A Dynamically Reconfigurable ECC Decoder Architecture

Awais Sani, Philippe Coussy, Cyrille Chavet
Lab-STICC / Université de Bretagne Sud
Lorient, France

***Abstract*— Due to their impressive error correction performances, Error Correcting Codes (ECC) are now widely used in communication systems. In order to achieve high throughput requirements ECC decoders are based on parallel architectures, which results in a major issue: memory access conflicts. In this paper, we introduce a new class of ECC decoder architectures that dynamically reconfigures by executing on-chip a memory mapping approach. For that purpose, a dedicated algorithm taking into account network constraint is presented. A smart architecture based on a butterfly network and a reconfiguration unit is also proposed. Experimental results show that real-time reconfiguration at reasonable hardware cost is possible.**

I. INTRODUCTION

ECC like turbo-codes and LDPC [1][2] have been adopted by many communication standards to exploit their excellent error-correction capability. However, executing these iterative algorithms sequentially leads to poor performances: parallelization of decoder architectures to achieve very high throughput at low power budget is thus required. Unfortunately, parallel architectures lead to conflicting memory accesses which can dramatically decrease the system's performances. Apart from defining conflict-free interleaving rules, existing works can be divided into two families to reduce or avoid collision problems: (1) *runtime approaches* in which extra memory elements and control logic are used in order to serialize conflicting accesses (e-g [4]-[7]) and (2) *design time approaches* that find a conflict-free memory mapping (e.g. [8]-[10],[12],[17]). In the first family, [4][5] propose to design a dedicated interconnection network called LLR distributor to tackle conflict problem while Butterfly and Benes networks are investigated in [6] to meet higher throughputs. Binary de Bruijn network is used in [7] to provide more scalability and allow any permutation to be routed efficiently. Conflicts are managed by deflecting the conflicting packets appropriately until they reach their targets. Unfortunately, runtime approaches suffer from large area due to the additional buffers which serialize conflicting accesses but also introduce a delay and degrade throughput.

The second family of approaches consists of mapping algorithms proposed to provide concurrent accesses to all processing elements without any conflict. For this purpose, pre-processing is realized off-line on a computer to determine the memory locations for each data element. In order to find conflict free memory mapping in turbo decoder, works in [8] [9] present meta-heuristic-based methods. The proposed approach is based on a simulated-annealing algorithm. In [10], a heuristic is proposed which finds conflict free memory mapping for a given interconnection network. The approach given in [17] also finds conflict free memory mapping by adding extra elements to the architecture in order to support a target interconnection network. However, all these approaches fail to remove the computational complexity of the problem and require off-line preprocessing to map data (see [15]). In [12] the authors introduced a polynomial time approach which is based on Euler partitioning and allows decreasing the computational complexity.

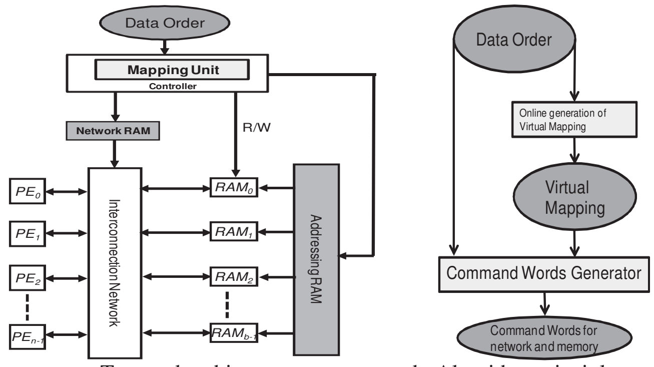
Design time approaches require their resulting memory mapping to be implemented by storing in a dedicated ROM a set of command words driving the architecture. Unfortunately, to support several block lengths and standards, multiple ROMs are required which results in important hardware cost. An approach has been introduced in [15] to combine design and runtime approaches by embedding methods from [8] and [10] on-chip to propose flexible decoders. Target architecture includes several processing elements and memory banks interconnected through a crossbar network. Decoder architecture executes mapping algorithm online whenever new block length needs to be decoded. Generated command words are stored in two RAMs that replace the set of classically used ROMs to provide conflict free access to the memory for the current block length. However, the computational complexity of the mapping approaches available at that time make the solution infeasible for the current telecommunication standards. In [18], the authors improved the performances by embedding the polynomial time memory mapping algorithm proposed in [12] on-chip by running it on soft-core processors. Benes network was considered and a simplified routing algorithm that is executed on-chip along with the mapping algorithm has been presented.

