International Journal of Computer Information Systems and Industrial Management Applications.
ISSN 2150-7988 Volume 8 (2016) pp. 001–011


# Hardware-Oriented Succinct-Data-Structure for Text Processing Based on Block-Size-Constrained Compression

Hasitha Muthumala Waidyasooriya, Daisuke Ono and Masanori Hariyama

Graduate School of Information Sciences, Tohoku University,
Aoba 6-6-05, Aramaki, Aoba, Sendai, Miyagi,980-8579, Japan
{*hasitha, ono1831, hariyama*}*@ecei.tohoku.ac.jp*

***Abstract***: **Succinct data structures are introduced to efficiently solve a given problem while representing the data using as little space as possible. However, the full potential of the succinct data structures has not been utilized in software-based implementations due to the large storage size and the memory access bottleneck. This paper proposes a hardware-oriented data compression method based on clustering of blocks. We use a parallel processing architecture to reduce the decompression overhead. According to the evaluation, we achieve 73% and 27% of storage size and memory access reductions respectively.**
***Keywords***: Succinct data structures, data compression, text-search, FPGA, big-data.

## I. Introduction

Succinct data structures are introduced in [1, 2] to efficiently solve a given problem while representing the data using as little space as possible. In early days, succinct data structures were not popular and not practically beneficial due to the lack of large storage devices and no demand for big-data processing. However, recent developments in big-data processing and large storage devices have re-focused the attention to succinct data structures. When using succinct data structures, we can compute most operations in constant time, irrespective of the data size. That means, the benefits of succinct data structures increase with the data size, which is the ideal situation for big-data applications. Moreover, succinct data structures can be used in many fields such as data mining [3, 4], information retrieval [5, 6], graph processing [7, 8], bio-informatics [9, 10], etc. The storage space required by the succinct data structures is close to the information theoretic lower bound. In many cases, the storage space grows linearly (O*(n)* storage space for $n$ bits of data) with the data size, so that they can be used in practical applications.
The two main operations of the succinct data structures are called $rank$ and $select$. The operation $rank_q(B, x)$ returns the number of "$q$"s from a data set $B$ up to the index $x$. The operation $select_q(B, x)$ returns the index, where the $x^{th}$ "symbol $q$" exists in $B$. The symbol $q$ could be a number, a character, a bit, a byte, etc, and the data set $B$ contains many such symbols. The main difference of the succinct data structures compared to the traditional compressed data structures, is related to the processing time. In traditional compressed data structures, the operations need not be supported efficiently, so that the processing time of the operations increases with the data size. However, in succinct data structures, the operations are supported efficiently so that the processing times required for the $rank$ and $select$ operations are independent of the data size. Since the other operations on succinct data structures are defined by a combination of $rank$ and $select$ operations, their processing times are also independent of the data size. This interesting behavior has attracted by the recent big-data applications. Note that, the processing time mentioned here is the time required to do a particular operation and not the total processing time of an application. Examples of succinct data structures are "level-order unary degree sequence (LOUDS)" [2], "balanced parentheses" [11], "depth first unary degree sequence (DFUDS)" [12], etc. However, to compute $rank$ or $select$ in constant time, the implementation of the data structure must be efficient. Depending on the implementation, we may require additional storage or additional processing time. The main problem of the implementation is to minimize the additional data overhead while minimizing the processing time required to compute $rank$ and $select$.
Although the processing time using many succinct data structures is a constant, this constant could take any value such as few clock cycles or thousands of clock cycles, depending on the implementation. Similarly, the actual storage size can be many times larger than the original data size. Although the data compression is a promising way to solve these problems, the decompression overhead of such a large amount of data could be huge so that the decompression is impractical. Due to such limitations, the full potential of the succinct data structures has not been exploited in software-based applications that use general purpose CPUs.
On the other hand, hardware-based implementations have a huge potential to get the maximum benefits from the succinct data structures. Parallel processing can be used not only to increase the processing speed, but also to handle compressed data efficiently. We propose a hardware-oriented succinct data structure where the data are compressed to reduce the storage space and data access time. The proposed method is based on block-based compression where a given data set is

| Position | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Symbol | A | B | B | A | B | A | C | A | C | C | C | B |

(a) A text data set.

| Position | rank_A | rank_B | rank_C |
|---|---|---|---|
| 1 | 1 | 0 | 0 |
| 2 | 1 | 1 | 0 |
| 3 | 1 | 2 | 0 |
| ... | ... | ... | ... |
| 11 | 4 | 3 | 4 |
| 12 | 4 | 4 | 4 |

(b) $rank$ result table.

**Figure. 1**: $rank$ operation on a text data set.

divided in to blocks. Usually, different blocks have different compression ratios. Since the hardware is designed considering the worst case, the data compression is restricted by the least compressible block. This has been the main problem in our previous works in [13] and [14]. To solve this problem, we proposed a compression method based on block clustering in [15]. The proposed method assigns the blocks in to clusters in such a way that we can achieve a near-uniform compression among all blocks. Therefore, the worst case improves and gets closer to the average case.

This paper is as extension of the work done in [15]. We extend the idea of compression based on block clustering to be used in different data sets that have different sizes of alphabets. We propose two compression methods for the data sets with small and large alphabets. We performed experiments on different data samples to evaluate the proposed methods in terms of data compression and memory access using an FPGA (Field programmable gate array) based hardware platform. Note that, an FPGA is a reconfigurable LSI that contains millions of programmable logic gates [16]. According to the evaluation, we were able to reduce the storage size by over 73% compared to the conventional methods. Moreover, by compressing data, we can reduce the memory access time by over 27%.

## II. Related works

In this paper, we consider succinct data structures for text processing, such as text search and statistical analysis. One of the most popular text search method is based on FM-index [17]. It is widely used in genome sequence alignment [9, 18] which is also a text search. In this method, the Burrows-Wheeler (BW) transform [19] is applied to text data. The $rank$ operation is mainly used for text search and statistical data analysis. The details of text search is discussed in [17] or our previous work in [13]. Figure 1 shows an example of the $rank$ operation. Figure 1(a) is the text data set and figure 1(b) is the $rank$ result table. The succinct implementation of the $rank$ operation is important for text processing applications.

