An FPGA Architecture to Accelerate the Burrows Wheeler Transform by Using a Linear Sorter

Juan Andrés Pérez-Celis, José Martínez-Carranza, Alicia Morales-Reyes, Claudia Feregrino-Uribe and René Cumplido

***Abstract*—Abstract—The Burrows-Wheeler Transform (BWT) has been used in several applications demanding high volume of data and real-time capabilities. Current research focuses on reducing the time to compute the BWT by software or hardware means. This paper presents a novel FPGA architecture based on a Linear Sorter (LS) to efficiently calculate the BWT. The architecture is composed of a control logic and several identical Comparison Units (CU), in order to provide scaling flexibility. The proposed hardware implementation sorts the string as it is fed, substitutes only the required characters and efficiently computes the BWT without knowing the Longest Common Prefix (LCP). The architecture is implemented in an FPGA showing a significant reduction in the cycles required to compute the BWT. The results show that the cycles involved to calculate the BWT are reduced for any given string in comparison to those reported in previous works where the LCP is not known in advance.**

***Index Terms*—Burrows-Wheeler Transform, String Sort, Linear Sorter, FPGA.**

I. INTRODUCTION

THE Burrows-Wheeler Transform (BWT) [1], [2], [3] has been widely used in real time and high volume data, demanding applications such as string matching in genome sequences [4], [11], shape matching in computer vision [5], compression and other applications [6]. Due to the nature of these applications it is necessary to have hardware efficient architectures capable of computing the BWT in a fast manner.

Hardware implementations of the BWT are required to perform several operations. The BWT calculation not only implies several lexicographic sorts at initial and semi-final stages of its computation, but it also requires to perform data substitutions when necessary. The BWT is fully obtained when no more substitutions are required. The output is a set of identification numbers corresponding to the suffix of the BWT.

A straightforward implementation of the BWT requires a large amount of resources, specially of memory. Figure 1 shows an example of a direct implementation. To compute the BWT, a rotation matrix is built from the input string. After this, the matrix's rows are lexicographically sorted until the BWT is placed in the last column along with an identifier (j) to locate the input string.

However, the BWT can be computed without the rotation matrix. The transform is computed by assigning an index to each character in the string and then sorting these. Afterwards, those sets of repeated characters are identified and marked

J. A. Pérez-Celis, J. Martínez-Carranza, A. Morales-Reyes, C. Feregrino-Uribe and R. Cumplido are with the Computer Science Department, Instituto Nacional de Astrofísica, Óptica y Electrónica,Tonantzintla, Puebla 72840. e-mail: (andres.perez.celis@ieee.org).

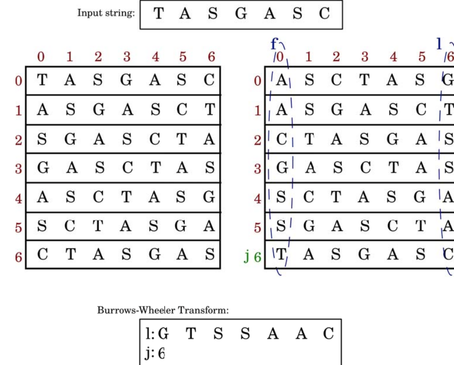

Fig. 1. Example of the BWT on a string.

for a further substitution with their suffix. The substitution continues until there are no sets of repeated characters. Finally, indexes are read and the BWT is computed by obtaining the prefix. The prefix can be obtained by applying the *mod* operation of each index minus one with the string size as shown by Eq. 1:

$$i_p = (i - 1) \ mod \ n; \quad (1)$$

where i_p is the prefix of index i and n is the string length. More details and an example are found in Section III.

Inspired by the procedure described above, in this paper, we present a novel FPGA architecture to carry out the BWT. Our approach deals with the fact that substitutions have to be performed several times when computing the BWT. However, not all characters of a string should be substituted. Identifying and replacing only those required characters decreases the number of clock cycles required to compute the BWT. Moreover, the proposed architecture is capable of sorting characters whilst these are being fed, thus decreasing the total number of clock cycles required to compute the BWT.