In this article, we propose to adapt the algorithm from [12] to target a butterfly network. We also propose an original dedicated HW architecture to implement efficiently the method on-chip. Results based on an HSPA interleaver used in 3GPP-WCDMA standard show that real-time reconfiguration is possible at a reasonable HW cost.

The rest of the paper is organized as follows. Section 2 describes the proposed approach in which we define the mapping algorithm and the target architecture. Section 4 presents experimental results before section 5 concludes.

II. PROPOSED APPROACH

Our architecture consists in a mapping unit and associated memories used to execute the mapping algorithm and to store mapping information (see Fig. 1.a). It also includes a set of Processing Elements PEs connected to a set of memory elements RAMs through an interconnection butterfly network. Data are exchanged by the PEs according to the interleaving law. Our architecture first generates a *Virtual Mapping (VM)* for a given block length and interleaving rule. Whenever a new block length (or standard) needs to be decoded, the mapping algorithm is executed. It generates a new VM that is stored into a dedicated memory to provide conflict free parallel accesses to the RAMs. Then this VM is translated and completed on-line at runtime to generate the command words that drive the interconnection network and the memory banks (cf. Fig. 1.b).

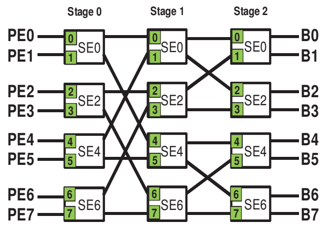


a. Targeted architecture b. Algorithm principle

Fig. 1. Proposed Approach

A. Mapping approach

In-place conflict-free memory mapping approach ensures that assignment of a data element in a memory element remains the same during all the processing steps of the data block. In [12] we have introduced a polynomial time algorithm which is based on Euler partitioning. However, the target network is a fully connected i.e. supports all the possible permutations. In this paper, we consider a ButterFly Network (BFN) as target. BFN is far less complex than fully connected Benes or crossbar networks, and provides access to the memory with reduced latency. In BFN, two consecutive PEs cannot access consecutive memory banks. Thus the algorithm proposed in [12] has to be modified to find conflict free memory mapping respecting the permutation characteristics of a targeted interconnection network. This new method will be referred in this paper as *Architecture Oriented Memory Mapping* approach (AOMM).



Switch	SE0		SE2		SE4		SE6	
Location	0	1	2	3	4	5	6	7
T1	0	5	10	15	20	25	30	35
T2	1	6	11	16	21	26	31	36
T3	2	7	12	17	22	27	32	37
T4	3	8	13	18	23	28	33	38
T5	4	9	14	19	24	29	34	39
T6	39	35	34	38	36	37	32	33
T7	25	27	26	30	28	29	24	31
T8	17	21	20	22	18	19	16	23
T9	9	11	10	14	12	13	8	15
T10	1	5	4	6	2	3	0	7

a. BNF with placement elements b. Matrix representation

Fig. 2 : AOMM approach using Two Matrices

AOMM uses matrix representation to find mapping solution. To better understand the placement constraints, a butterfly network is shown in Fig. 2.a where the small block in each switch represents the location within the matrix at which each data is currently placed. Matrix representation used in AOMM approach is shown in Fig. 2.b. The columns of this matrix are divided into four partitions each corresponding to the switch used in each stage of AOMM algorithm. Location column of each switch within this matrix correspond to the placement element of each switch used in butterfly network.

While the algorithm from [12] is based on a single matrix, AOMM approach uses two matrices in which element placement representations are generated for each switch. Using a single matrix requires the search algorithm to traverse the matrix to find the next valid element of the partition. So, in order to further reduce computational complexity of the algorithm and thus finally runtime, AOMM approach uses two matrices. Hence, the first matrix of AOMM is called "*Euler Matrix*" whereas the second matrix is named "*Euler Matrix Comp*" as shown in Fig 3. At the beginning of the algorithm, Euler matrix contains elements with respect to their placement on butterfly network. For example, data *0* is consumed by PE_0 that is connected with placement element *1* of SE_0 at T_1. Therefore, data *0* is placed at location *0* of Euler matrix.