### *A. Implementations of succinct data structures on CPUs using software*

In this section, we discuss the implementation of succinct data structures on general purpose processor based systems using software. A method to compute $rank$ in a constant time is proposed in the initial work of succinct data structures [2]. In this proposal, a two-level block structure is built for a given binary sequence $B[0, n-1]$ of size $n$. The first level contains large blocks of size $log_2 n \times log_2 n$. For each large block, the $rank$ of the first entry is stored in a separate array. This requires $n/log_2 n$ storage size. Each large block is divided in to small blocks of size $\frac{1}{2}log_2 n$. Therefore, each large block contains $2log_2 n$ small blocks. Within a large block, another array is used to store the rank of the first entry of all small blocks. For all large blocks, this array requires $4nlog_2(log_2 n)/log_2 n$ bits. A look-up table is used to store the answer to every possible $rank$ on a bit string of size $\frac{1}{2}log_2 n$. It requires $2^{\frac{1}{2}log_2 n} \times \frac{1}{2}log_2 n \times log_2(\frac{1}{2}log_2 n)$ bits. All arrays and tables can be implemented using O($n$) bits, and it supports $rank$ operation in a constant time. Please refer [2] and [15] for more details. Since we use many arrays and tables, this method needs multiple (although a constant number of) memory reads to compute $rank$. Another popular method is proposed in [20] and often called "RRR" encoding. This method is originally introduced for a bit vector. Similar to [2], a two level block structure is used. The small blocks are classified in to different classes according to the number of ones (or zeroes) in their bit vector. For each class, a separate table of $rank$ results at each bit index is stored. In the small blocks, the class number and a pointer to the bit index are stored. The $rank$ operation is done by adding the global rank from the large block and the local rank from the table entry which is find easily by referring the pointers stored in the small blocks. Although these methods are initially proposed for bit vectors, they are later extended to be used with multi-bit words.

These methods are mainly used in software that use general purpose CPUs. They require multiple memory accesses when the data size increases. As a result, the performances are often limited by the memory bandwidth as shown in many applications such as genome sequence alignment [9, 18]. One promising way of reducing the storage and memory access is data compression. However, this is not an effective solution for software, since the decompression is complicated and time consuming. However, this problem can be solved by hardware. Parallel processing in hardware can be used to decrease the decompression overhead greatly. In this paper, we propose a succinct data structure that can be decompressed easily using hardware.

### *B. Implementation of succinct data structures on hardware*

Hardware-based succinct data structures are already used in text search [14, 15] and genome sequence alignment [18, 21], which is basically a text search. Succinct data structures allow efficient search operations on genome data that contain only four symbols. The implementation methods used in those works are based on a two-level block structure as shown in figure 2. The text data set is divided into equal sized blocks. The first level consists of an array of $rank$ computation results where each result corresponds to the first entry

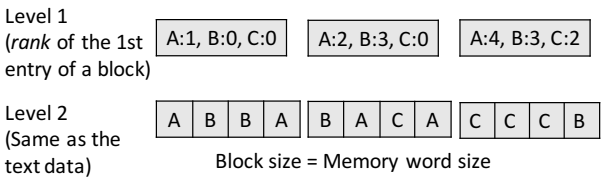



**Figure. 2**: Succinct data structure implementation on hardware.

of each block. The second level consists of multiple blocks, where each block contains fixed amount of words from the text. A popcount (population count) module is used to count the number of appearance of each symbol in the alphabet. The $rank$ operation is done by adding the popcount value of a block in level two to the corresponding $rank$ computation result in level one. For example, to calculate $rank_C(\text{Data set}, 7)$, we access the second block from level two since the seventh word is stored there. We count the number of $C$ symbols up to the third position and add it to the second entry of the level one. Since, the number of $C$ symbols in the second entry is zero, $rank_C(\text{Data set}, 7)$ equals to one.

The storage size is calculated as follows. We consider a data set of $N$ words with an alphabet of $A$ symbols. To store a word, $\lceil log_2(A) \rceil$ bits are required. The block size is given by $B_{size}$. Note that, $B_{size}$ is a given constraint usually decided by the hardware. Each block contains $P$ words, where $P$ equals $B_{size}/\lceil log_2(A) \rceil$. The number of blocks required equals $\lceil N/P \rceil$. The storage size of level two data ($Size_{L2}$) is given by Eq.(1).

$$Size_{L2} = \lceil N/P \rceil \times B_{size} \tag{1}$$

Defining an alphabet $A$ as $\{A_0, A_1, ...\}$, the number of bits required to store the number of $A_i$ (symbol count) is denoted by $(nA)_i$. The number of bits required to store all symbol counts in the alphabet is given by $\Sigma(nA)_i$ bits. This equals to the storage size of a $rank$ operation result. Since we store only the first $rank$ result of a block, the storage size of the data in level one can be determined by Eq.(2).

$$Size_{L1} = \lceil N/P \rceil \times \Sigma(nA)_i \tag{2}$$

Data compression can be used in hardware effectively to reduce the storage size while applying parallel processing to reduce the data decompression time. In our earlier works, we used word-pair encoding [13] to compress the data. Note that, although this method is very similar to the byte-pair encoding (BPE) [22], a pair can have any number of bits unlike 8 bits in BPE. Figure 3 shows an example of the word-pair encoding. The data set is shown in figure 3(a). In this data set, the most common word-pair is replaced by a new word that has not been used before. After replacing the word pair, the next most common word pair is searched and replaced by a new word. This process continues until the end of all available symbols. Figure 3(b) shows the compressed data set. We replaced the word pairs "c,o" and "a,n" by new words "X" and "Y" respectively. In this example, we assume that we can use an alphabet of 16 symbols. Therefore, each symbol is represented by 4 bits. The 112-bit data set is compressed to 84 bits, where the compressed one contains 21

| Position | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Symbol | c | o | c | o | n | u | t | ␣ | a | n | d | ␣ | b | a | n | a | n | a | ␣ | c | o | o | k | i | e | s | . | ␣ |

(a) Data set. Total size = 28 symbols × 4 bits = 112 bits

| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Symbol | X | X | n | u | t | ␣ | Y | d | ␣ | b | Y | Y | a | ␣ | X | o | k | i | e | s | . | ␣ |

Dictionary : co ← X, an ← Y

(b) Compressed data set after word-pair encoding. Total size = 22 symbols × 4 bits = 88 bits

**Figure. 3**: Word pair encoding (WPE).

| Block 1 | | | | | | Block 2 | | | | | | Block 3 | | | | | | Block 4 | | | | | | Block 5 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Position | 1 | 2 | 3 | 4 | 5 | 6 | 1 | 2 | 3 | 4 | 5 | 6 | 1 | 2 | 3 | 4 | 5 | 6 | 1 | 2 | 3 | 4 | 5 | 6 | 1 | 2 | 3 | 4 | 5 | 6 |
| Symbol | c | o | c | o | n | u | t | ␣ | a | n | d | ␣ | b | a | n | a | n | a | ␣ | c | o | o | k | i | e | s | . | ␣ | | |

(a) Data sequence divided in to blocks.

| Block 1 | | | | | Block 2 | | | | | Block 3 | | | | Block 4 | | | | | Block 5 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Position | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 |
| Symbol | X | X | n | u | t | ␣ | Y | d | ␣ | b | Y | Y | a | ␣ | X | o | k | i | e | s | . | |

