Hardware-Oriented Succinct-Data-Structure 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
Email: {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 have 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 to reduce the storage space without increasing the processing time. We use a parallel processing architecture to reduce the decompression overhead. According to the evaluation, we can compress the data by 37.5% and still have fast data access with small decompression overhead.**

***Index Terms*—Succinct data structures, data compression, text-search, FPGA, big-data.**

I. Introduction

Succinct data structures are first introduced in 1989 [1] to efficiently solve a given problem while representing the data using as little space as possible. During that time, 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 the data size grows, we always require a larger storage device. As a result, on-chip memories are replaced by off-chip memories (such as DRAMs), DRAMs are replaced by SSDs and SSDs are replaced by hard disks, etc. In each replacement, the access time increases by over ten times. As a result, the processing time increases exponentially with the amount of data.

Let us see why succinct data structures can solve the problems in big-data applications. When using many 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, image processing, computer vision, bio-informatics, numerical simulations, 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.

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. Moreover, the actual storage size is $k \times n$, where k usually takes a value from tens to thousands. As a result, the storage size of the succinct data structures can be many times larger than the original data size. Since recent computers have a very large memory capacity and extremely large hard disk space, implementing such data structures is possible. However, due to the limitations such as memory access bottlenecks, the full potential of the succinct data structures have not been explored.

In this paper, we propose a hardware-oriented succinct data structure where the data are compressed to reduce the storage space without increasing the processing time. Data compression can solve both storage and memory access problems simultaneously. However, the decompression overhead can be large. Therefore, we use hardware-based parallel processing to reduce this overhead. The proposed method is based on block-based compression where a given sequence is divided in to blocks. Usually, different blocks have different compression ratios. Note that, the compression ratio equals to "data reduction" $\div$ "original data size" $\times 100\%$. Since the hardware is designed considering the worst case, the data compression is restricted by the least compressible block. This has been seen in our previous works in [2] and [3]. The proposed method assigns the blocks in to clusters in such a way that all clusters have similar compression ratios. Therefore, the worst case improves and gets closer to the average case. According to the evaluation on an FPGA (Field programmable gate array), we can compress the data by 37.5% and still have fast data access with small decompression overhead. Note that, an FPGA is a reconfigurable LSI that contains millions of programmable logic gates [4].

II. Implementation of succinct data structures

A. Succinct data structures

Succinct data structures are introduced in [1], [5] for static trees and graphs. 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 sequence 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 sequence 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 (or queries) need not be supported efficiently, so that the processing time of

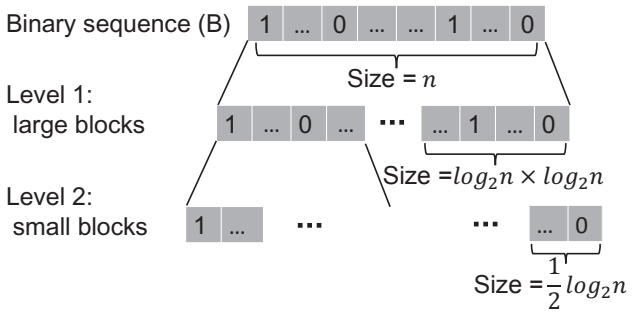


Fig. 1. Implementation of succinct data structures using a two level block structure.

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)" [5], "balanced parentheses" [6], "depth first unary degree sequence (DFUDS)" [7], 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*.

B. Implementations methods

In the initial work of succinct data structures [5], a method to compute the *rank* in a constant time is proposed. As shown in Fig.1, for a given binary sequence $B[0, n-1]$ of size n, a two-level block structure is built. 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 $2log_2 n$ number of 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* queries in a constant time. Please refer [5] for more details. Since we use many arrays and tables, this method needs multiple (although a constant number of) memory reads to compute the *rank*. In hardware, we can use a popcount (population count) [8] module to count the number of symbols in a constant time. Therefore, the pre-calculated rank tables are not required.

Another popular method is proposed in [9] and often called "RRR" encoding. This method is originally introduced for a bit vector. Similar to [5], a two level block structure is used. The small blocks are classified in to different classes according to the number of ones (or zeros) in their bit vector. For each class, a separate table of cumulative *ranks* 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 the pointers in small blocks. Although these methods are initially proposed for bit vectors, they are later extended to be used with multi-bit symbols.

