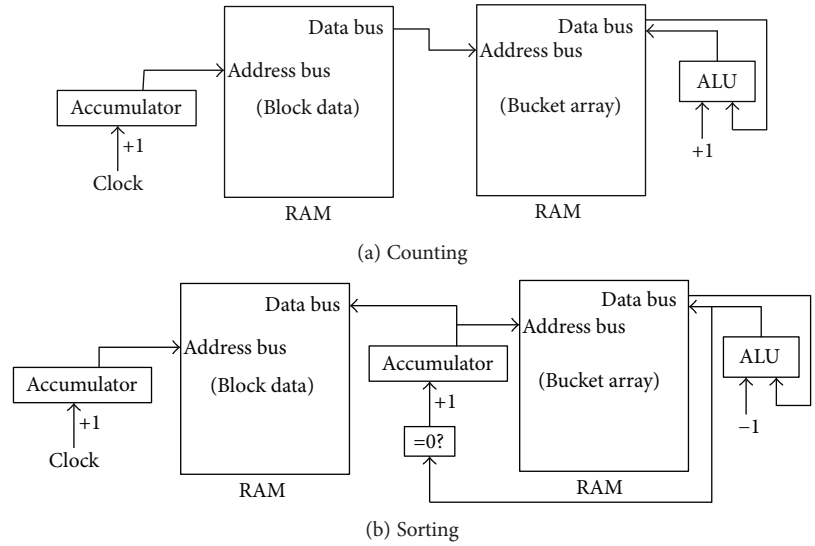

Figure 11: Primary hardware design of bucket sort.

Table 1: Comparison of typical BWT algorithms.

BWT algorithm	Software	Time complexity	Space complexity	Memory usage	Advantages	Weaknesses
Traditional BWT	bzip2	$O(N^2 \text{lb} N)$	$O(N)$	$2.5N$	Simple for general usage	Small block size; low compression ratio; weak big data support
BWT with SA-IS	BWA	$O(N)$	$O(N)$	$5.37N$	Well known fastest standard BWT	Slower encoding than CZ-BWT; complex implementation; difficult hardware design
BWT with GSACA	GitHub: gsaca	$O(N)$	$O(N)$	$16N$	The first linear nonrecursive SA algorithm for BWT	Slower encoding and more memory consumption than SA-IS and CZ-BWT
CZ-BWT	ComZip	$O(N)$	$O(N)$	$4N$	Faster than BWT with SA-IS; simple hardware design	Slower decoding than standard BWT; need larger block size for higher compression ratio

According to Algorithm 1, we find that the bucket sorting also has the time complexity $O(N)$, but we can compare more details.

SA-IS requires special ending symbol for the block and recursive reduction to a shorter string [2, 13], which are more complex than CZ-BWT. SA-IS needs to scan the BWT block for more than 3 times, while according to Algorithm 1, CZ-BWT encoding just scan the block twice in phases 1 and 2. Thus, CZ-BWT encoding is faster indeed.

GSACA also requires special ending symbol. It is nonrecursive, and it also has 2 phases [14], but each phase has much more operations than simply scanning the block in CZ-BWT encoding. These operations makes GSACA slower than SA-IS and CZ-BWT currently.

4.2. Space Complexities. The memory usage is important for the sensors. SA-IS, GSACA, and CZ-BWT have the same space complexity $O(N)$. In detail, BWA uses the RAM (random access memory) of $5.37N$; GSACA uses $12N$ besides $4N$ for the suffix array, and CZ-BWT uses $4N$ to store the links for the block size up to 2 GB. Moreover, it is easy for CZ-BWT to use $5N$ for the block size of up to 512 GB. In this view, CZ-BWT needs less memory for the block than BWA with SA-IS and the GSACA program.

But CZ-BWT requires extra RAM for the bucket array. If the block size is not more than 2 GB, a bucket counter uses 4 B, and a bucket array has 256^k elements. If $k = 3$, a bucket array uses 64 MB, which is feasible in a heavy sensor node. And if $k = 4$, it needs 16 GB, which is feasible in the current cloud platforms.