Switch	SE0		SE2		SE4		SE6	
Location	0	1	2	3	4	5	6	7
T1	0	5	10	15	20	25	30	35
T2	1	6	11	16	21	26	31	36
T3	2	7	12	17	22	27	32	37
T4	3	8	13	18	23	28	33	38
T5	4	9	14	19	24	29	34	39
T6	39	35	34	38	36	37	32	33
T7	25	27	26	30	28	29	24	31
T8	17	21	20	22	18	19	16	23
T9	9	11	10	14	12	13	8	15
T10	1	5	4	6	2	3	0	7

Switch	SE0		SE2		SE4		SE6	
Location	0	1	2	3	4	5	6	7
T1								
T2								
T3								
T4								
T5								
T6								
T7								
T8								
T9								
T10								

a. Euler Matrix b. Euler Matrix Comp

Fig 3 : AOMM Euler Matrices

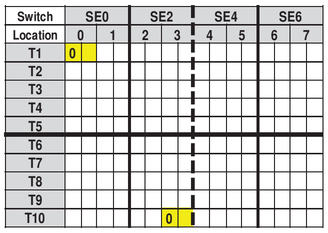
After the matrix generation, data are alternatively placed in the two matrices by using the placement algorithm of Fig. 4.

```
In each partition DO
        offset = partiton_size/2
        half_of_partition = start_location+offset
        If (Par_0) then
            If (S < half_of_partition) then Location = SE
            Else Location = SE – offset + 1
        Else
            If (S < half_of_partition) Then Location = SE + offset
            Else Location = SE + 1
```

Fig. 4. Placement algorithm

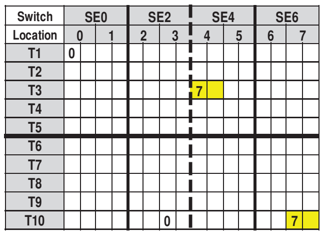
For the 1st iteration, the initial values are *partition_size=8*, *start_location 0*; therefore *offset=4* and *half_of_partition=4*. We start with data *0* at T_1 connected with SE_0 and we place it in *Par_1*. Since SE_0 less than (*half_of_partiton=4*), therefore, location of data *0* (T_1) is *0*. As a consequence the algorithm places data *0* at location *0* of T_1 in Euler comp matrix.
The 2nd instance of data *0* exists at T_9 and is connected with SE_6. Since SE_6 is greater than *half_of_partition*, therefore location of data $0(T_9)=6-4+1=3$ as shown in Fig. 5.a. The 2nd data connected with SE_6 (i.e. data *7*) is now placed in the partition *Par_2*. Since SE_6 > *half_of_partition*, then location of data $7(T_9)$ = 6+1 = *7*. Similarly, the 2nd access of data exists

at T_3 and is connected with SE_0. Since SE_0<6, then location of data $7(T_3) = 0+4 = 4$ (cf. Fig. 5.b).



Switch	SE0		SE2		SE4		SE6	
Location	0	1	2	3	4	5	6	7
T1	0							
T2								
T3								
T4								
T5								
T6								
T7								
T8								
T9								
T10				0				

a. Euler Comp Matrix after step 1



Switch	SE0		SE2		SE4		SE6	
Location	0	1	2	3	4	5	6	7
T1	0							
T2								
T3					7			
T4								
T5								
T6								
T7								
T8								
T9								
T10				0				7

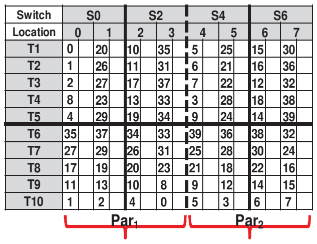
b. Euler Comp Matrix after step 2

Fig. 5. Execution of AOMM approach using Two Matrices

The algorithm continues until all the data is placed in Euler matrix comp (cf. Fig. 6.a). For the 2nd iteration, the algorithm calculates the new values of *offset* and *half_of_partition* for each partition:

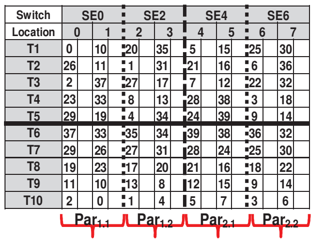
For the partition *Par_1*: *partition_size=4* and *start_location=0*; Therefore *offset=2* and *half_of_partition=2*.

For the partition *Par_2: partition_size=4 and start_location=4*; Therefore, *offset=2* and *half_of_partition=6*.