Dictionary : co ← X, an ← Y

(b) Compressed sequence after word-pair encoding. Total size = 5 blocks × 5 symbols × 4 bits = 100 bits

**Figure. 4**: Block based compression used in [14]. Different blocks have different compression ratio. Hardware is designed for the worst compression ratio.

words. Note that, the space (indicted by "␣") is also considered as a symbol. The compression ratio is 1.33 as calculated according to Eq.(3). In WPE, we have to store the dictionary data which also require storage space.

$$\text{compression ratio} = \frac{\text{Uncompressed size}}{\text{Compressed size}} \tag{3}$$

The advantage of WPE is that we can decompress from any position. However, it has a major disadvantage when accessing a particular data value from the memory. As shown in figure 3, the compression is not uniform in every parts of the data set. That is, some parts are heavily compressed while some other parts are not compressed at all. Therefore, we cannot find a direct relationship between the position of the initial data set and the memory address of the corresponding compressed data are stored. In order to find a particular data value, either we have to decompress the whole compressed data, or we have to maintain a separate index that gives the corresponding positions the words in the original and the compressed data sets. Neither of these methods are efficient.

To solve this problem, we have considered a block-based compression in [14]. Figure 4 gives an example of this method. As shown in figure 4(a), the data set is divided in to multiple blocks of the same size. Then, each block is compressed using WPE as shown in figure 4(b). Now, every block is directly corresponds to its compressed block. For example, the third symbol in the block 3 (that is the symbol "n"), is found in the compressed block 3. Due to the compression, the position of a symbol inside a block has been changed. Therefore, we have to decompress the whole block to find a symbol. However, this decompression overhead is

very small compared to the overhead required to decompress the whole compressed data set. As shown in figure 4, some blocks are easy to compress while the others are difficult. As a result, the compressed data set contains blocks of different sizes. To store these blocks in the memory, we have to consider a constant block size. If we store the blocks with variable sizes, we need additional information such as the block sizes, memory address of each block, etc to access the correct block. Therefore, we chose the largest block size as the constant block size and allocate the same memory capacity for every block, irrespective of their sizes. As a result, although the data are compressed, the compression ratio is small compared to the previous method shown in figure 3.
As shown in the above example, hardware are designed for the worst case where the least compressed block is important. On the other hand, the software are designed for the average case, where the average compression ratio is important. Therefore, to increase the compression ratio in hardware, we have to improve the worst case. In text processing, we often use BW transformed data for text search. Some parts of the BW transformed text contain repeated symbols, while some other parts have different symbols. As a result, it is much harder to achieve a near-uniform compression for BW transformed text. Therefore, providing an efficient solution to this problem is necessary to achieve a good hardware implementation.

## III. Hardware-oriented compressed succinct data structure based on clustering of blocks

In this section, we propose a compressed succinct data structure for hardware. Figure 5 shows the basic hardware structure that consists of a computational module, a memory controller and memory modules. The computation module accesses the memory through the memory controller. When we talk about memory access, we mean the memory controller access while the memory controller takes care of the memory access. In this paper, we use the word memory access instead of memory controller access. The number of bits accessed in one clock cycle is decided by the memory controller bus width. As shown in figure 5, $D_{length}$ bits can be accessed in one clock cycle. In succinct data structures, if the block size is smaller than or equals to $D_{length}$, one memory access is enough to access the data. Otherwise, we need multiple memory accesses which increase the processing time. In the proposed compressed succinct data structure, we consider not only reducing the data size, but also reducing the memory access.
The proposed data structure also uses the same two level block structure explained in section II-B. According to the type of the data, we classify the proposed data compression in to the following two methods.

- Word vector compression.
- Bit vector compression.

The first method is suitable for data sets with small alphabets, such as genome data (which has only four symbols in the alphabet), small-range temperature data etc. According to Eq.(2), level one storage size depends on the number of bits required to store a $rank$ result given by $\Sigma(nA)_i$. For small alphabets, $\Sigma(nA)_i$ is small so that the level one storage size is small. Therefore, the overall storage size is mostly affected by the level two data. The second method is suitable for the data sets with large alphabets, such as text data. For such data sets, the level one data occupy a significant portion of the total storage size. Therefore, we use a different data structure to reduce the level one data. For the data sets that have medium size alphabets, we can use either of these methods considering the pros and cons of each. In next sections, we discuss each of these methods in detail.

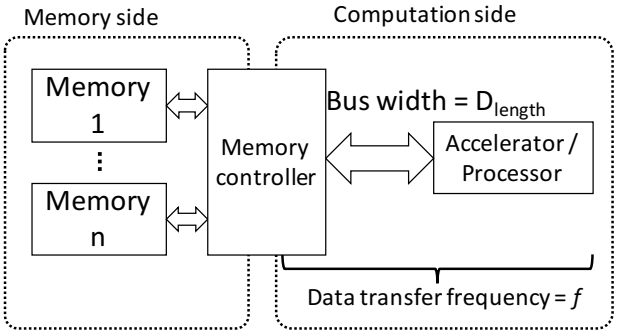



**Figure. 5**: Memory access through the memory controller in hardware. $D_{length}$ bits can be transferred in one clock cycle at frequency $f$.

### A. Word vector compression

A word vector is an array of equal size words. Although we separate word vectors from bit vectors, both are represented by bits. However, in a word vector, a word or a group of bits has a meaning. For example, in a text data set, a word or a group of eight bits represents a character such as an English letter. In a genome data set, a word or a group of two bits represents one of the four nitrogenous bases presents in any living organism. Since a word has a meaning, some words or a group of words appear more frequently than the others. Such data sets can be compressed easily. On the other hand, a group of bits in a bit vector does not have a particular meaning and that is the main reason we call them bit vectors. In this section, we focus on the compression of a word vector, which will be the base for both compression methods.
As we have explained at the beginning of section III, we access $D_{length}$ bits from the memory controller in one clock cycle. It is usually a constant for a given system such as an FPGA board. The proposed succinct data structure is also based on the two-level block structure explained in section II-B. The block size of the compressed data must be a multiple of $D_{length}$. As explained in section II-B, it is difficult to achieve a uniform compression among different blocks. One way of overcoming this problem is to use multiple dictionaries. However, it is impractical to use separate dictionaries for every block, since that will increase the dictionary size. Better way is to use a small number of dictionaries by sharing one dictionary among many blocks. However, without calculating the compression ratio, we do not know which dictionary should be applied for a particular block.
Therefore, we consider grouping similar blocks together and

