Milcom 04 paper ID#1158
A New Class of Turbo-like Codes with Efficient and Practical High-Speed Decoders

Aliazam Abbasfar[1], Dariush Divsalar[2], and Kung Yao[1]
[1]Dept. of Elec. Eng., UCLA, Los Angeles, USA
[2]Jet Propulsion Laboratory, Pasadena, USA
(Point of contact Dariush Divsalar)
(Unclassified)
abbasfar@ee.ucla.edu, Dariush.Divsalar@jpl.nasa.gov, yao@ee.ucla.edu

Abstract— **Turbo codes not only achieve near Shannon-capacity performance, but also have decoders with modest complexity, which is crucial for implementation. So far efficient architectures for decoding of turbo codes have been proposed that is suitable for serial processing. In this paper a novel architecture for very high-speed turbo decoder is presented. The performance of this decoder is illustrated and the tradeoff between speed and efficiency is discussed. It is shown that some decoders can run faster by some order of magnitude while maintaining almost the same processing load.**

The memory access poses as bottleneck in practical implementation of such decoder. This problem is addressed by introduction of a new structure for the interleaver. This structure not only makes the implementation of the high-speed decoder practically feasible, but also lowers the latency of the decoder without extra hardware. It is shown that such an interleaver can be designed to have good BER performance as well. We also present a fast algorithm to design such an interleaver, which can be used to design S-random interleavers as well. It has been shown that the new interleaver structure can perform as well as other good interleavers.

Graphical interpretation of the proposed code structure is provided, which leads to the introduction of a new class of Turbo-like codes that have high-speed decoding capability. A general architecture for high-speed decoder of the codes in this class is presented. Regularity and simplicity of the interleaver makes it the architecture of choice for VLSI implementation of very high-speed decoders.

Keywords- Turbo decoder; parallelization; high speed Turbo decoder; Interleaver design;

I. INTRODUCTION

Recently, some new classes of channel encoders have been introduced that achieve near Shannon-capacity performance. Turbo codes [4] and low-density parity-check (LDPC) Codes are the most important examples. The basic property of these codes is the capability of iterative decoding. The iterative algorithms can be viewed as a probability or "belief" propagation algorithm, which is based on message passing [1].

Although iterative decoding has a parallel nature for LDPC codes, for turbo codes it is very attractive in a serial way as they use BCJR algorithm, which is a recursive algorithm. Although the message-passing algorithm can be parallelized in theory, it is quite inefficient and impractical for implementation [2]. In [2] a concurrent turbo decoder is studied, which can run by some orders of magnitude faster than its serial counterpart. However, the number of components used for processing is so large that makes it quite impractical. Moreover, the processing load has been increased dramatically, which translates to low efficiency. Another approach that is proposed in the literature is using overlapping windows [3]. However, for a very high-speed decoder the extra processing load for overlapping bits causes inefficiency and irregularity.

In this paper we first propose a method that make parallel turbo decoding feasible, while efficiency of the decoder is maintained. Then, we extend the idea to a broader range of Turbo-like codes that create a new class of codes. All the codes in this class are not only regular and parallelizable, but also have practical high-speed decoders. A general architecture for high speed decoding is presented for these codes.

In section II we describe the decoding algorithm for turbo decoders. In section III the proposed architecture for the high-speed decoder is described and the tradeoff between the speed gain and the efficiency is discussed. In section IV we present a new structure for the interleaver that makes the implementation of the proposed decoder practically feasible. The performance of the proposed decoder is discussed in section V and simulation results are illustrated. In section VI a new class of turbo-like codes are introduced that have capability of high-speed decoding.

II. TURBO CODE

Turbo code was introduced in [4]. Berrou, et al. presented the Parallel Concatenated Convolutional Code, (PCCC) and the iterative decoding algorithm. Later Serial Concatenated Convolutional Codes (SCCC) was presented in [5]. PCCC has been remained the most popular type of turbo code, which has

This work is partially supported by an UC CoRE sponsored by ST Microelectronics. This work was performed at the Jet Propulsion Laboratory, California Institute of Technology under contract with the NASA.

been adopted in UMTS standards as channel coding scheme. In the following we briefly describe the PCCC encoder and its iterative decoder.