Switch	S0		S2		S4		S6	
Location	0	1	2	3	4	5	6	7
T1	0	20	10	35	5	25	15	30
T2	1	26	11	31	6	21	16	36
T3	2	27	17	37	7	22	12	32
T4	8	23	13	33	3	28	18	38
T5	4	29	19	34	9	24	14	39
T6	35	37	34	33	39	36	38	32
T7	27	29	26	31	25	28	30	24
T8	17	19	20	23	21	18	22	16
T9	11	13	10	8	9	12	14	15
T10	1	2	4	0	5	3	6	7
	Par_1				Par_2			

a. Euler Matrix Comp

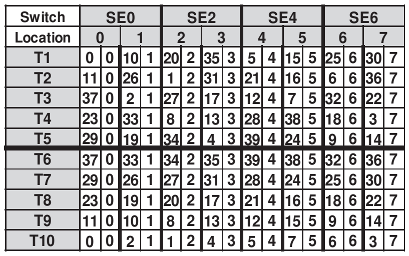


Switch	SE0		SE2		SE4		SE6	
Location	0	1	2	3	4	5	6	7
T1	0	10	20	35	5	15	25	30
T2	26	11	1	31	21	16	6	36
T3	2	37	27	17	7	12	22	32
T4	23	33	8	13	28	38	3	18
T5	29	19	4	34	24	39	9	14
T6	37	33	35	34	39	38	36	32
T7	29	26	27	31	28	24	25	30
T8	19	23	17	20	21	16	18	22
T9	11	10	13	8	12	15	9	14
T10	2	0	1	4	5	7	3	6
	$\text{Par}_{1.1}$		$\text{Par}_{1.2}$		$\text{Par}_{2.1}$		$\text{Par}_{2.2}$	

b. Euler Matrix

Fig. 6 : Matrices during the 2nd iteration of AOMM

Then with these parameters, the algorithm finds new location for each data by using the placement algorithm, and transfers the resulting placement from Euler Matrix Comp to Euler Matrix. There are two sub-iterations during this 2nd iteration as explained in Euler Partition Algorithm. The first sub-iteration places data connected with SE_0 & SE_2 into *Par_1.1* & *Par_1.2* of Euler matrix; whereas the second sub-iteration transfers the data connected with SE_4 & SE_6 into *Par_2.1* & *Par_2.2* of the Euler Matrix. After the 2nd iteration, SE_0 belongs to *Par_1.1*, SE_2 belongs to *Par_1.2*, SE_4 belong to *Par_2.1* and SE_6 belong to *Par_2.2* as shown in Fig. 6.b.



Switch	SE0				SE2				SE4				SE6			
Location	0		1		2		3		4		5		6		7	
T1	0	0	10	1	20	2	35	3	5	4	15	5	25	6	30	7
T2	11	0	26	1	1	2	31	3	21	4	16	5	6	6	36	7
T3	37	0	2	1	27	2	17	3	12	4	7	5	32	6	22	7
T4	23	0	33	1	8	2	13	3	28	4	38	5	18	6	3	7
T5	29	0	19	1	34	2	4	3	39	4	24	5	9	6	14	7
T6	37	0	33	1	34	2	35	3	39	4	38	5	32	6	36	7
T7	29	0	26	1	27	2	31	3	28	4	24	5	25	6	30	7
T8	23	0	19	1	20	2	17	3	21	4	16	5	18	6	22	7
T9	11	0	10	1	8	2	13	3	12	4	15	5	9	6	14	7
T10	0	0	2	1	1	2	4	3	5	4	7	5	6	6	3	7

Fig. 7 : Final Euler matrix thanks to AOMM approach

Afterward, the algorithm transfers data from Euler matrix to Euler matrix Comp by using the placement algorithm until each column of the matrices is related to a single partition. Then, data elements at location *0* are mapped in memory bank *B0*, data elements at location *1* are mapped in memory bank *B1* and so on...

The resultant memory mapping is: *B0*={0,37,23,29,11}; *B1*={2,33,19,26,10}; *B2*={20,1,27,8,34}; *B3*={35,31,17,13,4}; *B4*={5,21,12,28,39}; *B5*={15,16,7,38,24}; *B6*={25,6,32,18,9}; *B7*={30,36,22,3,14} (see Fig. 7).

Finally, this *virtual memory mapping* that is generated on-chip at runtime is used to generate easily the command words driving the network and the memory banks.

B. Dynamically Reconfigurable Architecture

To determine addressing and control information, traditional approach first determines the memory mapping for a particular block or interleaver. Based on this memory mapping and on the interconnection network used in the architecture, all the addressing and control information are generated and stored in its respective ROM to process a code block.