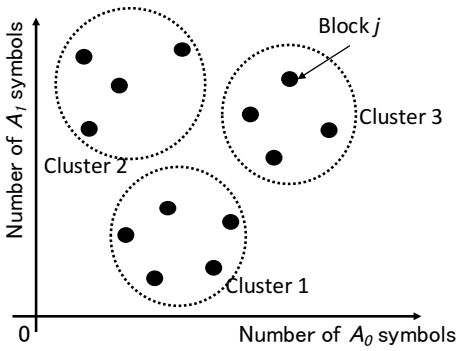


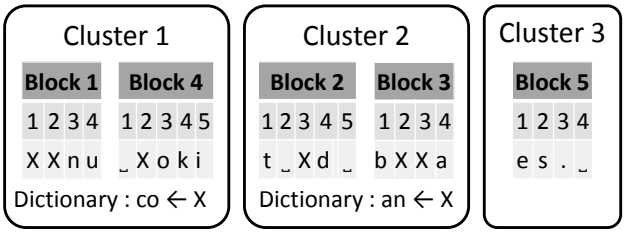

**Figure. 6**: Clustering of blocks.

apply a separate dictionary for every group. This way, we can reduce the number of dictionaries. To find similar blocks, we use $k$-means clustering [23]. Figure 6 shows an example of block clustering. Considering an alphabet that contains two symbols $A_0$ and $A_1$, the $x$ and $y$ axis show the number of $A_0$ and $A_1$ symbols in a block respectively. Blocks are plotted according to their $A_0$ and $A_1$ symbol counts. In this example, blocks are divided into three clusters. The blocks in cluster 1 contain a lot of $A_0$ symbols while the blocks in cluster 2 contain a lot of $A_1$ symbols. Since the blocks in the same cluster share the same properties, such blocks can be compressed easily using a common dictionary. The less compressible blocks are usually very different from each other. Therefore, such blocks are not belong to the same cluster and are distributed among different clusters. As a result, different clusters have a similar compression ratio. That means the worst case is changed to the average case. Note that, in actual clustering, there are more than two axis depending on the alphabet size. We also used the number of symbol pairs as axis and that also gives similar results. The number of symbol pairs increases exponentially with the alphabet size, so that it takes a lot of processing time for clustering. Therefore, we used the number of symbols instead of symbol pairs to represent the similarity of the blocks.

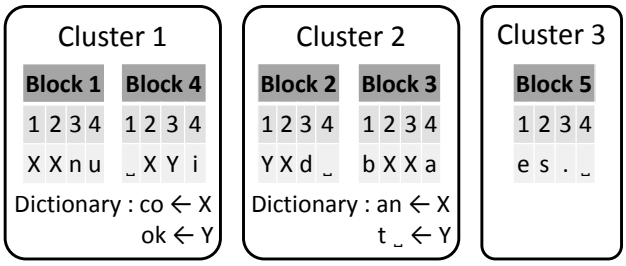
The $k$-means clustering has several problems. One of such is to determine the number of clusters or the value $k$. According to the experimental results, we found that $k$ varies from 100 to 500 depending on the size of the data set. Therefore, by prior experimental knowledge, we can guess the value $k$ by considering the data size. Since the compression is done offline, we can experiment with different $k$ values to find a suitable one. Another problem in the $k$-means clustering is that the clustering result depends on the initialization method. To reduce this effect, clustering starts with a larger $k$ value and we later reduce the number of clusters by merging the similar ones.

Figure 7 shows an example of the proposed succinct data structure. The word vector is divided in to blocks of 6 symbols similar to the example in figure 4. The blocks 1 and 4 are assigned to the cluster 1, the blocks 2 and 3 are assigned to the cluster 2 and the block 5 is assigned to the cluster 3. The data compression is explained as follows.

**Step 1:** As shown in figure 7(a), the symbol pair "c,o" is replaced by the new symbol "X" in cluster 1 and the symbol pair "a,n" is replaced by the new symbol "X" in cluster 2.




(a) Encoding in step 1. Storage size= 5 blocks × 5 symbols × 4 bits = 100 bits




(b) Encoding in step 2. Storage size = 5 blocks × 4 symbols × 4 bits = 80 bits

**Figure. 7**: Clustering of blocks. Different blocks have a similar compression ratio.

**Step 2:** As shown in figure 7(b), the symbol pair "o,k" is replaced by the new symbol "Y" in cluster 1 and the symbol pair "t,␣" is replaced by the new symbol "Y" in cluster 2.

After Step 1, we used only 15 symbols in clusters 1 and 2. Since we are allowed 16 symbols we can use another symbol in each cluster. We use the new symbol to compress the blocks with low compression ratios such as the blocks 2 and 4, where each contains 5 symbols compared to 4 in the other blocks. After Step 2, all the blocks contain only 4 symbols. That is, the worst case is reduced from 5 symbols to 4 symbols. As a result, we can improve the compression ratio in hardware.

The compressed data size is calculated as follows. We assume that a compressed block can be decompressed in to $Q$ words. That is, we initially divide the word vector in to blocks by assigning each $Q$ words to a new block. At this stage, the size of a block exceeds $B_{size}$. However, After the compression, the sum of the largest block size and the flag bits is smaller than $B_{size}$. Note that, in every block, we keep a flag that shows the assigned cluster number. The total number of blocks equals $\lceil N/Q \rceil$. The level two storage size ($Size_{L2(word)}$) is given by Eq.(4).

$$Size_{L2(word)} = \lceil N/Q \rceil \times B_{size} \tag{4}$$

The level one storage size ($Size_{L1(word)}$) is given by Eq.(5).

$$Size_{L1(word)} = \lceil N/Q \rceil \times \Sigma(nA)_i \tag{5}$$

Note that, although the alphabet is changed after compression, the level one data use the original alphabet. This is not a problem since the $rank$ is computed after completely decompressing a block. After the decompression, the data contain only the original alphabet.

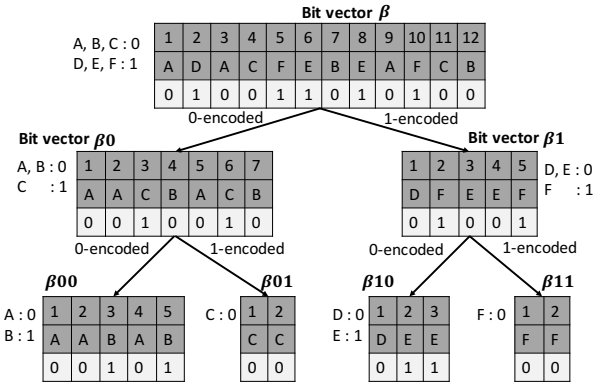
In this method, we directly compress the level two data using WPE. Due to this compression, we can pack more words into one block. Since one block contains more words, we can reduce the number of blocks. Note that, the number of blocks $\lceil N/Q \rceil$ in Eq.(5) is smaller than the number of blocks $\lceil N/P \rceil$in Eq.(2). Since the level one storage size depends on the number of blocks, it also reduces even though we have not performed any direct data compression on it. Moreover, from Eqs.(1) and (4) and also from Eqs.(2) and (5), we can say that the compression ratios of both level two and level one are the same.