In order to describe our approach, this paper is organized as follows: Section II discusses previous hardware implementations. Section III describes the proposed approach to compute the BWT with a Linear Sorter (LS). Section IV describes in detail the proposed architecture design. Section V presents an analysis of the steps required to compute the BWT and a comparative analysis to previous proposed hardware implementations. Section VI shows the experimental setup and discusses the results obtained. Finally, conclusions are presented in Section VII.

978-1-5090-3682-0/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPSW.2016.174

IEEE computer society

II. RELATED WORK

Hardware implementations to compute the BWT without the rotation matrix have been presented in several works [9], [8], [10]. The main difference among these works relies on the approach taken to sort the strings as this is one of the most time consuming tasks of the BWT.

In [9], the authors based their architecture on the Weavesorter algorithm to sort the string. The operation begins by right shifting a string's value and comparing it at every step. The sorted string is obtained by left shifting the stored data. However, in order to complete the BWT, equal elements should be substituted via software operations. After this the whole string should be sorted again.

Another approach was taken in [8], where the authors use a parallel sorting strategy. Strings can be sorted in $n/2$ steps, which is faster than the previous approach. However, the sorted string should be output completely and fed again with substituted data. Afterwards, the parallel sorting architecture will only sort data that has changed. This work is faster than applying the Weavesorter algorithm [9].

Finally, another architecture presented in [10] takes advantage of a FIFO network to sort the data. To the best of our knowledge, this approach takes the minimum number steps to perform the BWT. However, it makes a significant assumption: the length of the Longest Common Prefix (LCP) should be known in advance. The architecture should be synthesized based on the length of the LCP. This assumption introduces an advantage, when compared to other implementations, as only one iteration is needed to perform the BWT, hence a block of size equal to the LCP is stored for all characters in the string, which provides all necessary data for the sorting. A main drawback in this architecture is the requirement to know LCP in advance, which also leads to its inability to work in those cases where the LCP may be variable.

Motivated by the above, our proposed architecture is based on a Linear Sorter (LS). The LS for BWT (LSBWT) is capable of computing the BWT without knowing the LCP beforehand. Moreover, the elements are sorted as they are fed to the architecture due to the LS characteristics. During the substitution phase in other implementations, the whole string was output and fed again. In contrast, the architecture proposed in this work addresses the challenging task of having to substitute only the required elements. Thus, not all string elements should be fed again for extra iterations, because only substituted elements are changed. Likewise, only the substituted elements should be sorted.

III. LINEAR SORTER FOR BWT

Our work is based on the suffix sorting approach [8], where the rotation matrix is not needed. For the sake of clarity, consider the same input string as in the previous example ("TASGASC"). Figure 2 illustrates the computation of the BWT. The steps involved are:

1) Assign an index to each character in the string.
2) Sort the characters lexicographically without modifying the index assigned to each character.
3) Set a sort counter to 1.
4) Identify groups of equal characters.
5) Substitute each character for the one positioned x places after it; where x is the value of the sort counter. If x is bigger than the remaining lenght of the string, the count of the remamining x places continues from the beginning of the string. Do not modify the assigned index.
6) Sort the substituted characters within each group.
7) Check for repeated characters in each group. If yes, increase the sort counter and repeat from step 5. If not, go to the next step.
8) Read the assigned indexes of all the characters in the sorted string.
9) Compute the prefixes of the assigned indexes read.
10) The result is the string comprised by the characters pointed by the prefixes.

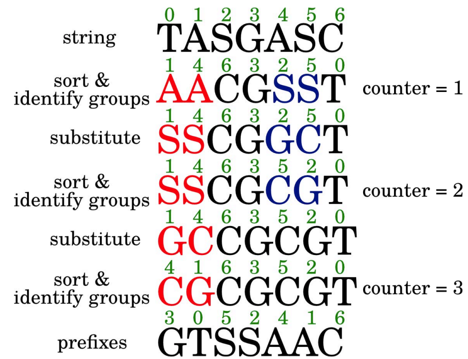

Fig. 2. Example of the BWT by the prefix method.

The LSBWT is based on an LS, which has the advantage of sorting data on the fly. Architectures inspired on LS have been used previously [7], but not for implementing the BWT. An example of an LS is depicted in Fig. 3. Each node is interconnected with its immediate neighbors. All nodes are connected to the input. The data input compares with all nodes. By knowing left, and in-node values, nodes can decide whether to shift right, hold its current value or store the incoming data.