A PCCC is constructed from two or more parallel convolutional encoders that are working on the input sequence and its permuted versions in parallel. Each convolutional code is called a constituent code. Without any loss of generality, the PCCC with two constituent codes is studied in the sequel. The generalization of the proposed method described for a simple PCCC will be given later for a wide range of Turbo-like codes. Fig. 1 depicts the structure of a PCCC with two constituent codes. The block denoted by I is the interleaver, which permutes the input sequence with a predefined random pattern.

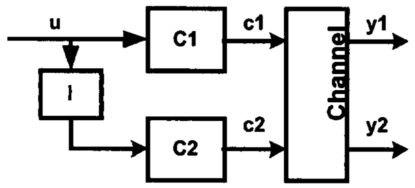

Figure 1. The structure of a PCCC encoder

The iterative decoding algorithm is based on Maximum-A-Posteriori (MAP) decision of the input sequence. However, since it is difficult to find the MAP solution by considering all the observations at the same time, the MAP decoding is performed on the observations of each constituent code separately. Since two codes have been produced from one input sequence, the A-Posteriori-Probability (APP) of data bits coming from the first decoder can be used by the second decoder and vice versa. The APP information passed between the constituent codes is called extrinsic information. Therefore the decoding process is carried out iteratively. In [5] a general unit, called SISO, is introduced that generates the APPs for a convolutional code in the most general case.

Since the second constituent code is using the permuted version of the input sequence, therefore, extrinsic information also should be permuted before being used by the second decoder. Likewise, the extrinsic information of the second decoder is to be permuted in reverse order for the next iteration of the first decoder. Fig. 2 shows the iterative decoding bock diagram. As we see two SISOs are used to process the constituent codes.

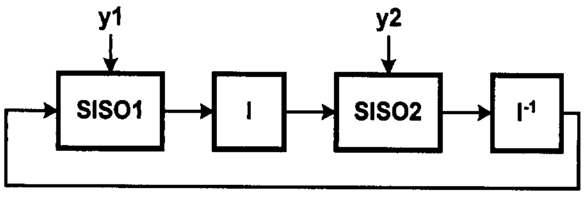

Figure 2. The iterative decoding block diagram

An efficient algorithm for APP computation of a convolutional code is known as BCJR algorithm [6]. In this algorithm A-Posteriori-Probabilities for a time-invariant trellis encoder can be computed with a complexity that depends linearly on the number of states and also on the size of input sequence. In fact, SISO is a block that implements the BCJR algorithm. Here we briefly describe the structure of this algorithm. The main three steps of this algorithm are as follow:

Forward recursion: In this step we compute the likelihood of all the states in the trellis given the past observations. Starting from a known state; the likelihood of that state is 1 and others are zero; we will go ahead along the trellis and compute the likelihood of all the states in one trellis section from the likelihood of the states in the previous trellis section. The computational complexity increases with the number of states. This iterative scheme is continued until likelihoods of all the states, which are called alpha variables, are computed in the forward direction.

Backward recursion: This step is quite similar to the forward recursion. Starting from a known state at the end of the block, we compute the likelihood of previous states in one trellis section. Therefore we compute the likelihood of all the states in the trellis given the future observations, which are called beta variables. This iterative processing is continued until the beginning of the trellis.

Output computation: Once the forward and backward likelihoods of the states are computed, the extrinsic information can be computed from them. The extrinsic information can be viewed as the likelihood of each bit given the observations.

The block diagram of a SISO for a convolutional code of length N is sketched in Figure 3. The inputs to the SISO block are the observations (r_1 or r_2), initial values for alpha and beta variables, (a_0 and b_N) and the extrinsics coming from other SISO. The outputs are the alpha and beta variables at the end of forward and backward recursions, which are not used any more, and the new extrinsics that will pass to the other SISO.