### B. Bit vector compression

In the bit vector compression, we use the wavelet tree structure with block clustering. The wavelet tree was introduced in [24] as a data structure to represent a sequence and answer some queries on it. Wavelet tree represents any word vector in a hierarchy of bit vectors. The number of bit vectors in the hierarchy equals to $log_2(A)$, where $A$ is the size of the alphabet. Figure 8 shows an example of a wavelet tree. Figure 8(a) shows the wavelet tree representation of a word vector. The word vector is represented by zeros and ones in the top bit vector $\beta$, where the first half of the alphabet is encoded as 0 and the second half as 1. In the bit vector $\beta 0$, the 0-encoded words are re-encoded as 0 and 1. The same process is applied for the 1-encoded words in the bit vector $\beta 1$. We apply this process repeatedly for all the bit vectors until there are only one or two symbols left. Figure 8(b) shows an example of a $rank$ computation. To compute $rank_E(\text{word vector}, 10)$, we first consider the top bit vector $\beta$. Since $E$ is 1-encoded, we compute $rank_1(\beta, 10)$. From the 1-encoded words in $\beta 1$, $E$ is re-encoded as 0. Therefore, we compute $rank_0(\beta 1, 5)$. Then, $E$ is re-encoded as 1 in $\beta 10$, so that, we compute $rank_1(\beta 10, 3)$. Since this is the end of the wavelet tree, $rank_E(\text{word vector}, 10) = rank_1(\beta 10, 3) = 2$.

As shown in figure 8a, if the initial word vector has $N$ words, the bit vectors in each stage contains $N$ bits each. Since there are $\lceil log_2(A) \rceil$ stages, the storage size of the wavelet tree is $N \times log_2(A)$ which equals to the size of the initial word vector. We apply the two level block structure to the wavelet tree. As explained in section II-B, the $rank$ results of the first entry of a block is stored in level one. As shown in Eq.(2), the storage size of the level one increases with the alphabet size $A$. Since the alphabet size of the bit vectors is two (only 0 and 1), the storage size is significantly small. Also note that, although the alphabet size is two, we store the $rank$ results of either 0 or 1. This reduces the storage size by further 50%. We can compute the $rank$ result of the other symbol by subtracting the $rank$ result of one symbol from the index ($rank_1 = index - rank_0$). However, to compute $rank$, we have to access several bit vectors. As a result, the memory access will increase.

Using wavelet trees, the level one data size is reduced by nearly $\Sigma(nA)_i$ from Eqs.(2) since we store the $rank$ results of only one symbol. The level two data size is not reduced, since the sum of bits in all bit vectors equals to the size of the word vector. To compress the level two data, we use the same WPE-based compression on bit vectors. As explained in section III-A, a group of bits in a bit vector does not have a strong meaning like in a word vector. However, for the applications such as text search, we use BW-transformed text. After BW transform, we can see patterns such as a series of zeros or ones. Therefore, we can compress the bit vectors by grouping few bits into words and applying WPE. However, we cannot expect a large compression ratio for the level two data since bit vectors are more random compared to the word vectors.




(a) Wavelet tree construction from the word vector "ADACFEBEAFCB"

Calculate $rank_E(\text{word vector}, 10)$

Step 1: Calculate $rank_1(\beta, 10)$=5

Step 2: Calculate $rank_0(\beta 1, 5)$=3

Step 3: Calculate $rank_1(\beta 10, 3)$=2

(b) $rank$ calculation using wavelet tree

**Figure. 8**: Wavelet tree

Note that, although the discussion of bit vector compression is done using wavelet trees, we use wavelet matrix [25] to implement the bit vectors. Wavelet matrix is almost the same as wavelet trees. However, it is more practical since it has continuous and the same size bit vectors in each stage unlike in a wavelet tree that has separated bit vectors as shown in figure 8(a). Interested readers can follow the works in [25, 26] that give a comprehensive discussion on wavelet matrix.

### C. Implementation on hardware

The overall hardware architecture based on the FPGA is shown in figure 9. In the hardware implementation, we consider that the compressed data are stored in the DRAM and accessed by the FPGA hardware. The dictionaries of different clusters are stored in the on-chip memory of the FPGA. The accessed data are decompressed and sent to the processing elements for parallel processing. The DRAM is accessed through a memory controller.

Figure 10 shows the format of the code word of the compressed data in level two. It consists of the cluster number and the compressed data. The cluster number points to the dictionary of the corresponding cluster. Since the dictionary data is small, we store those in the on-chip memory. All symbols in the compressed data set are of the same size. However, they can represent one or more words in the original data set. Figure 11 shows the architecture of the decompression

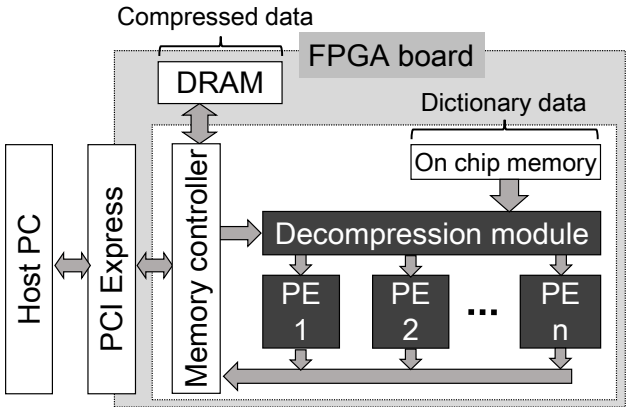



**Figure. 9**: FPGA based system architecture

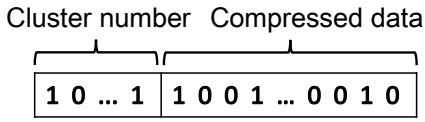



**Figure. 10**: Compressed data format

module. In the decompressing process, one block is fully decompressed in to its original words by referring the dictionary. The correct dictionary is selected according to the cluster number. The decompressing of each word is done in parallel so that the processing time overhead is minimized. After the decompression is completed, a popcount hardware is used to count the number of symbols in the alphabet. The $rank$ or $select$ operations are computed by adding the popcount value to its corresponding $rank$ result stored in the level one data.

## IV. Evaluation

The evaluation is done using an FPGA board called "DE5" [27]. It contains an "Altera 5SGXEA7N2F45C2 FPGA" and two 2GB DDR3-SDRAMs. The memory controller produces 512 bit large data in one clock cycle. The $k$-means clustering is done using Matlab R2014b, and it takes around 10 minutes to complete. The WPE is applied using a c-program compiled by gcc compiler. The compression takes less than one hour on an Intel Xeon E5-2643 CPU and CentOS 6.6 operating system.