In our approach, a *Virtual Mapping Table* (VMT) is used to store the mapping information for each data element. For example with no loss of generality, in case of turbo-codes, this information is retrieved for both natural and interleaved order processing. For natural order, the architecture retrieves PE mapping values corresponding to data required concurrently at T_i and concatenates them to construct a command word. The formula used to generate addresses for retrieving mapping from VMT in natural order at T_i is:

$T_i(j) = (i - 1) + (j * n)$

where, n = Total Number of time instance in Natural order

$i = 1, 2, \ldots\ldots, n$ and $j = 0, 1, \ldots\ldots, (PE-1)$

The complete formula to construct a command word for crossbar network at each time instance T_i is given below (& means *concatenation*):

$Cmd_{Ti} = (i-1)+(0*n)$ & $(i-1)+(1*n)$ &… & $(i-1)+(j*n)$

For interleaved order, the approach uses data order provided by architecture to retrieve the mapping information. For example, at time instance T_6, we retrieve mapping information for *39,35,34,38,36,37,32,33* directly by accessing concurrently VMT and by concatenating these mapping information to construct the command word at T_6.

For address generation, we cleverly store data in each RAM so that the addresses can be generated online for both natural and interleaved order. For that purpose, data are stored in RAM respecting the data concurrency in natural order. In this scheme, data that need to be accessed in parallel at any time instance in natural order are given the same address. For example, the data that need to be accessed at T_1 are placed at address *0* of each RAM. After data allocation, a simple counter is used to generate addresses at each time instance in natural order whereas a simple modulo operations is required to generate address for each data in interleaved order. The formula to generate address for each data is given as,

Address_of_a_data = data ***modulo*** *n*

For our example, at T_6 in interleaved order, Address_of_39 = 39 mod 5 = 4 ; Address_of_35 = 35 mod 5 = 0 ; …

By using this formula, we compute the addresses of all the data used at any time instance in the interleaved order and concatenate them to construct addressing word used at that time instance. Our approach significantly reduces the memory requirement since only mapping table that has a size of $K*|log_2(PE)|$ is required to generate both addressing and command words (instead of $2*K*\ log_2(PE)|$ for network ROM and $2*K*|log_2(PE)|$ for addressing ROM.

III. Experimental Results

As explained in the previous section, the architecture developed in this work is based upon a mapping unit and associated memories used to execute algorithm and to store mapping information. In our experiments, we implemented mapping unit either in software that runs on embedded processor or developed a dedicated hardware that generates mapping information on runtime.

The embedded processor that we used to test our algorithm is NIOS II soft processor used in Altera FPGAs. NIOS II has been implemented on Cyclone-III NIOS II Embedded Evolution Kit with Processor clock frequency of 195MHz and System clock frequency of 50MHz. Latency to compute mapping information for different block lengths and parallelisms for HSPA interleaver used in 3GPP-WCDMA standard are shown in Fig. 8.

Block Length	256				1024				2048			
Parallelism	4	8	16	32	4	8	16	32	4	8	16	32
Time (in Seconds)	0,01	0,02	0,03	0,03	0,05	0,08	0,11	0,16	0,11	0,17	0,24	0,33
Block Length	3072				4096				5120			
Parallelism	4	8	16	32	4	8	16	32	4	8	16	32
Time (in Seconds)	0,17	0,24	0,36	0,51	0,23	0,35	0,49	0,71	0,35	0,42	0,61	0,87

Fig. 8 : Algorithm latency for embedded processor NIOS II

For hardware accelerator, the algorithm is implemented on VHDL and synthesized on Virtex-5 FX70T FPGA with system clock frequency of 100MHz. From the explanation of the algorithm, it is clear that main complexity of the algorithm arises from requirement of the memory to store updated matrices. This results in controller with small area and cost. For this report, single port memory is used to store matrices and latency of the algorithm for different block lengths and parallelism of 8 (PE = 8) are shown in Fig. 9. Thanks to our modified AOMM algorithm, there is no latency for generating routing information or control bits for the butterfly network since routing information is generated during the execution of the algorithm. This is another advantage of our approach that reduces the latency associated with computing the routing information related to the network after finding the memory mapping online. The latencies shown in the figure are in compliance with HSPA standard that requires 50 to 100 ms of latency for call setup and data transfer.

Block Length	256	1024	2048	3072	4096	5120
Time (in *u*sec)	19	76	153	230	307	384

Fig. 9 : Architecture's latency with PE = 8

From synthesis results, it is clear that only 64 out of 44800 available LUTs are used to generate controller. Moreover, only 10 out of 148 available 36 Kbits block RAMs are used to generate memory requires to implement our algorithm. The results show that our algorithm can easily be embedded on FPGA and required the area of the controller and associated memory is quite small compared to overall FPGA capacity.