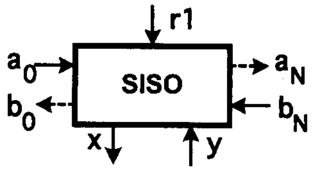

Figure 3. Block diagram of the SISO

In the traditional realization of the SISO, the timing scheduling for the three mentioned steps is as follows. The backward recursion is done completely for the entire block and all beta variables are stored in a memory. Then, the forward recursion starts from the first trellis section and computes the alpha variables one by one. Since at this time both alpha and beta variables are available for the first trellis section, the extrinsic for the first bit is computed at this time. Therefore the extrinsic computation is done along with the forward recursion. The sequence of variables in time is as follows:

Backward:	$y_{N-1}\, y_{N-2} \ldots y_1\, y_0$	
	$b_N\ \ b_{N-1} \ldots b_2\, b_1\, b_0$	
Forward:		$y_0\, y_1 \ldots y_{N-2}\, y_{N-1}$
		$a_0\, a_1 \ldots a_{N-2}\, a_{N-1}\, a_N$
Output:		$x_0\, x_1 \ldots x_{N-2}\, x_{N-1}$

Where alpha and beta variables are denoted by a and b, and incoming and outgoing extrinsics are denoted by y and x. We could exchange the order in which forward and backward recursion is done. However, this scheduling outputs the extrinsics in reverse order. It should be emphasized that since the BCJR algorithm is a recursive one, the processing is done serially.

III. Parallel Turbo Decoder

In this section we present a novel method for iteratively decoding the turbo codes. Although this method is applicable for every turbo code, we will explain it in the case of a block PCCC code.

The algorithm is as following. First of all, the received data for each constituent codes are divided into several contiguous non-overlapping sub-blocks; so called windows. Then, each window is decoded separately in parallel using the BCJR algorithm. In other words, each window processor is a decoder for a block of the information bits. However, the initial values for alpha and beta variables come from previous iteration of adjacent windows. Since all the windows are being processed at the same time, in the next iteration the initial values are ready to load for all of them. Therefore, there is no extra processing needed for the initialization of state probabilities at each iteration. The size of windows is a very important parameter that will be discussed later. Fig. 3 shows the structure of the decoder.

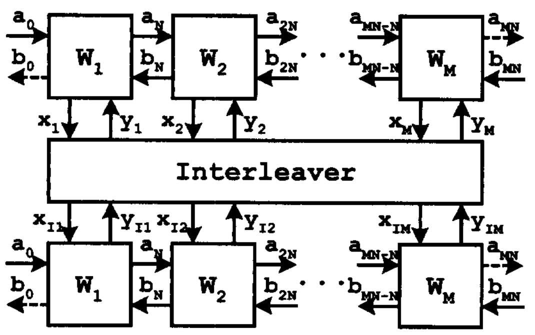

Figure 4. Parallel turbo decoder structure

The optimum way to process a window is the serial processing using forward and backward recursions; i.e. BCJR algorithm. Therefore each window processor is a SISO.

The proposed structure stems from the message-passing algorithm itself. An example of the graph of the PCCC is depicted in Figure 5. The graph is partitioned into several (M) sub-graphs, which correspond to different windows. Each sub-graph still represents a convolutional code, but their initial and end states are determined by adjacent windows.

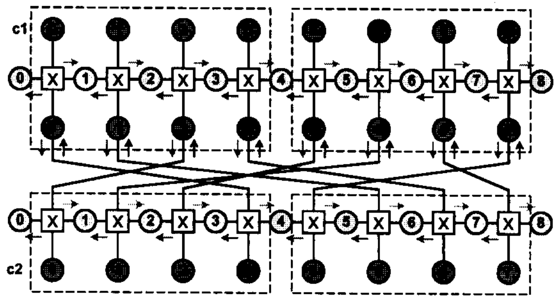

Figure 5. Partitioned graph of a simple PCCC

There are two types of messages that are communicated between sub-graphs. First, the messages associated with the information bits, i.e. the extrinsic information, which are communicated between two constituent codes in the traditional approach. Second, the messages that are related to the states in window boundaries, we call them state messages. In fact, we have introduced new messages that are passed between sub-graphs at each iteration. These messages are the same as alpha and beta variables that are computed in forward and backward recursion of the BCJR algorithm. In the first iteration there is no prior knowledge available about the state probabilities. Therefore the messages are set to equal probability for all the states. In each iteration, these messages are updated and passed across the border of adjacent partitions.