4.3. Complexities of Hardware Design. Hardware acceleration is valuable for the sensors which have limited computing resources. Due to the complexity of SA-IS, it is difficult to implement the hardware BWT with SA-IS. As a contrast, CZ-BWT is simpler and easier for the hardware acceleration.

The figures in [16] show that the truncated BWT fits the hardware design, but it uses merge sort, which has the time complexity $O(N^2 \text{lb} N)$. CZ-BWT uses bucket sort, which is both fast and easy for the hardware design. Figure 11 shows the primary hardware design of bucket sort. We take Figure 11(a), for example. The accumulator can indicate the current position of the block $(0, \ldots, N-1)$; thus, it can directly connect to the address bus of the block data RAM.

Table 2: x86/64: comparison of compressed file (book.htm) size (B).

Data window/block size	ComZip with CZ-BWT filter	bzip2	ComZip without CZ-BWT filter	p7zip
0.1 MB	—	6,554,263	—	—
0.2 MB	—	4,819,250	—	—
0.3 MB	—	4,130,297	—	—
0.4 MB	—	3,767,164	—	—
0.5 MB	—	3,524,473	—	—
0.6 MB	—	3,355,229	—	—
0.7 MB	—	3,220,144	—	—
0.8 MB	—	3,122,484	—	—
0.9 MB	—	3,041,287	—	—
1 MB	3,647,168	—	4,793,400	5,876,069
2 MB	3,351,504	—	4,615,872	5,797,084
4 MB	3,131,992	—	4,474,424	5,724,417
8 MB	2,953,456	—	4,387,672	5,667,650
16 MB	2,823,168	—	4,318,288	5,614,144
32 MB	2,709,192	—	4,264,320	5,563,178
64 MB	2,588,584	—	4,227,920	5,518,719
128 MB	2,467,504	—	4,200,928	5,481,954
256 MB	2,397,504	—	4,187,456	5,464,699
512 MB	2,306,528	—	4,186,576	5,464,344

And this RAM can provide the data, for example, "LXY," to the address bus of the bucket array RAM. Then, the latter RAM provides the current counter value to the ALU (arithmetic and logic unit), which will update the value and write it back to the RAM. We can use simple sequence-control logic circuits to make this module work. In the view of hardware, both (a) and (b) in Figure 11 are succinct and easy to optimize the hardware speed.

4.4. Weaknesses. As a truncated BWT, CZ-BWT cannot use the standard BWT decoding, which can be used by the BWT with SA-IS. Both CZ-BWT and the standard BWT decoding have the time complexity O(N), but the latter is faster. According to Algorithm 2, CZ-BWT decoding has to scan the block for 4 times from phases 1 to 4, while the standard BWT decoding can scan the block only twice.

This weakness is acceptable for the sensors. Although CZ-BWT decoding is slower than the standard BWT, its speed is still in the linear level. And it has no complex implementation such as the special ending symbol and the recursive algorithm, so the hardware acceleration for CZ-BWT can be used to the sensors in a relatively easy way. Moreover, the typical scene in Figure 1 infers that most of the BWT decoding events occur in the cloud platform, which has plenty of computation resources. So the speed of CZ-BWT decoding is fast enough in this case.

Another weakness is that the truncated BWT has lower compression ratios than the standard BWT. But we can use a larger block in CZ-BWT to keep up with the compression ratio. The experiment results show this accomplishment.

Table 1 shows the comparison of typical BWT algorithms. We ought to distinguish the concepts of BWT: CZ-BWT is a kind of truncated BWT, while bzip2, SA-IS, and GSACA use standard BWT, but bzip2 uses the traditional BWT, which is slow.