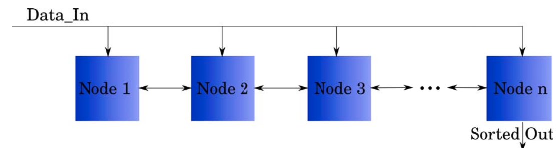

Fig. 3. Example of a Linear Sorter.

The LSBWT architecture when implemented on a FPGA presents high flexibility to scale the design, as it replicates the Comparison Unit (CU); an advantage of sorting while data is fed; and it does not need to know the Longest Common Prefix (LCP) beforehand. It is worth pointing out that LSBWT presents a key innovation: substituting only required data instead of the whole string.

IV. Proposed Architecture

The proposed architecture is based on a LS. In LSBWT nodes are referred as Comparison Units (CU). Each CU is capable of storing data and id and comparing to left and right incoming data. These capabilities are implicit to an LS. However, CUs are also capable of computing several control signals required to identify sets of equal characters that should be substituted.

Figure 4 shows the internal logic of a CU. Behavior of the CUs can be described in the next processes:

- Comparing data.
  - *If data at the left CU ($DATA_LEFT$) is equal to data in the actual CU ($DATA_NOW$).* The process sets EQL flag when true.
  - *If data at the right CU ($DATA_RIGHT$) is equal to $DATA_NOW$.* The process sets EQR flag when true.
  - *If input data is less than $DATA_NOW$.* The process sets $LESS_OUT$ when true.
- Operating data.
  - *Checking if a substitution should be performed.* When a substitution is required data stored is set to the maximum value (MAX_VALUE) by performing a parallel load in the $DATA_NOW$ register.
  - *Storing input data or left data and shifting actual data.* The CU stores data when the data fed ($DATA_INPUT$) is less than $DATA_NOW$. When false, $DATA_NOW$ remains unchanged. To decide which data is stored between $DATA_LEFT$ and $DATA_INPUT$; CU checks if input $LESS_IN$ (which is CU's output $LESS_OUT$ to the left) equals 1. This means that CU to the left stores $DATA_INPUT$ and current CU stores $DATA_LEFT$. $LESS_IN$ equal to zero means that current CU stores $DATA_INPUT$. During this process, the behavior is similar to a shift register — before storing data the CUs shift their data to the right.

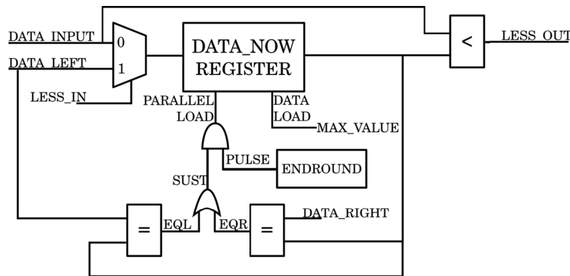

Fig. 4. Logic diagram of a CU.

The LSBWT comprises several CUs, the number of CUs needed is equal to the size of the string to be transformed. The proposed architecture is capable of only substituting elements that are equal to one of its immediate neighbors. In order to achieve this, additional resources are needed. The most important components are:

- Priority encoder, outputs the location of the first '1' read from left to right of the SUST flags.
- One-hot decoder, has the output of the priority encoder as input.
- Enable multiplexer, enables are selected depending on several conditions.
  1) For the first sorting iteration all enables are set.
  2) For ongoing iterations if EQL flag is 0 enables are one-hot decoder's output.
  3) For ongoing iterations if EQL flag is 1, the enables are the sum of all previous outputs of the one-hot decoder.
- Round end: Set one of its outputs when the first iteration is completed. The other output generates a pulse each time another iteration is completed. The pulse remains set when the BWT is completed.

The architecture diagram, in Fig. 5, draws four CUs to demonstrate its functionality. The BWT is computed as follows:

1) A reset signal writes the maximum number to the data value of CUs and sets enables.
2) Data and indexes are fed to the CUs.
3) Data is sorted while being input.
4) The first iteration ends when the last data arrives and a pulse is generated.
5) With this pulse EQL and SUST values are stored in a register.
6) CUs' data value is set to maximum where a substitution takes place.
7) Substitution starts.
8) Data is substituted when required by using the priority decoder and the one-hot decoder.
9) Another pulse is generated when SUST register is empty.
10) With this pulse EQL and SUST values are stored in a register.
11) If all SUST values equal zero, the BWT is completed. If not, the architecture iterates from step 5.