All the text data are used after doing the BW transform [19], so that we can use them for text search. Temperature data

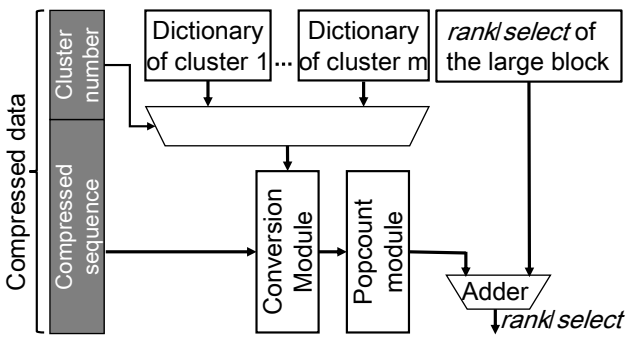
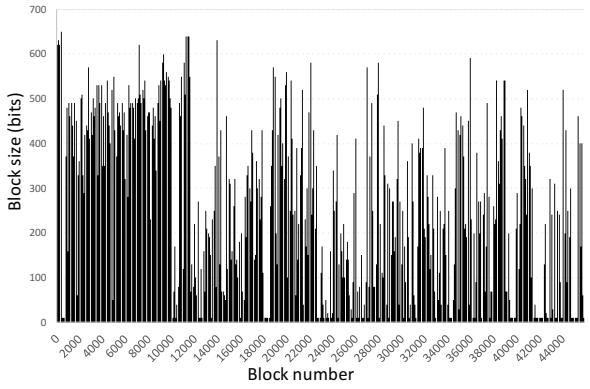
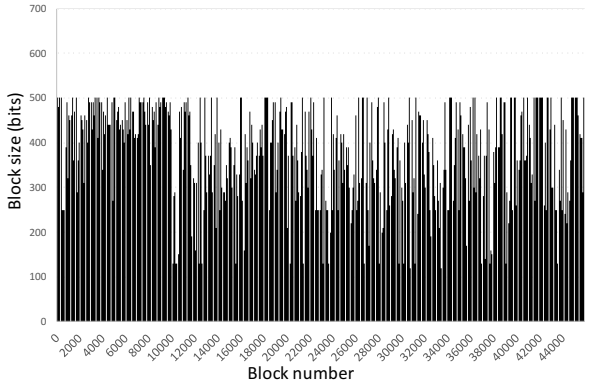



**Figure. 11**: Architecture of the decompression module.




(a) Conventional data compression using a common dictionary.




(b) proposed method based on clustering of blocks.

**Figure. 12**: Block size distribution after compression.

(which are also represented as texts) used in this evaluation are not BW transformed. Those data are mainly used for the statistical analysis which does not require BW transform. In the proposed compression, we have to store the cluster number of each block. That requires $log_2(\text{number of clusters})$ bits per block. We consider this overhead also in the evaluation. For example, 512 bit block may consists of 500bits and 12bits for the compressed data and the cluster number respectively.

### A. Evaluation of the data compression

Figure 12 shows the block size distribution after compression. Figure 12(a) shows the distribution after the conventional compression method [14] that uses a common dictionary. As we can see, some blocks are very small while some others are very large. Figure 12(b) shows the distribution after the proposed compression that uses cluster-based multiple dictionaries. In the proposed method, there are no extremely large or small blocks. The worst block size is close to the average size. This method is better for hardware since it reduces the worst case.

Table 1 shows the comparison of the proposed method compared to the other methods. The proposed method uses word vector compression. The sample data sets contain the average daily temperature data of different cities [28]. According to the comparison, the proposed method has a larger compression ratio than that of the conventional method [14]. These data samples use a small alphabet. As a result, the lev-

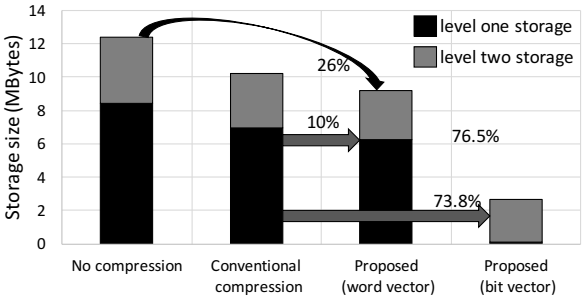



**Figure. 13**: Storage size comparison.

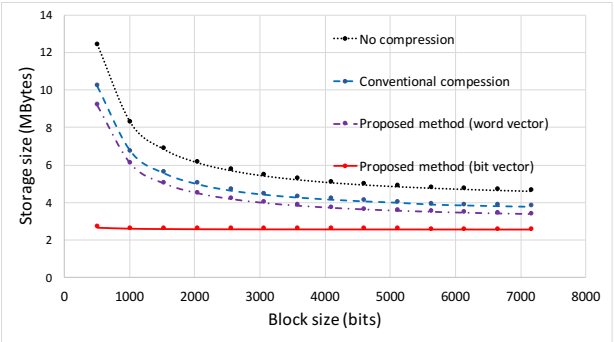



**Figure. 14**: Storage size against block size.

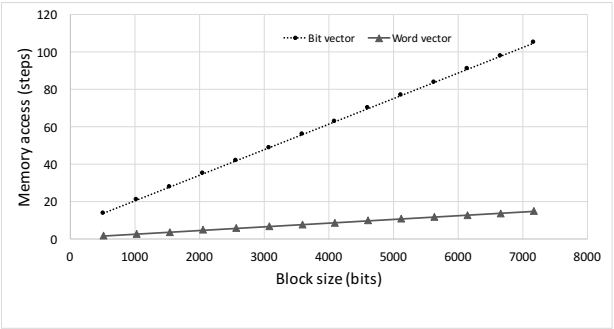



**Figure. 15**: Number of memory access for a single data value.

el two storage is the largest component in the total storage size.

Table 2 shows the word vector compression using data sets with large alphabets. We use text data from the Bible [29] and Wikipedia [30]. Although the data are compressed, the compression ratio is smaller compared to that in Table 1. The dictionary size does not affect the compression ratio, since it is very small compared to the data size. When the alphabet is large, the level one storage is much larger than the level two storage. To achieve a large compression ratio, we have to reduce the level one storage size. Note that, the dictionary size is excluded from the total storage in Table 2, since it is stored separately in the internal memory for fast access.

To reduce the level one storage size, we use the bit vector compression. Figure 13 shows the comparison of the bit vector compression with other methods. This method gives a large compression ratio for level one storage. Since the level one storage is the largest component, the storage size reduction is over 73% compared to the conventional methods.