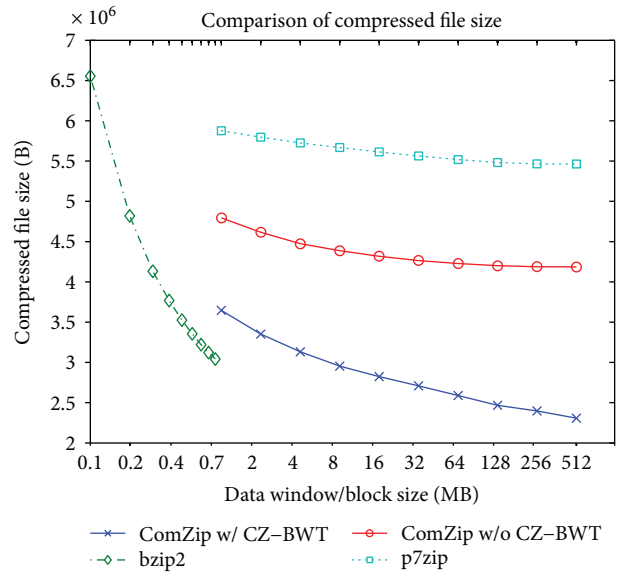

Figure 12: Compressed file size and data window/block size in Table 2.

5. Experimental Results

We have done some experiments to compare ComZip, WinRAR, and 7-zip in [5]. The results indicate that ComZip with a large data window has better compression ratio than WinRAR and 7-zip in most cases, and its compression speed

Table 3: x86/64: comparison of file (book.htm) compression/decompression time (seconds).

Data window/ block size	Compression time				Decompression time			
	ComZip with CZ-BWT filter	bzip2	ComZip without CZ-BWT filter	p7zip	ComZip with CZ-BWT filter	bzip2	ComZip without CZ-BWT filter	p7zip
0.1 MB	—	56	—	—	—	5	—	—
0.2 MB	—	64	—	—	—	6	—	—
0.3 MB	—	73	—	—	—	6	—	—
0.4 MB	—	78	—	—	—	6	—	—
0.5 MB	—	83	—	—	—	5	—	—
0.6 MB	—	86	—	—	—	6	—	—
0.7 MB	—	91	—	—	—	6	—	—
0.8 MB	—	91	—	—	—	6	—	—
0.9 MB	—	95	—	—	—	6	—	—
1 MB	38	—	3	19	23	—	6	1
2 MB	32	—	3	20	19	—	6	1
4 MB	28	—	3	19	16	—	5	1
8 MB	25	—	4	19	15	—	5	1
16 MB	24	—	4	20	15	—	5	1
32 MB	24	—	4	20	10	—	5	1
64 MB	24	—	3	20	15	—	6	1
128 MB	24	—	4	20	15	—	6	1
256 MB	26	—	4	20	17	—	5	1
512 MB	29	—	5	21	16	—	5	1

is faster than 7-zip. But those experiments do not use the BWT filter, which has CZ-BWT algorithms for ComZip.

In this paper, we compare the following softwares in the experiments: ComZip with CZ-BWT, bzip2, ComZip without CZ-BWT, and p7zip (7-zip for Linux). When we test ComZip without CZ-BWT, we observe its data window size. When we test ComZip with CZ-BWT, we observe its block size and use a fixed 4 MB data window for its LZ77 algorithm [8]. We choose a small data window of 4 MB to extrude the abilities of CZ-BWT, and a data window smaller than 4 MB may reduce the performance of the BWT filter.

The experiments in this paper are on 2 hardware platforms: x86/64 and ARM. Their performances may provide references to the future and current heavy sensor nodes. The operating systems of both experiment platforms are Linux. We have developed ComZip for Linux, and we still provide ComZip in the website. Researchers may use it to do more experiments with new data. It can be downloaded from http://www.28x28.com/doc/cz_bwt.html.