The problems of these methods are the large storage size and the memory access bottleneck. As shown in the above implementation methods, although the storage size is linearly related to the number of elements in the sequence, it is always larger than the sequence size. As a result, the implementation of an application is restricted by the storage capacity. Moreover, the processing time is often decided by the memory access bandwidth and not by the computation overhead as shown in many applications such as genome sequence alignment [10]. One promising way of reducing the storage capacity and increasing the memory access speed is to compress the data. Since the data size is small, both the storage capacity and the access time is reduced. However, this is not an effective option 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.

III. Hardware oriented succinct data structure

A. Related works

In our earlier works, we used word-pair encoding [2] to compress the data. Note that, although this method is very similar to the byte-pair encoding (BPE) [11], a pair can have any number of bits unlike 8 bits in BPE. Fig.2 shows the word-pair encoding. The data sequence is shown in Fig.2(a). In this sequence, 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. In this example, we assume that we can use up to 16 symbols, so that each symbol is represented by 4 bits. We replace the word-pairs "c,o" and "a,n" by new words "X" and "Y" respectively. The 112-bit sequence is compressed to 84 bits, where the compressed sequence contains 21 symbols. Note that, the space (indicted by "␣") is also considered as a symbol.

This method is not efficient in hardware. In hardware, we have to store each block in the main memory (usually in

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 sequence. 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 sequence after word-pair encoding. Total size = 22 symbols × 4 bits = 88 bits

Fig. 2. Searching the quarry Q in text X

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	.	␣						

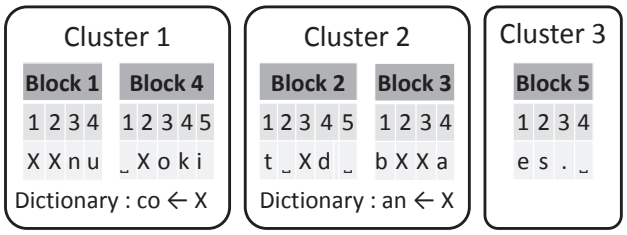
(a) Data sequence divided in to blocks.

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	.	

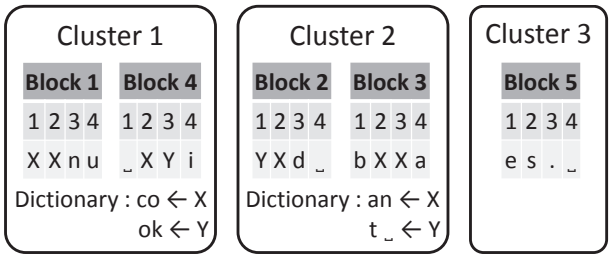
Dictionary : co ← X, an ← Y

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

Fig. 3. Block based compression. Different blocks have different compression ratio. Hardware is designed for the worst compression ratio.



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



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

Fig. 4. Clustering of blocks. Different blocks have a similar compression ratio.

a DRAM). To access a particular data, we have to know the location or the memory address of the stored data. As shown in Fig.2, different parts of the sequence are compressed differently. That is, some parts are heavily compressed and while the other parts are not compressed at all. Therefore, we cannot find a direct relationship between the position of the sequence data and the memory address of its corresponding compressed data. In order to find the required data, either we have to decompress the whole compressed sequence, or we have to maintain a separate index information that gives the corresponding positions the symbols in the original and the compressed sequences. Neither of these methods are effective.

To solve this problem, we have considered a block-based compression in [3]. Fig.3 gives an example of this method. As shown in Fig.3(a), the data sequence is divided in to multiple blocks of the same size. Then, each block is compressed using word-pair encoding as shown in Fig.3(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. However, 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. As shown in Fig.3, some blocks are easy to compress while the others are difficult. As a result, the compressed sequence contains different sized blocks. 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 such as the block sizes, memory address of each block, etc to access the correct block. Therefore, we have to 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 Fig.2.

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.