Another way of reducing the level one storage is to increase the block size. However, if the block size is larger than $D_{length}$, we have to access the memory many times. Figure 14 shows the storage size for larger block sizes. If the block size is large, the number of block are small. As shown in Eq.(2) level one storage size depends on the number of blocks. Therefore, level one storage is reduced by reducing the number of blocks. Note that, in the bit vector implementation, level one storage is very small even for small block sizes. Therefore, increasing the block size does not show any significant reduction in the total storage size. Also note that, using large blocks does not guarantee a level two storage reduction. However, large blocks are easy to compress in practice since they contain more patterns compared to small blocks. Also note that, even though the storage size is reduced in word vector compression by increasing the block size, the bit vector compression gives the largest reduction.

We can also reduce the storage size by using more clusters. Table 3 shows the relationship between the number of clusters and the compressed data size. However, there is a limit to this reduction. Maximum compression for Bible data is achieved by using 100 clusters. Using over 100 cluster has disadvantages such as a large dictionary size, large compression time, large decompression hardware, etc. Therefore, it is always better to use small number of clusters. However, in $k$-means clustering, we have to provide $k$ in order to start the clustering. According to the experimental results, we found that the number of clusters are quite similar for the same types of data. For example, compressing of text data requires around 100 to 500 clusters. Small data sets at the range of few megabytes require around 100 clusters, while large data sets of over few gigabytes require around 500 clusters. Moreover, the data compression is done offline and it takes less than an hour of processing time. Therefore, we can examine different cluster sizes to find a near optimal cluster number. Another method is to start clustering using a large $k$ value. Then, we can merge the similar ones to reduce the number of clusters.

### *B. Evaluation of the memory access time reduction*

In section IV-A, we have shown that the bit vector compression gives a significantly large compression ratio compared to word vector compression. However, such advantages are accompanied with a significant draw back. That is the large memory access time. Figure 15 shows the number of memory accesses required to access one data value. The memory access of the bit vector compression is significantly larger than that of the word vector compression. This means, we have to trade memory access time to achieve large data compression ratio.

However, if we can cache the already accessed data, we can reduce the memory access time. Figure 16 shows the memory access percentage assuming that we can use cache. If we compress the data, we can retrieve more data in one memory access. That means, if the compression ratio is large, there is a better chance to hit the cache. Note that, we store the data in cache in the compressed format, so that no additional cache memory is required. The proposed word vector compression reduces the memory access by 27% and 9.88% compared to

*Table 1*: Word vector compression using data sets with small alphabets.

| Data set | No compression Storage (Bytes) | | | Conventional method [14] Storage (Bytes) | | | Compression | Proposed method Storage (Bytes) | | | Compression |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | level 1 | level 2 | Total | level 1 | level 2 | Total | ratio | level 1 | level 2 | Total | ratio |
| USA - Hawaii | 975 | 3840 | 4815 | 732 | 2880 | 3612 | 1.33 | 618 | 2432 | 3050 | 1.58 |
| Singapore | 675 | 3840 | 4515 | 473 | 2688 | 3161 | 1.43 | 428 | 2432 | 2860 | 1.58 |
| Indonesia - Jakartha | 900 | 3840 | 4740 | 675 | 2880 | 3555 | 1.33 | 570 | 2432 | 3002 | 1.58 |
| Sri Lanka - Colombo | 825 | 3840 | 4665 | 633 | 2944 | 3577 | 1.30 | 523 | 2432 | 2955 | 1.58 |

*Table 2*: Word vector compression using data sets with large alphabets.

| | No compression (Storage MB) | | | Proposed method (Storage MB) | | | | |
|---|---|---|---|---|---|---|---|---|
| | level 1 | level 2 | Total | level 1 | level 2 | Total | Compression ratio | Dictionary (storage MB) |
| Bible (4.37MB) | 8.04 | 3.78 | 11.82 | 5.95 | 2.80 | 8.75 | 1.35 | 0.02 |
| Wikipedia (4MB) | 3.70 | 3.01 | 6.71 | 3.15 | 2.56 | 5.71 | 1.18 | 0.15 |
| Wikipedia (8MB) | 16.30 | 7.01 | 23.32 | 11.90 | 5.12 | 17.02 | 1.37 | 0.26 |
| Wikipedia (24MB) | 26.82 | 18.07 | 44.89 | 22.80 | 15.36 | 38.16 | 1.18 | 0.39 |
| Wikipedia (48MB) | 66.71 | 36.14 | 102.85 | 56.70 | 30.72 | 87.42 | 1.18 | 0.83 |

*Table 3*: Number of clusters vs. compression ratio. Data sample is Bible data.

| Compressed block size (bits) | Number of clusters | Compressed data size (MB) | Dictionary size (kB) |
|---|---|---|---|
| 540 | 10 | 9.41 | 4.6 |
| 530 | 30 | 9.21 | 8.2 |
| 510 | 80 | 8.84 | 14.4 |
| 500 | 100 | 8.75 | 20.1 |
| 500 | 130 | 8.75 | 21.4 |
| 500 | 500 | 8.75 | 53.4 |
| 500 | 1000 | 8.75 | 74.8 |

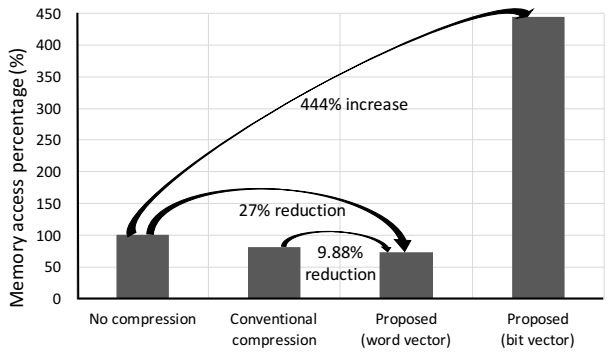



**Figure. 16**: Aveage memory access for a large amount of data. Bloack size is 512 bits

the no-compression and the conventional compression [14] methods respectively. However, for the bit vector compression, the memory access is increased significantly, although it provides a large data compression ratio.

## V. Conclusion

In this paper, we propose a hardware-oriented succinct data structure. The proposed method is based on block-based compression where a given data set is divided in to blocks. Since the hardware are designed considering the worst case, the data compression is restricted by the least compressible block. To solve this problem, we assign the blocks to clusters where all clusters have a similar compression ratio. The least compressible blocks are distributed among multiple clusters in such a way that their compression ratio is improved. The proposed data compression method is evaluated considering an FPGA based hardware platform.

We proposed two method, word vector and bit vector compression for data sets with small and large alphabets respectively. Bit vector compression reduces the storage size significantly at the cost of large memory access time. Irrespective of the disadvantages, it is hugely beneficial for the data sets with large alphabets due to the significantly large data compression ratio. For the data sets with smaller alphabets, it is better to use word vector compression, since it reduce both the data size and the memory access time. For the data sets with moderate alphabet sizes, we can use either of the two methods weighing each method's advantages and disadvantages. We can also use word vector compression with large block sizes to achieve a reasonable data compression with small increase of memory access time.