IV. Conclusion

From the implementation of AOMM approach on hardware, it is clear that latency required to implement our algorithm is in compliance of current telecommunication standards. Moreover, low architectural cost of the algorithm makes it suitable to embed on telecommunication and signal processing system for supporting multiple applications and standards.

REFERENCES

[1] C. Berrou, A. Glavieux and P. Thitimajshima, "Near-Shannon limit error-correcting coding and decoding: Turbo codes", *IEEE Int. Conf. Com..*, vol.2, 1993.

[2] R.G. Gallager, "Low-Density Parity-Check Codes*", Cambridge, MA: MIT Press*, 1963.

[3] O.Y. Takeshita, "On maximum contention-free interleavers and permutation polynomials over integer rings", *IEEE Trans. Inf. Theory*, vol. 52, no. 3, March 2006.

[4] M. Thul, N. Wehn and L. Rao, "Enabling high-speed turbo decoding through concurrent interleaving", *IEEE Int. Symp. on Cir. And Sys.*, vol. 1, pp. 897–900, May 2002.

[5] M. Thul, F. Gilbert and N. Wehn, "Concurrent interleaving architectures for high-throughput channel coding", *IEEE Int. Conf. on Acou., Spe. and Sig. Proc.,* vol.2, pp. 613–616, March 2003.

[6] H. Moussa, O. Muller, A. Baghdadi and M. Jézéquel, "Butterfly and Benes-based on-chip communication networks for multiprocessor turbo decoding", *IEEE Design. Auto. and Test in Eu.*, April 2007.

[7] H. Moussa, A. Baghdadi and M. Jézéquel, "Binary de Bruijn interconnection network for a flexible LDPC/turbo decoder", *IEEE Int. Symp. on Cir. And Syst.*, pp. 97–100, May 2008.

[8] A. Tarable, S. Benedetto and G. Montorsi, "Mapping interleaving laws to parallel turbo and LDPC decoder architectures", *IEEE Trans. On Inf. Theory*, vol. 50, no.9*,* September 2004.

[9] J. Yang, "Parallel Interleavers Through Optimized Memory Address Remapping", *IEEE Trans. VLSI Systems,* vol.18, no.6, pp.978-987, June 2010.

[10] C. Chavet, P. Coussy P. Urard and E. Martin, "Static Address Generation Easing: a Design Methodology for Parallel Interleaver Architecture", *IEEE Int. Conf. on Acou. Spe. and Sig. Proc.,* May 2010.

[11] J.L. Gross and J. Yellen, "Handbook of Graph Theory", *353, CRC Press*. 2003.

[12] A. Sani, C. Chavet and P. Coussy, "A First Step Toward On-Chip Memory Mapping for Parallel Turbo and LDPC Decoders: A Polynomial Time Mapping Algorithm", *IEEE Trans. on Sig. Proc.,* vol.61, issue:16, pp.4127 - 4140, 2013.

[13] J. Duato, S. Yalamanchili and L. Ni, "Interconnection Networks an Engineering Approach", *Morgan Kaufman Publishers,* 2003, pp.20-30.

[14] K.Y. Lee, "A new Benes network control algorithm", *IEEE Trans. on Computer,* vol. C-36, no. 6, pp.768–772, June 1987.

[15] S. Ur Rehman, A. Sani, P. Coussy and C. Chavet, "On-Chip Implementation Of Memory Mapping Algorithm To Support Flexible Decoder Architecture", *IEEE Int. Conf. on Acou., Spe. and Sig. Proc,*March 2013.

[16] C. Chavet and P. Coussy, "A memory Mapping Approach for Parallel Interleaver design with multiples read and write accesses", *IEEE Int. Symp. on Cir. And Syst.*, pp.3168-3171, May 2010.

[17] A. Briki, C. Chavet and P. Coussy, "A Conflict-Free Memory Mapping Approach To Design Parallel Hardware Interleaver Architectures With Optimized Network And Controller", *IEEE Int. Workshop on Sig. Proc. Sys.*, October 2013

[18] S. Ur Rehman, A. Sani, P. Coussy and C. Chavet, "Embedding Polynomial Time Memory Mapping And Routing Algorithms On-Chip To Design Configurable Decoder Architectures", *IEEE Int. Conf. on Acou., Spe. and Sig. Proc,* March 2014.