B. Proposed compressed data structure

The proposed succinct data structure is also based on the two-level block structure. The first level is similar to [5] where the results of the *rank* operation of the first entry of each large block is stored in a separate array. The compression is applied to the blocks in the second level. We perform k-means clustering [12] to assign blocks to clusters. According to the experimental results, we found that around 100 clusters are sufficient. Therefore, k is set to 100. The similarity of the symbols are considered as the criteria for the clustering. The similarity is roughly represented by the number of the same symbols, or the number of the same symbol pairs. Since the blocks in the same cluster are similar, such blocks can be compressed easily. The less compressible blocks are usually very different from each other. (Note that, if those are alike, those would have been easily compressible in the first place). 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.

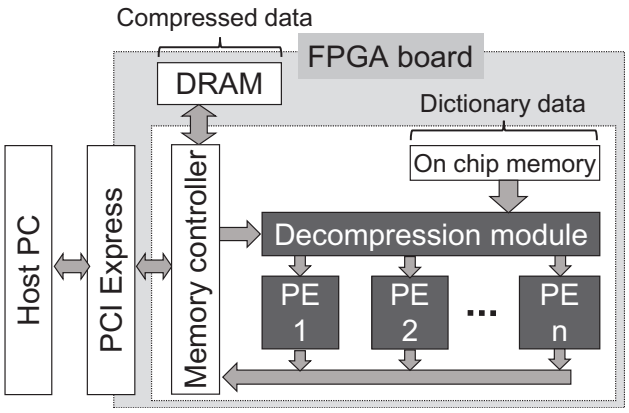


Fig. 5. FPGA based system architecture

Fig.4 shows an example of the proposed succinct data structure. We divide the data sequence in to blocks of 6 symbols similar to the example in Fig.3. As shown in Fig.4(a), 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: 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.

Step 2: 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.

C. Implementation on hardware

The overall hardware architecture based on FPGA is shown in Fig.5. In the hardware implementation, we consider that the compressed sequence 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 send to the processing elements for parallel processing. The DRAM is accessed through a memory controller.

Fig.6 shows the format of the code word of the compressed data. It is composed with the cluster number and 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 sequence are of the same size. However, they can represent one or more symbols in the original sequence. Fig.7 shows the architecture of the decompression module. In the decompressing process, we separate all symbols in the compressed data. For each symbol, we refer the dictionary and

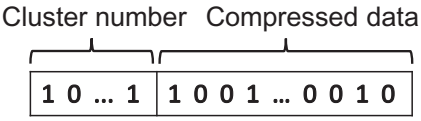


Fig. 6. Compressed data format

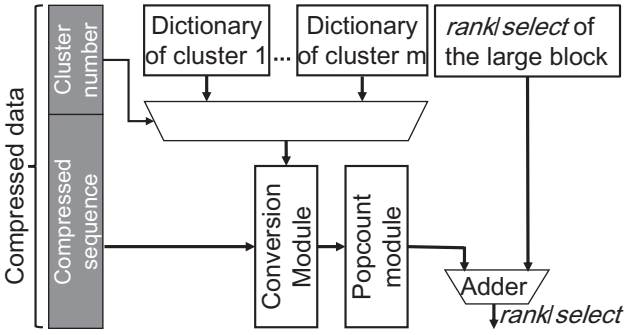


Fig. 7. Architecture of the decompression module.

convert them to the original symbol. The correct dictionary is selected according to the cluster number. The decompressing of each symbol 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. The *rank* or *select* operations are computed by adding the popcount value to the *rank* of the large blocks.

IV. Evaluation

We use the text file "Bible.txt" [13] as the example to evaluate the proposed data structure. We consider a text search application using succinct data structures [3]. BW transform [14] is used for the text search so that the succinct data structure is applied for the transformed text. Note that, some parts of the BW transformed text contain repeated symbols while some other parts have different symbols. Therefore, it is very hard to achieve a uniform compression in every parts of the text so that its efficient implementation on hardware is difficult to achieve.

For the hardware, we used the DE5 board [15] that contains "Altera 5SGXEA7N2F45C2 FPGA" and two 2GB DDR3-SDRAMs. The memory controller has a word width of 512 bits. The k-means clustering is done using Matlab R2014b, and it takes around 10 minutes to complete. The text file contains 8 bit symbols. Since word-pair encoding requires new symbols, we allow up to 10 bits per a symbol. Therefore the compressed text contains symbols of 10 bits large.