5.1. Tests on the x86/64 Platform. This platform is a common laptop with the following equipments: Intel Core i7-4700MQ 4-core & 8-thread CPU, 16 GB DDR3 RAM, and 128 GB SSD (Solid State Disk) and Ubuntu Linux 12.10 (x64). We regard this laptop as a future high-end mobile sensor when the fuel cell can provide enough energy. The software versions are ComZip v20171019 (64b), bzip2 1.0.6, and p7zip 9.18.

In this experiment, we use different data windows or block sizes to compress the same original file named "book.htm," which is an example that some kinds of data can show the advantage of BWT in the compression ratio.

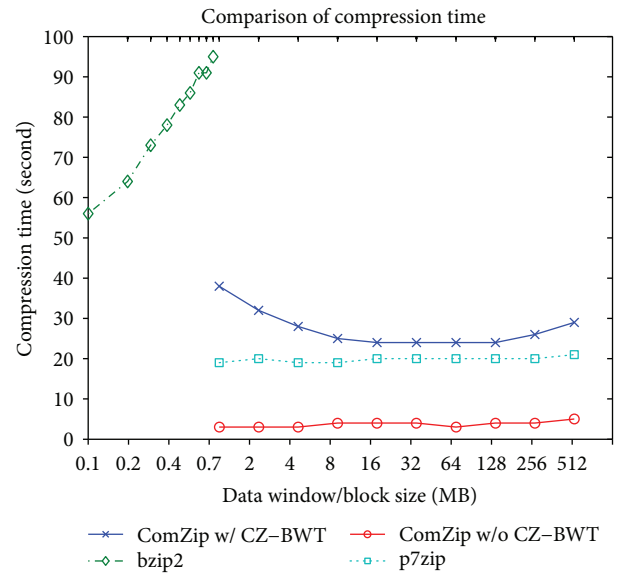

Figure 13: Compression time and data window/block size in Table 3.

This is a real Chinese bookshop data file of storage records in HTML/XML format. Its original length is 346,499,594 B. It can be downloaded from http://www.28x28.com/doc/book.htm.bz2.

Table 2 and Figure 12 show the relationship of the compressed file size and the data window/block size. From (1), we can find that this relationship is virtually the relationship of the compression ratio and the window size.

Table 4: ARM: comparison of compressed file (book.htm) size (B).

Data window/block size	ComZip with CZ-BWT filter	bzip2	ComZip without CZ-BWT filter	p7zip
0.1 MB	—	6,554,263	—	—
0.2 MB	—	4,819,250	—	—
0.3 MB	—	4,130,297	—	—
0.4 MB	—	3,767,164	—	—
0.5 MB	—	3,524,473	—	—
0.6 MB	—	3,355,229	—	—
0.7 MB	—	3,220,144	—	—
0.8 MB	—	3,122,484	—	—
0.9 MB	—	3,041,287	—	—
1 MB	3,647,176	—	4,793,424	5,876,069
2 MB	3,351,448	—	4,615,888	5,797,084
4 MB	3,131,976	—	4,474,472	5,724,417
8 MB	2,953,472	—	4,387,688	5,667,650
16 MB	2,823,152	—	4,318,312	5,614,144
32 MB	2,709,176	—	4,264,304	5,563,178
48 MB	2,605,088	—	—	—
56 MB	2,603,848	—	—	—
64 MB	Insufficient RAM	—	4,227,920	5,518,719
96 MB	—	—	4,210,688	—

Table 3 and Figure 13 show the relationship of the compression/decompression time and the data window/block size. Figure 13 hides the decompression time because this weakness of CZ-BWT is analyzed in Section 4. We focus on the compression performance first, and the optimization of decompression for ComZip is our future work.

In Table 2 and Figure 12, we observe that ComZip with CZ-BWT has the best compression ratio among these softwares, and p7zip has the worst except the 0.1 MB block of bzip2. The 0.1 MB block is too small for the big data compression. If the block is large enough, the standard BWT in bzip2 has better compression ratio than truncated BWT indeed. When bzip2 uses 0.9 MB block, ComZip has to use about 7 MB block to gain better compression ratio.