Partitioning of the graph of code helps us to parallelize the decoding of one constituent code. The processing for each partition is usually done serially. This makes the algorithm more efficient. Moreover, the hardware complexity is lowered. In fact, we have used a serial-parallel scheduling for the message-passing algorithm. In other words, partitioning provides a method for serial-parallel method for iterative decoding, which achieves both speed gain and low complexity.

Table I shows the parameters of a decoder. For window size at two extremes, the approach is reduces to known methods. If window size is B, the number of windows is 1, it turns out to the sequential approach. If the window size is 1, the architecture reduces to what was proposed in [2]. It should be noted that the memory requirement for all cases is the same.

TABLE I. The Decoder Parameters

Parameter	***Definition***
N	Window size
M	Number of windows (SISOs)
B = M x N	Block size
I	Number of iterations
$T_W = 2N \times T_{clk}$	Window Processing Time
$T = 2I \times T_W$	Processing Time (Latency)
P = k 2I B	Processing Load

Processing time is the time needed to decode one block. Since all windows are processed at the same time, each SISO is done after T_W. We assume that all computations associated with one trellis section is done in one clock cycle (T_{clk}). We have I iterations and each iteration has two constituent codes, so it takes 2I x T_W to complete the decoding. It is worth mentioning that the processing time determines the latency as well. Therefore any speed gain is equivalent to lower latency.

Processing load is the amount of computations that we need. The processing load for each SISO is proportional to the number of the state variables. Hence, it is kB, where k is the constant and depends on the complexity of the convolutional code. It should be noted that processing load in both serial and parallel SISO are the same. Therefore the total processing load is 2I x kB.

The timing diagram can be simplified by using vector notation, which is shown as following:

Backward:	$\mathbf{y}_{N-1}\ \mathbf{y}_{N-2} \ldots \mathbf{y}_1\ \mathbf{y}_0$	
	$\mathbf{b}_N\ \ \mathbf{b}_{N-1} \ldots \mathbf{b}_2\ \mathbf{b}_1\ \mathbf{b}_0$	
Forward:		$\mathbf{y}_0\ \mathbf{y}_1 \ldots \mathbf{y}_{N-2}\,\mathbf{y}_{N-1}$
		$\mathbf{a}_0\ \mathbf{a}_1 \ldots \mathbf{a}_{N-2}\,\mathbf{a}_{N-1}\,\mathbf{a}_N$
Output:		$\mathbf{x}_0\ \mathbf{x}_1 \ldots \mathbf{x}_{N-2}\,\mathbf{x}_{N-1}$

The variables that computed at the same time are simply replaced with a vector. Each vector has M elements, which belong to different window processors (SISOs). For example, we have $\mathbf{a}_0 = [a_0\ a_N\ a_{2N} \ldots a_{MN-N}]^T$ and $\mathbf{b}_0 = [b_0\ b_N\ b_{2N} \ldots b_{MN-N}]^T$. This notation is the generalization of the serial decoder. It also will help to appreciate the new interleaver structure for the parallel decoder discussed later.

IV. Speed Gain vs. Efficiency

A. *Definitions:*

Two characteristic factors should be studied as performance figures. One is the speed gain and the other is the efficiency. In ideal parallelization the efficiency is always 1. It means that there is no extra processing load needed for parallel processing.