Table I shows the comparison against the method in [3]. The compression ratio of the proposed method is 37.5% which is 3 times larger than that of [3]. The compression ratio C_r is calculated by Eq.(1), where the original block size and the compressed block size are given by B_s and C_s respectively. The compression based on [3] is restricted to the worst case. Since it is difficult to achieve a uniform compression in every parts of the BW transformed text, the compression ratio in [3] is low. However, in the proposed method, we improve the

TABLE I
COMPARISON WITH [3]

Method	Data sequence			Compressed sequence			Compression ratio
	Symbols per a block	Bits per a symbol	Block size B_s (bits)	Symbols per a block	Bits per a symbol	Block size C_s (bits)	C_r (%)
Work in [3]	100	8	800	71	10	710	11.25
Proposed method	100	8	800	50	10	500	37.50

TABLE II
NUMBER OF CLUSTERS VS. COMPRESSION RATIO.

Compressed block size (bits)	Number of clusters	Compression ratio C_r (%)	Dictionary size (kB)
540	10	32.50	4.6
530	30	33.75	8.2
510	80	36.25	14.4
500	100	37.50	20.1
500	130	37.50	21.4
500	500	37.50	53.4
500	1000	37.50	74.8

worst case so that the it is close to the average case. As a result, the compression ratio is large.

Since the word size of the memory controller is 512 bits, a one memory read on DE5 board provides 512 bits. We store the compressed 500 bits in one memory word. The rest of the 12 bits are used to store the cluster number. As a result, we require only one memory read to retrieve the data of 100 symbols. If we did not compress the data, two memory reads are required to access the original 800 bits long block. Therefore, in the proposed method, the memory accesses are reduced by 50%. In other words, the memory access speed increased by two times.

$$C_r = \frac{B_s - C_s}{B_s} \times 100\% \quad (1)$$

Table II shows the compression ratio against the number of clusters. The initial text sequence has 100 symbols large blocks so that the block size is 800 bits. According to the results, initially the compression ratio increases with the number of clusters. After using 100 clusters, the compression ratio remains the same. However, the dictionary size is increased with the number of clusters. On the other hand, the dictionary size is very small compared to the size of the "Bible.txt", which is 4480kB large. Moreover, the FPGA we used contain a large on-chip memory of over 5MB, so that we can easily store the dictionary data. As a result, the dictionary size is not a problem in the proposed data structure.

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 sequence 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. According to the evaluation using FPGA, we shows that the data can be compressed by 37.5% while allowing fast data access with small decompression overhead.

ACKNOWLEDGMENT

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

REFERENCES

[1] G. Jacobson, "Succinct static data structures," 1989.
[2] H. M. 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 *IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pp. 639–642, Nov 2014.
[3] 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 *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pp. 354–359, Jul 2015.
[4] 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 *Procedings of the Custom Integrated Circuits Conference*, pp. 233–235, May 1986.
[5] G. Jacobson, "Space-efficient static trees and graphs," in *30th Annual Symposium on Foundations of Computer Science*, Nov 1989.
[6] J. I. Munro and V. Raman, "Succinct representation of balanced parentheses, static trees and planar graphs," in *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pp. 118–126, 1997.
[7] 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.
[8] H. S. Warren, *Hacker's Delight (2^{nd} edition) - Chapter 5*.
[9] R. Raman, V. Raman, and S. S. Rao, "Succinct indexable dictionaries with applications to encoding k-ary trees and multisets," in *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 233–242, 2002.
[10] H. M. Waidyasooriya and M. Hariyama, "Hardware-acceleration of short-read alignment based on the burrows-wheeler transform," *IEEE Transactions on Parallel and Distributed Systems*, 2015.
[11] P. Gage, "A new algorithm for data compression," *C/C++ Users Journal*, vol. 12, no. 2, pp. 23–28, 1994.
[12] J. Macqueen, "Some methods for classification and analysis of multivariate observations," in *5-th Berkeley Symposium on Mathematical Statistics and Probability*, pp. 281–297, 1967.
[13] "The king james version of the bible." http://www.gutenberg.org/ebooks/10.
[14] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," tech. rep., 1994.
[15] "Altera development and education boards." https://www.altera.com/support/training/university/boards.html#de5.