But enlarging the block for bzip2 is not practical. Table 3 and Figure 13 show that bzip2 has the slowest compression speed, and its curve raises rapidly, which can exhibit the analysis that traditional BWT has the time complexity $O(N^2 \text{lb} N)$. We can estimate the speed of bzip2 with a 512 MB block.

According to the compression speed shown in Figure 13, ComZip with CZ-BWT is slower than p7zip, but their curves are close. The curve from 1 to 8 MB show that a block smaller than 8 MB may reduce the performance of CZ-BWT with 4 MB data window, and the curve from 8 to 512 MB can exhibit the analysis that CZ-BWT has the time complexity $O(N)$. If we can find an universal compression software with SA-IS, we suppose its curve will like this one for CZ-BWT.

ComZip without CZ-BWT is much faster than others in Figure 13. We can provide 2 possible reasons. The first reason is the parallel CZ encoding pipeline, which is introduced in [5]. This platform with 8-thread CPU, large RAM, and SSD may release the good performance of the pipeline. The second reason is the data file for this experiment fits the optimized LZ77 algorithm, which is mentioned in [4], so the performance of ComZip is evident.

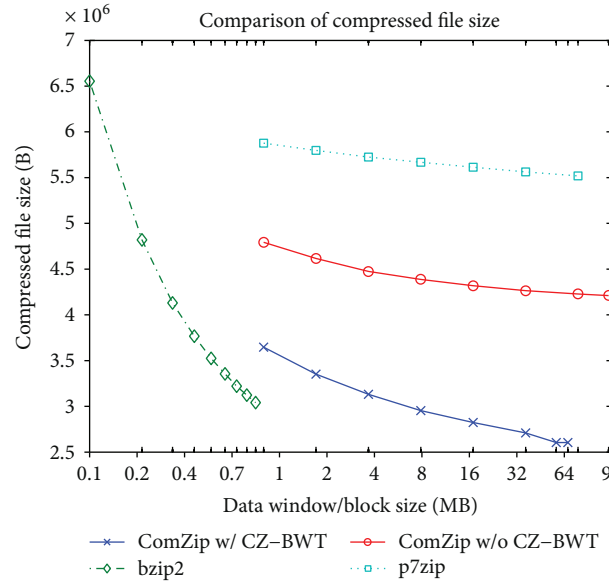

Figure 14: Compressed file size and data window/block size in Table 4.

Above all, the experiment results on this x86/64 platform show that ComZip with CZ-BWT can have the best compression ratio among these softwares, and its compression speed is near p7zip, which is practical for the big data.

5.2. Tests on the ARM Platform. This platform is a popular Raspberry Pi 2 Model B with the following equipments: ARM Cortex-A7 4-core CPU, 1 GB DDR RAM, 64 GB Micro

Table 5: ARM: comparison of file (book.htm) compression/decompression time (seconds).

Data window/block size	Compression time				Decompression time			
	ComZip with CZ-BWT filter	bzip2	ComZip without CZ-BWT filter	p7zip	ComZip with CZ-BWT filter	bzip2	ComZip without CZ-BWT filter	p7zip
0.1 MB	—	829	—	—	—	59	—	—
0.2 MB	—	949	—	—	—	62	—	—
0.3 MB	—	1099	—	—	—	86	—	—
0.4 MB	—	1265	—	—	—	91	—	—
0.5 MB	—	1667	—	—	—	100	—	—
0.6 MB	—	1851	—	—	—	106	—	—
0.7 MB	—	1990	—	—	—	107	—	—
0.8 MB	—	2045	—	—	—	97	—	—
0.9 MB	—	2223	—	—	—	103	—	—
1 MB	325	—	156	303	151	—	27	19
2 MB	226	—	156	309	107	—	27	16
4 MB	178	—	162	309	83	—	25	18
8 MB	240	—	160	313	73	—	24	17
16 MB	407	—	160	310	69	—	24	23
32 MB	556	—	156	331	70	—	23	18
48 MB	511	—	—	—	67	—	—	—
56 MB	567	—	—	—	69	—	—	—
64 MB	—	—	159	320	—	—	23	20
96 MB	—	—	164	—	—	—	23	—

SDXC (SD eXtended Capacity), and Raspbian Linux 7. We regard this Raspberry Pi as a current heavy node of mobile sensors, which is inexpensive. The software versions are ComZip v20171019 (32b), bzip2 1.0.6, and p7zip 9.20.