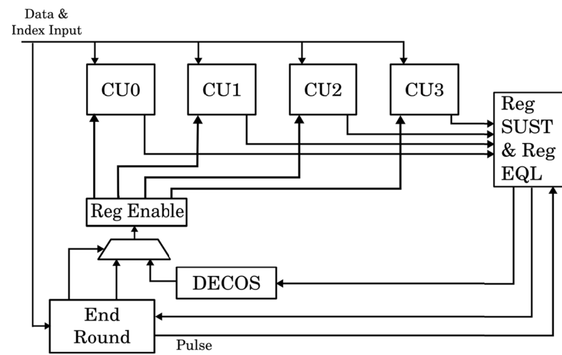

Fig. 5. Reduced block diagram with 4 CUs.

During the substitution phase identifying the groups of characters to be substituted was a challenging task. The task was accomplished by considering that:

- Two groups of different characters can be together (e.g. "AAASSS"). The groups should be divided.

- After a substitution and a sort iteration of a group of characters, characters of the sorted group may be equal to adjacent characters. The architecture should not compare the characters within the group with adjacent characters. The comparisons are only within each group.

For the first case the signals EQL and SUST are used. The signal SUST will inform when a character is equal to one adjacent. The additional information provided by EQL –character equal on the left– allows to know the beginning of a group of characters. If SUST = '1' and EQL = '0', the character is the first in a group.

For the second case, after identifying repeated characters, the CUs with the remaining characters on the string are disabled.

The next section compares the proposed hardware architecture to previous developed approaches.

V. Steps Analysis and Comparison

The LSBWT is capable of sorting a string of a length equal to the number of CUs in the architecture. The steps needed are the following:

- The LS sorts the string as it is fed, taking n cycles in this process; where n is the string size.
- The architecture takes n cycles to output the string when finished.
- The next cycle after the first sort is used to store the required registers.
- In the next cycle, the architecture indicates the BWT is completed or resets the registers if a substitution is needed.
- If another iteration is needed the architecture takes m_i cycles to sort the string; where m is the number of characters that should be substituted for the ith iteration. Additionally, it takes 2 cycles to generate the final pulse of the iteration and resets the CUs if needed.
- The previous point may be performed k times; where k is the number of iterations.

For the worst case, m_i is equal to n for all i iterations. The expression for the number of steps needed in the worst case scenario is $2n$ for the string's input and output; plus 2 for storing in registers and reseting or identifying the end; plus $(n + 2)(k - 1)$ which represents the steps for each iteration additional to the first one. The expression is given by Eq. 2.

$$steps = n(k + 1) + 2k; \quad (2)$$

Table I compares the number of steps taken by the architectures proposed in [8] and [9]. The work regarding FIFO networks was not considered in this comparison as there is not a reported expression for the steps needed to compute the BWT [10]. The steps comparison is for the worst case scenario. For the Parallel Sorter, the worst case is when it takes $n/2$ to sort a string. For our approach, the worst case is when the whole string should be substituted during every iteration. The number of steps for the Weavesorter does not depend on the content of the string, only on its size and necessary iterations. Table I also shows the increase in the number of steps in relation to the proposed approach.

TABLE I
Comparison of the number of steps to compute the BWT in hardware.

	Expression of steps	Steps for n=128, k=8	Step count (normalized)
Weavesorter [9]	$2n(k + 1)$	2304	1.97
Parallel Sorter [8]	$(\frac{3}{2}k + 1)n$	1664	1.42
Linear Sorter Proposed approach	$n(k + 1) + 2k$	1168	Base line

TABLE II
Resource summary.

Selected Device: XC6VLX75T

	Used	Available	Percentage
Slice Registers	1419	93120	1%
Slice LUTs	5434	46560	11%
Fully used LUT-FF pairs	1419	5434	26%
Bonded IOBs	22	240	9%
BUFG/ BUFGCTRLs	1	32	3%

From the results shown in Table I it can be seen that the Weavesorter implementation takes almost two times the number of steps to compute the BWT when compared to our approach. LSBWT speed up may increase when the complete string is not substituted in every iteration. The Parallel Sorter (PS) is sensitive to the string content; namely, the characters order. The LSBWT is sensitive to the string's content too; namely, the number of repeated characters. However, the proposed approach outperforms the PS. The best case scenario for the PS is when the input string is already sorted. The expression of the steps in PS is $(n + 1)(k + 1) + k$ which is 1 step slower than LSBWT worst case scenario.