They are defined as following:

$$\textit{Speed gain} = T_0/T$$

$$\textit{Efficiency} = P_0/P$$

Where T_0 and P_0 are the processing time and processing load for the serial approach, i.e. W=B case. The factors can be further simplified to:

$$\textit{Speed gain} = M \times I_0/I$$

$$\textit{Efficiency} = I_0/I$$

This is very interesting result. The speed gain and the efficiency are proportional to the ratio between number of iterations needed for serial case and parallel case. If the number of iterations required for the parallel case is the same as the serial case, we enjoy a speed gain of M without degrading the efficiency, which is ideal parallelization. Therefore we should look at the number of iterations required for a certain performance to further quantify the characteristic factors. In next section we will investigate these factors with some simulations.

B. *Simulations results*

For simulations a PCCC with block size of 4800 is chosen. The first constituent code is a rate one-half systematic code and the second code is a rate one non-systematic recursive code. The feed forward and feedback polynomials are the same for both codes and are $1+D+D^3$ and $1+D^2+D^3$ respectively. Thus coding rate is 1/3. The simulated channel is an AWGN channel. The bit error rate performance of the proposed high-speed decoder has been simulated for window sizes of N=256,128, 64, 48, 32, 16, 8, 4, 2, and 1.

The first observation was that this structure does not sacrifice performance for speed. We can always increase the maximum number of iterations to get similar performance as of the serial decoder. The maximum number of iterations for each case is chosen such that the BER performance of the decoder equals that of the serial decoder after 10 iterations (I_0=10). Figure 6. shows the BER performance of the decoder with different window sizes. The curves are almost indistinguishable.

However, in practice, the iterations are stopped based on a criterion that shows the decoded data is reliable or correct. We have simulated such a stopping criterion in order to obtain the average number of iterations needed. The stopping rule that we use is the equality between the results of two consecutive iterations. The average number of iterations is used for the efficiency computation. The average number of iterations for low signal to noise ratio is the maximum number of iterations for each window size.

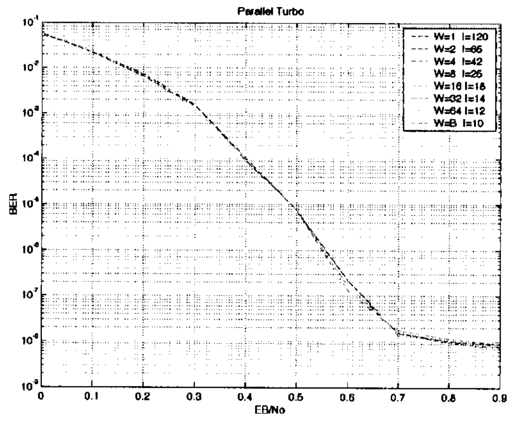

Figure 6. Performances of parallel decoder

Efficiency and speed gain of the parallel decoder with different window sizes is shown in Figure 7. It clearly shows that we have to pay some penalty in order to achieve the speed gain. Also we observe that the efficiency of parallel decoder decreases gracefully for window sizes greater than 32. The efficiency is degraded dramatically for very small windows, which prohibits us to get speed gain as well. However, the speed gain is a decreasing function.

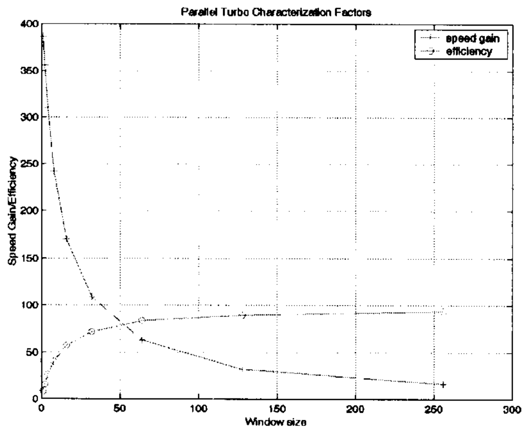

Figure 7. Efficiency and speed gain

As a summary, in Table II the maximum number of iterations, the average number of iterations, and the characteristic factors are tabulated for different window sizes at $E_b/N_0 = 0.7$ (BER = 1e-8).