In this experiment, we still use different data windows or block size to compress the same original file "book.htm." We can see the difference of the results between the platforms of x86/64 and ARM.

Table 4 and Figure 14 show the relationship of the compressed file size and the data window/block size, and Table 5 and Figure 15 show the relationship of the compression/decompression time and the data window/block size.

The only difference between Tables 2 and 4 is the size of the file compressed by ComZip. Even the data window or block size is the same; ComZip generates different compressed file. The reason is explained in [5]. ComZip is also a chaotic encryption software. If the same file is compressed by ComZip twice, we will get 2 thoroughly different compressed files. But the difference of the lengths is so tiny that the influence on the compression ratio can be ignored.

This experiment is limited by the platform hardware, especially the 1 GB RAM. When the block size is enlarged to 64 MB, ComZip with CZ-BWT aborts for insufficient RAM. Both the bucket array and the operating system occupy extra RAM; thus, the total RAM capacity of 1 GB is inadequate. If the RAM is enlarged to 2 GB, we estimate that the workable block size may reach 300 MB.

Figure 15 shows bzip2 is much slower than the others, and its curve also raises rapidly. ComZip with CZ-BWT is faster than p7zip when their data window/block size is between 2 and 8 MB and slower than p7zip in the other cases.

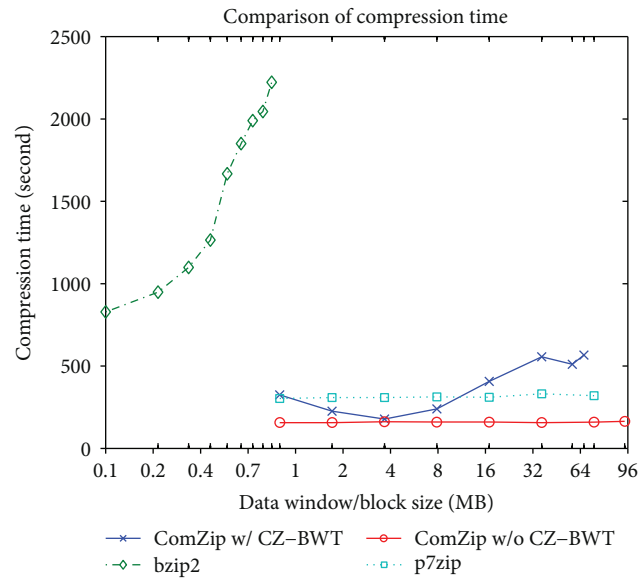

Figure 15: Compression time and data window/block size in Table 5.

ComZip without CZ-BWT is also the fastest on this platform, but the 4-core CPU limits the performance of the parallel CZ encoding pipeline.

Above all, the experiment results on this ARM platform also show that the compression speed of ComZip with CZ-BWT is practical. Although the block size is limited by the RAM, ComZip with CZ-BWT has the best compression ratio among these softwares.

Table 6: x86/64: comparison of compressed file (lamp.vdi) size (B).

Data window/block size	ComZip with CZ-BWT filter	bzip2	ComZip without CZ-BWT filter	p7zip
0.1 MB	—	324,209,650	—	—
0.5 MB	—	316,111,009	—	—
0.9 MB	—	309,637,283	—	—
1 MB	309,677,896	—	291,227,528	288,751,149
8 MB	289,778,024	—	247,728,240	244,375,665
64 MB	285,257,552	—	238,947,232	235,146,686
512 MB	194,737,904	—	126,284,264	122,996,451

Table 7: x86/64: comparison of file (lamp.vdi) compression/decompression time (seconds).