## Acknowledgment

This work is supported by MEXT KAKENHI Grant Numbers 24300013 and 15K15958.

## References

[1] G. Jacobson, *Succinct static data structures*. PhD thesis, Carnegie Mellon University Pittsburgh, 1988.

[2] G. Jacobson, "Space-efficient static trees and graphs," in *Proc. 30th Annu. Symp. Foundations of Computer Science*, pp. 549–554, 1989.

[3] S. Jonassen, "Large-scale real-time data management for engagement and monetization," in *Proc. 2015 Workshop on Large-Scale and Distributed System for Information Retrieval*, pp. 1–2, 2015.

[4] G. Navarro and S. V. Thankachan, "Bottom-k document retrieval," *Journal of Discrete Algorithms*, vol. 32, pp. 69–74, 2015.

[5] K. Sadakane, "Succinct data structures for flexible text retrieval systems," *Journal of Discrete Algorithms*, vol. 5, no. 1, pp. 12–22, 2007.

[6] S. Kim and H. Cho, "A new approach for approximate text search using genomic short-read mapping model," in *Proc. Inter. Conf. Ubiquitous Information Management and Communication*, pp. 58:1–58:8, 2015.

[7] J. Fischer and D. Peters, "A practical succinct data structure for tree-like graphs," *WALCOM: Algorithms and Computation*, vol. 8973, pp. 65–76, 2015.

[8] Y. Tabei and K. Tsuda, "Kernel-based similarity search in massive graph databases with wavelet trees," in *Proc. Custom Integrated Circuits Conference*, pp. 154–163, 2011.

[9] H. Li and R. Durbin, "Fast and accurate short read align- ment with burrows-wheeler transform," *Bioinfomatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[10] S. Grabowski, S. Deorowicz, and L. Roguski, "Disk-based genome sequencing data compression," *Bioinformatics*, vol. 31, no. 9, pp. 1389–1395, 2015.

[11] J. I. Munro and V. Raman, "Succinct representation of balanced parentheses, static trees and planar graphs," in *Proc. 38th Annu. Symp. Foundations of Computer Science*, pp. 118–126, 1997.

[12] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao, "Representing trees of higher degree," *Algorithmica*, vol. 43, no. 4, pp. 275–292, 2005.

[13] H. Waidyasooriya, D. Ono, M. Hariyama, and M. Kameyama, "Efficient data transfer scheme using word-pair-encoding-based compression for large-scale text-data processing," in *Proc. IEEE Asia Pacific Conf. Circuits and Systems (APCCAS)*, pp. 639–642, 2014.

[14] H. M. Waidyasooriya, D. Ono, M. Hariyama, and M. Kameyama, "An fpga architecture for text search using a wavelet-tree-based succinct-data-structure," in *Proc. Inter. Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pp. 354–359, 2015.

[15] H. M. Waidyasooriya, D. Ono, and M. Hariyama, "Hardware-oriented succinct-data-structure based on block-size-constrained compression," in *Proc. 7th Inter. Conf. Soft Computing and Pattern Recognition (SoCPaR)*, pp. 136–140, 2015.

[16] W. Carter, K. Duong, R. H. Freeman, H. Hsieh, J. Y. Ja, J. E. Mahoney, L. T. Ngo, and S. L. Sze, "A user programmable reconfigurable gate array," in *Proc. Custom Integrated Circuits Conference*, pp. 233–235, May 1986.

[17] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proc. 41st Annu. Symp. Foundations of Computer Science*, pp. 390–398, 2009.

[18] H. Waidyasooriya and M. Hariyama, "Hardware-acceleration of short-read alignment based on the burrows-wheeler transform," *IEEE Trans. Parallel and Distributed Systems*, 2015.

[19] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," tech. rep., Digital Equipment Corporation, 1994.

[20] R. Raman, V. Raman, and S. S. Rao, "Succinct indexable dictionaries with applications to encoding k-ary trees and multisets," in *Proc. 13th Annu. ACM-SIAM Symp. Discrete Algorithms*, pp. 233–242, 2002.

[21] Y. Xin, B. Liu, B. Min, W. X. Y. Li, R. C. C. Cheung, A. S. Fong, and T. F. Chan, "Parallel architecture for dna sequence inexact matching with burrows-wheeler transform," *Microelectronics Journal*, vol. 44, no. 8, pp. 670–682, 2013.

[22] P. Gage, "A new algorithm for data compression," *C Users J.*, vol. 12, no. 2, pp. 23–38, 1994.

[23] J. Macqueen, "Some methods for classification and analysis of multivariate observations," in *Proc. 5th Berkeley Symp. Mathematical Statistics and Probability*, pp. 281–297, 1967.

[24] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proc. 14th Annu. ACM-SIAM Symposium on Discrete Algorithms*, pp. 841–850, 2003.

[25] F. Claude and G. Navarro, "The wavelet matrix," in *Proc. 19th Inter. Symp. String Processing and Information Retrieval (SPIRE)*, pp. 167–179, 2012.

[26] F. Claude, G. Navarro, and A. Ordóñez, "The wavelet matrix: An efficient wavelet tree for large alphabets," *Information Systems*, vol. 47, pp. 15–32, 2015.

[27] "Altera development and education boards." `https://www.altera.com/support/training/university/boards.html#de5`.

[28] "Average daily temperature archive. the university of dayton." `http://academic.udayton.edu/kissock/http/Weather/`.

[29] "The king james version of the bible." `http://www.gutenberg.org/ebooks/10`.

[30] "Wikipedia." `https://en.wikipedia.org`.

## Author Biographies

**Hasitha Muthumala Waidyasooriya** received the B.E degree in Information Engineering, M.S degree in Information Sciences and Ph.D. in Information Sciences from Tohoku University, Japan, in 2006, 2008 and 2010 respectively. He is currently an Assistant Professor with the Graduate School of Information Sciences, Tohoku University. His research interests include reconfigurable computing, processor architectures for big-data processing and high-level design methodology for VLSIs.

**Daisuke Ono** received the B.E degree in Information Engineering from Tohoku University, Japan, in 2014. He is currently an M.S. student with the Graduate School of Information Sciences, Tohoku University. His research interests include text data compression methods for hardware.

**Masanori Hariyama** received the B.E. degree in electronic engineering, the M.S. degree in information sciences, and the Ph.D. degree in information sciences from Tohoku University, Sendai, Japan, in 1992, 1994, and 1997, respectively. He is currently an Associate Professor with the Graduate School of Information Sciences, Tohoku University. His research interests include real-world applications such as robotics and medical applications, big data applications such as bio-informatics, high-performance computing, VLSI computing for real-world application, high-level design methodology for VLSIs, and reconfigurable computing.