TABLE II. CHARACTERISTIC FACTORS FOR THE PARALLEL DECODER @SNR= 0.7 dB (BER=10E-8)

Window Size	*Max # of iterations*	*Ave. # of iterations*	*Speed Gain*	*Efficiency %*
64	12	5.0	63	84
32	14	5.8	109	72
16	18	7.4	170	57
8	25	10.4	242	40
4	42	16.3	310	26
2	65	28.3	356	15
1	120	52.0	386	8

Efficiency curves with respect to SNR are illustrated in Figure 8. The interesting observation in the efficiency curves is flatness of the curves. In other words, the efficiency of the parallel decoder is almost constant in all SNR. This observation translates to almost constant speed gain over the whole SNR range.

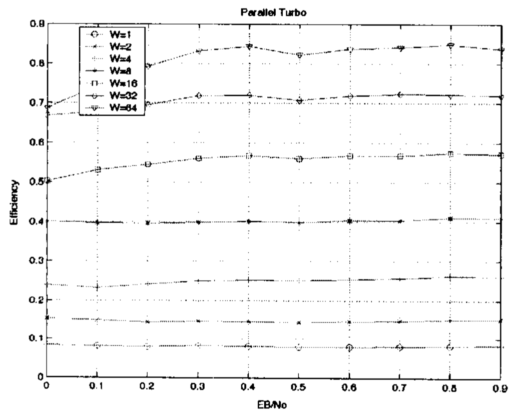

Figure 8. Efficiency vs. signal to noise ratio.

V. INTERLEAVER DESIGN

In traditional decoder, the extrinsics are usually stored in a memory whose locations is accessed one at a time. The order in which the extrinsic memory is accessed is different for two constituent codes because of the interleaver. It is accessed in sequential order for the first constituent code and in interleaved order for the second constituent codes. In practice memory addressing is used to access the memory in desired order.

Although the message-passing algorithm allows us to parallelize the decoding process, accessing so many extrinsics at the same time poses a practical problem. Since M SISOs are running at the same time, M extrinsics are being used simultaneously. Having a memory with M read and write ports is practically infeasible, especially for large M. The solution is to have M sub-memories each one is accessed only by one SISO. This should be true not only for extrinsics in sequential orders but also in interleaved orders. Therefore, it imposes a constraint on the interleaver. In this section we propose an interleaver structure that not only makes this possible, but also lowers the latency of the overall decoder.

To explain the interleaver structure we start with the reverse interleaver in a serial turbo decoder. It is observed that using a reverse interleaver the next iteration can start processing as soon as the first extrinsic is ready. In this case every new computed extrinsic is used right away. This property is only true for the reverse interleaver. The reason for this property is that the sequence of extrinsics computed in the current iterations matches the sequence needed in the backward recursion in the next iteration. The following sequences show the observation and extrinsic sequences used for two constituent codes in one iteration.

$y_{N-1}\, y_{N-2} \ldots y_1\, y_0$
$b_N\;\; b_{N-1} \ldots b_2\, b_1\, b_0$
$\qquad\qquad y_0\, y_1 \ldots y_{N-2}\, y_{N-1}$
$\qquad\qquad a_0\, a_1 \ldots a_{N-2}\, a_{N-1}\, a_N$
$\qquad\qquad x_0\, x_1 \ldots x_{N-2}\, x_{N-1}$ extrinsic outputs

$\qquad\qquad x_0\;\, x_1 \ldots x_{N-2}\, x_{N-1}$ extrinsic inputs
$\qquad\qquad b_N\, b_{N-1} \ldots b_2\;\, b_1\, b_0$
$\qquad\qquad\qquad x_{N-1}\, x_{N-2} \ldots x_1\, x_0$
$\qquad\qquad\qquad a_0\;\, a_1 \ldots a_{N-2}\, a_{N-1}\, a_N$
$\qquad\qquad\qquad y_{N-1}\, y_{N-2} \ldots y_1\, y_0$

The indices used for alpha and beta variables denote the trellis section number. For extrinsics they denote the bit number in the block. The alpha and beta variables in two iterations are totally independent, although for simplicity the same notation is used for them.