VI. Experimental Results

LSBWT design was synthesized in a Kintex 7 XC7K70T using ISE 14.7. The architecture reaches a frequency of $152.022MHz$ (Clock period: $6.578ns$) for 128 CUs. The resource utilization is presented in Table II.

The general expression for the steps required to compute the BWT is given by eq. 3

$$steps = 2n + 2 + \sum_{i=1}^{k-1}(2 + m_i); \quad (3)$$

where n represents the length of the string, k the iterations needed and m_i the number of characters that should be substituted at iteration i.

To verify eq. 3, experiments were performed with 10 random strings taken from this text. Table III shows the number of steps required to sort each string. Likewise, the number of iterations and substitutions at each iteration are presented. Results show that the number of steps counted are the same as the number of steps computed with eq. 3.

TABLE III
STEPS FOR 10 RANDOM STRINGS OF LENGTH 16.

Iterations (k)	Substitutions per iteration	Total steps
2	7	43
2	8	44
2	11	47
3	7,2	47
3	10,2	50
3	11,4	53
3	13,2	53
3	10,6	54
4	12,5,2	59
6	13,8,6,4,2	77

The number of steps to compute the BWT given a certain string for the PS and the proposed approach depends on different characteristics of the string. String details used in [8] are not enough to make a fair comparison. However, the experiment for the worst case scenario, where 127 out of 128 characters are the same, is a good test to assess such worst case in our approach. Experimental results are shown in Table IV. These results suggest that LSBWT reduces 47% the number of steps required by the PS architecture and 73% the steps taken by the Weavesorter. The approach presented in [10] for the worst case yields an architecture that should store the complete rotation matrix, which translates in a high demand of memory.

TABLE IV
STEPS TAKEN FOR A STRING WITH 127 EQUAL CHARACTERS OUT OF 128.

	Steps for worst case	Percentage increase
Weavesorter [9]	32768	73%
Parallel Sorter [8]	16639	47%
Linear Sorter Proposed approach	8768	Base line

The last experiment consist of comparing the three reported architectures and the proposed architecture by performing the BWT for two strings of size 128 with LCP equal to 8. The characteristics of the string were chosen to match the reported cases. The comparison against the Weavesorter and the FIFO network architecture was chosen due to the fact that both architectures are not sensitive to the string's content. However, the comparison against the PS architecture serves only to show the proposed approach performance. Note that even when both strings are taken from English text, have the same size and LCP, the content of the string affects the results and it was not possible to get the same strings. Moreover, is worthy to point out that to make the comparison as fair as possible, the comparison was made based on the steps taken by each architecture. The corresponding results are presented in Table V.

Results in Table V show that our approach outperforms the Parallel Sorter and the Weavesorter. The number of steps are reduced by 76% for the Weavesorter approach and by 57% for the parallel sorter approach. However, the FIFO network computes the BWT in the least number of steps. The main reason is that the approach taken in [10] is based on knowing the LCP beforehand and synthesizing a custom architecture for the LCP. The advantage of knowing the LCP beforehand translates in an architecture which only sorts once to complete the BWT. The cost is a higher need of memory resources as the LCP grows [10]. Moreover, the architecture based on a FIFO network will not compute the BWT if the string has a LCP greater than the one the architecture was synthesized for. The proposed architectural design computed the BWT for a given string with no LCP constraints.

TABLE V
BWT PERFORMANCE COMPARISON FOR 128 STRING LENGTH WITH 8 LCP.