Data window/ block size	Compression time				Decompression time			
	ComZip with CZ-BWT filter	bzip2	ComZip without CZ-BWT filter	p7zip	ComZip with CZ-BWT filter	bzip2	ComZip without CZ-BWT filter	p7zip
0.1 MB	—	47	—	—	—	20	—	—
0.5 MB	—	48	—	—	—	20	—	—
0.9 MB	—	63	—	—	—	32	—	—
1 MB	57	—	34	49	74	—	41	19
8 MB	49	—	38	50	69	—	42	17
64 MB	55	—	43	60	77	—	40	15
512 MB	84	—	28	58	104	—	25	8

5.3. Tests with Other Data. The compression ratio of BWT is not always better than the others without BWT. We find that only some kinds of special data fit the BWT compression well. This experiment uses the same x86/64 platform, but the data file is changed into "lamp.vdi," which is a real virtual machine image file of a Linux data partition. The original length of this file is 527,467,008 B.

Table 6 shows the relationship of the compressed file size and the data window/block size, and Table 7 shows the relationship of the compression/decompression time and the data window/block size.

In Table 6, we observe that bzip2 has the lowest compression ratio among these softwares, and ComZip with CZ-BWT has the second lowest compression ratio. In Table 7, we observe that bzip2 and ComZip with CZ-BWT cannot be faster than p7zip and ComZip with CZ-BWT. Thus, the experiment results provide an example that some kinds of data cannot get better compression ratio and speed by using BWT.

This paper focuses on the BWT algorithms. Researchers may use their own data to find what kind of data fit the BWT well.

From all of the above experiment results, we can get some support about the advantages of CZ-BWT: the compression ratio for some kinds of data, and the compression time contrasting to the other universal BWT compression software. And these results provide some references to the performance of CZ-BWT running on x86/64 and ARM platforms, which may infer the feasibilities and practicalities of using CZ-BWT in the future and current sensors.

But these results also reveal that BWT cannot always gain better compression ratio than other compression algorithms. Thus, the BWT filter in ComZip remains alternative. And compared to the standard BWT, CZ-BWT has lower compression ratio, and its decompression is slower. So we regard the elimination of the weaknesses from CZ-BWT as our future work.

6. Conclusions and Future Work

The rapid expansion of IoT leads to numerous sensors, which generate massive data and bring the challenges of data transmission and storage. A valuable way for this requirement is data compression, and BWT can gain good compression ratios for some kinds of data, which can be used in the sensors.

But the problems of BWT in the sensors for big data still exist. Due to the limited computation resources of each sensor, enlarging the BWT block without the rapid decrease of the encoding/decoding speed is a problem. If the sensor needs hardware acceleration for BWT, simplifying the complex BWT to design the hardware is another problem.

To solve these problems, this paper presents CZ-BWT algorithm, a fast algorithm of truncated BWT using bucket sort. CZ-BWT is implemented in the shareware ComZip. It supports the BWT block up to 2 GB currently, and it is easy to support a larger block, which meets the requirements of big data compression.

The analyses indicate that CZ-BWT encoding has the time complexity $O(N)$, and it's faster than the BWT encoding with SA-IS. The space complexity of CZ-BWT encoding is also $O(N)$, and it uses less RAM than that with SA-IS, if the block size is large enough and the RAM for bucket array can be ignored. The primary hardware design of bucket sort

infers that the hardware acceleration for CZ-BWT is relatively easy to realize.

The experiment results support that ComZip with CZ-BWT is obviously faster than bzip2, and it can obtain better compression ratio than bzip2, p7zip, and ComZip without CZ-BWT for some kinds of data. And these results provide references to the performance of CZ-BWT running on x86/64 and ARM platforms, which may infer that using CZ-BWT in the future and current sensors is feasible and practical.