As it is observed from the sequences in Fig. 6, the output sequence of the second decoder is compatible with the input sequence of the first decoder, which means that we can repeat this pipelining process for succeeding iterations as well.

This phenomenon results in a very important advantage: the latency of the decoder decreases by almost a factor of two, which translates to speed gain as well. We have

$$\text{Latency} = (2I+1)\text{x}T_w/2$$

This advantage comes at the expense of running both the forward and backward recursion circuits all the time. There is no extra hardware needed, though.

Despite this advantage, this interleaver is never used for turbo codes. The reason is its poor BER performance. In the sequel we use the reverse interleaver in the context of parallel turbo decoder. By this we get around this problem by incorporating one more permutation while we still exploit the advantages.

For the parallel decoder; the timing diagram for two consecutive iterations with the proposed interleaver is shown in the following.

$y_{N-1}\ y_{N-2} \cdots y_1\ y_0$
$b_N\ \ b_{N-1} \ldots b_2\ b_1\ b_0$
$y_0\ y_1 \cdots y_{N-2}\ y_{N-1}$
$a_0\ a_1 \ldots a_{N-2}\ a_{N-1}\ a_N$
$x_0\ x_1 \ldots x_{N-2}\ x_{N-1}$ extrinsic outputs
6 6 6 6
$\pi_0\ \pi_1 \ldots \pi_{N-2}\ \pi_{N-1}$ interleavers
6 6 6 6
$x_0\ x_1 \ldots x_{N-2}\ x_{N-1}$ extrinsic inputs
$b_N\ b_{N-1} \ldots b_2\ b_1\ b_0$
$x_{N-1}\ x_{N-2} \ldots x_1\ x_0$
$a_0\ a_1 \ldots a_{N-2}\ a_{N-1}\ a_N$
$y_{N-1}\ y_{N-2} \cdots y_1\ y_0$

The idea is to use the same vector with permuted elements. So the x_0 in the next iteration is permuted, $\pi_0(x_0)$, and y_0 will be replaced by $\pi_0^{-1}(x_0)$. If we did not have the $\pi_0, \pi_1, \ldots, \pi_{N-1}$ interleavers, there would be nothing more than M parallel decoders each one working on a separate block. However, the presence of the interleavers creates a randomness that will improve the performance of the code while the architecture of the code is almost intact, i.e. the advantages of the reverse interleaver are still in place.

The permutations are done differently for each vector. Therefore the interleaver block is time-variant, but memory-less. Because the number of parallel blocks is usually small, the interleaver implementation is feasible. The structure of the interleaver can be best understood by organizing the bits in a matrix with each row having the bits processed in a SISO. Each column of this matrix is a vector that is computed at a time. We have a two-step interleaver: a reverse row interleaver and a random column interleaver. If the matrix elements are denoted by $P_{i,j}$, where i=0,1,...,M-1 and j=0,1,..., N-1, then the equivalent interleaver sequence is $\{ Q_n \} = \{ N^*P_{i,j} + N\text{-}j \}$. As an example a turbo code with block length of 20 is decomposed into M=5 sub-blocks of N=4 bits. Therefore, there are 5 SISOs working in parallel; each one works on a block of 4 bits. Table III shows an example of the interleaver in matrix format. The equivalent interleaver is {11, 2, 5, 4, 15, 18, 9, 12, 19, 14, 1, 16, 3, 6, 13, 8, 7, 10, 17, 0}.

TABLE III. AN EXAMPLE OF THE INTERLEAVER

Bit index in SISO	0	1	2	3
Interleaved bit index	3	2	1	0
SISO0 gets from SISO	2	0	1	1
SISO1 gets from SISO	3	4	2	3
SISO2 gets from SISO	4	3	0	4
SISO3 gets from SISO	0	1	3	2
SISO4 gets from SISO	1	2	4	0

In this section we will explain how to design such an interleaver. The algorithm has two main steps: constructing a random interleaver and updating based on a certain constraint. Fig. 8 shows the flowchart of this algorithm.