	Steps for First String	Steps for Second String	Max. Frequency	Device
Weavesorter [9]	2304	2304	45 MHz	Virtex xcv300
Parallel Sorter [8]	1234	1229	51.6 MHz	Virtex2 xv2v2000
FIFO Network [10]	285	285	51.4 MHz	Stratix EP1S10B672C6
Linear Sorter Proposed approach	525	543	152 MHz	Kintex 7 XC7K70T

VII. CONCLUSIONS

We have presented an FPGA architecture to carry out the BWT. Our approach is based on the use of a Linear Sorter, which greatly helps to accelerate the processing time invested in performing the transform. Our proposed architecture presents several remarkable advantages as it does not need to substitute the whole string for additional iterations. In addition, the LSBWT can automatically stops when the BWT is done, thus there is no need to know the LCP in advance. Another advantage is that the architecture is based on CUs, these units are all the same, which simplifies the implementation. Few changes are needed in order to expand the architecture. Moreover, along with the flexibility provided by FPGAs, the architecture can be scaled based on the application. The latter is related to fact that LSBWT can scale until no resources are left in the FPGA with minimum changes in the VHDL file.

The LSBWT architecture outperforms the PS and Weavesorter approaches for any given string. The PS may take 42% more cycles to compute the BWT than our approach and in the worst case it takes 89% more. The Weavesorter may take 96% more cycles to compute the BWT and in the worst case scenario it takes 273% more cycles. The number of clock cycles taken by our approach is less than those reported in previous works. It is true that the work in [10] implies a reduced number of clock cycles, but its functionality is based on knowing the LCP beforehand, which may not be known or may even be variable in some applications.

As a future work, expanding the CUs depth in order to store more than one character will be explored. The idea is to reduce the number of iterations needed to compute the BWT while expanding the memory use. A trade-off analysis between the performance and the resource utilization will also be carried out.

VIII. ACKNOWLEDGEMENTS

This work was partially supported by MSc scholarship No 392068 by CONACyT and under CONACyT grant

PDCPN2013-01-216689.

References

[1] Burrows, M., & Wheeler, D. J,. “A block-sorting lossless data compression algorithm,” SRC Research Report 124, Digital Systems Research Center, Palo Alto, Calif., 1994.

[2] Chen, N.C., Chiu, T.Y., Li, Y.C., Chien, Y.C. and Lu, Y.C., 2015, October. “Power efficient special processor design for burrows-wheeler-transform-based short read sequence alignment,” IEEE Biomedical Circuits and Systems Conference (BioCAS), 2015.

[3] Kimura K, Koike A. “Ultrafast SNP analysis using the Burrows–Wheeler transform of short-read data,” Bioinformatics, 2015.

[4] Adjeroh, D., Zhang, Y., Mukherjee, A., Powell, M., and Bell, T. “DNA sequence compression using the Burrows-Wheeler transform,” IEEE Computer Society Bioinformatics Conference, pp. 303–313. 2002.

[5] Donald Adjeroh, U. Kandaswamy, N. Zhang, A. Mukherjee, M. T. Brown, and Tim Bell. “BWT-based efficient shape matching,” Proceedings of the 2007 ACM Symposium on Applied Computing (SAC ’07). ACM, New York, NY, USA, 1079-1085, 2007.

[6] Marcelino, R.; Neto, H.C.; Cardoso, J.M.P., “Unbalanced FIFO sorting for FPGA-based systems,” Electronics, Circuits, and Systems. ICECS 2009. 16th IEEE International Conference, pp.431,434, 13-16 Dec. 2009.

[7] Perez-Andrade, Roberto, Rene Cumplido, Claudia Feregrino-Uribe, and Fernando Martin Del Campo. “A versatile linear insertion sorter based on an FIFO scheme,” Microelectronics Journal 40, no. 12: pp.1705-1713. 2009.

[8] Martinez, J.; Cumplido, R.; Feregrino, C., “An FPGA-based parallel sorting architecture for the Burrows Wheeler transform,” Reconfigurable Computing and FPGAs. ReConFig 2005. International Conference, pp. 28-30. 2005.

[9] Mukherjee, A.; Motgi, N.; Becker, J.; Friebe, A.; Habermann, C.; Glesner, M., “Prototyping of efficient hardware algorithms for data compression in future communication systems,” Rapid System Prototyping, 12th International Workshop, pp.58-63, 2001.

[10] Cheema, Umer I., and Ashfaq A. Khokhar. “A high performance architecture for computing burrows-wheeler transform on FPGAs,” ReConFig, pp. 1-6. 2013.

[11] Waidyasooriya, H., and Hariyama, M. “Hardware-acceleration of short-read alignment based on the Burrows-Wheeler transform,” IEEE Transactions on Parallel and Distributed Systems. 2015.