On the other hand, these experiment results also provide the proofs of the weakness analyses in the CZ-BWT. Compared to the standard BWT, CZ-BWT has lower compression ratio, and its decompression is slower. How can the loss of the compression ratio be analyzed for the truncated BWT? Can we enhance the truncated BWT encoding to obtain better compression ratio? Can we change CZ-BWT into standard BWT and keep its advantages? Can we optimize the decompression algorithms of ComZip, especially the CZ-BWT, to get better speed? Solving these problems is the future work.

Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

References

[1] M. Burrows and D. J. Wheeler, *A Block-Sorting Lossless Data Compression Algorithm*, Systems Research Center, Palo Alto, CA, USA, 1994.

[2] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows–Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[3] Y. Liu, T. Hankeln, and B. Schmidt, "Parallel and space-efficient construction of burrows-wheeler transform and suffix array for big genome data," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 13, no. 3, pp. 592–598, 2016.

[4] J. C. Qin and Z. Y. Bai, "Design of new format for mass data compression," *The Journal of China Universities of Posts and Telecommunications*, vol. 18, no. 1, pp. 121–128, 2011.

[5] Q. Jiancheng, L. Yiqin, and Z. Yu, "Parallel algorithm for wireless data compression and encryption," *Journal of Sensors*, vol. 2017, Article ID 4209397, 11 pages, 2017.

[6] A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic coding revisited," *ACM Transactions on Information Systems*, vol. 16, no. 3, pp. 256–294, 1998.

[7] A. Moffat, "Implementing the PPM data compression scheme," *IEEE Transactions on Communications*, vol. 38, no. 11, pp. 1917–1921, 1990.

[8] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[9] S. J. Puglisi, W. F. Smyth, and A. H. Turpin, "A taxonomy of suffix array construction algorithms," *ACM Computing Surveys*, vol. 39, no. 2, p. 4, 2007.

[10] J. Kärkkäinen, P. Sanders, and S. Burkhardt, "Linear work suffix array construction," *Journal of the ACM*, vol. 53, no. 6, pp. 918–936, 2006.

[11] P. Ko and S. Aluru, "Space efficient linear time construction of suffix arrays," *Journal of Discrete Algorithms*, vol. 3, no. 2–4, pp. 143–156, 2005.

[12] G. Nong, S. Zhang, and W. H. Chan, "Two efficient algorithms for linear time suffix array construction," *IEEE Transactions on Computers*, vol. 60, no. 10, pp. 1471–1484, 2011.

[13] N. Timoshevskaya and W. C. Feng, "SAIS-OPT: on the characterization and optimization of the SA-IS algorithm for suffix Array construction," in *2014 IEEE 4th International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, pp. 1–6, Miami, FL, USA, 2014.

[14] U. Baier, *Linear-time suffix sorting – a new approach for suffix array construction, [M.S. thesis]*, Ulm University, Baden-Württemberg, Germany, 2015.

[15] V. Z. Grajeda, C. F. Uribe, and R. C. Parra, "Parallel hardware/software architecture for the BWT and LZ77 lossless data compression algorithms," *Computación y Sistemas*, vol. 10, no. 2, pp. 172–188, 2006.

[16] U. I. Cheema and A. Khokhar, "A high performance architecture for computing Burrows-Wheeler transform on FPGAs," in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pp. 1–6, Cancun, YUC, Mexico, 2013.

[17] M. Deo and S. Keely, "Parallel suffix array and least common prefix for the GPU," *ACM SIGPLAN Notices*, vol. 48, no. 8, p. 197, 2013.

[18] C. H. Chang, M. T. Chou, Y. C. Wu et al., "sBWT: memory efficient implementation of the hardware-acceleration-friendly Schindler transform for the fast biological sequence mapping," *Bioinformatics*, vol. 32, no. 22, pp. 3498–3500, 2016.