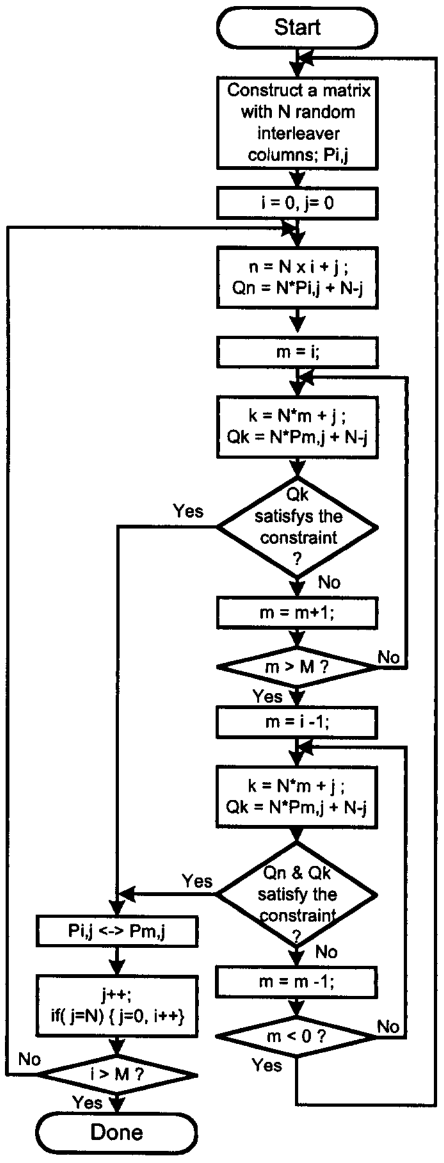

Fig. 8: The flowchart of the algorithm.

Using the matrix format for the interleaver, we initialize the interleaver design by taking a random interleaver for each column. In the next step we will update the interleaver by applying a certain constraint. To update the interleaver we use column-wise bit swapping, which ensures that the structure of the interleaver is preserved.

Since the constraints for interleaved designs are usually applicable for one-dimensional interleaver, to update the interleaver it is best to compute the equivalent interleaver, which can be done on the fly. The constraint that we have used is the spread of the interleaver. In other words, we want to design an S-random interleaver with the proposed structure. An ordinary S-random can be viewed as a special case for this structure, i.e. when we have only one column. Therefore, this algorithm not only presents an algorithm with the proposed structure, but also gives a fast algorithm for designing S-random interleavers.

Starting from the first row, i.e. the first SISO bits, the constraint for each bit is checked given the previously designed bits. If the constraint is met, we go to the next bit. Otherwise we check the remaining elements in the column in place of this bit. If one of them satisfies the constraint, we exchange the indices in the column and go to the next bit. If not, we try exchanging this bit with the previously designed bits in the column. In this situation, when we exchange two bits the constraint for both bits should be satisfied. If none of the previously designed bits can be exchanged with this bit, then the algorithm fails. There are two options available in case the algorithm fails: one is to make the constraint milder and the other one is to redo everything with a new random matrix.

We have observed that this algorithm is very fast for S-random interleaver design when the spread is less than sqrt(B/2). The maximum spread that we can achieve for the structured interleaver is slightly smaller than that of the ordinary one. Therefore, one should expect some degradation in performance.

A. *SIMULATION RESULTS*

For simulations two PCCCs with block sizes of 1024 and 4096 are chosen. The first constituent code is a rate one-half systematic code and the second code is a rate one non-systematic recursive code. The feed forward and feedback polynomials are the same for both codes and are $1+D+D^3$ and $1+D^2+D^3$ respectively. Thus coding rate is 1/3. The simulated channel is an AWGN channel.

Two interleavers with block length of 1024 (32x32) and 4096 (128x32) have been designed with the proposed algorithm. The bit error rate performance of the decoders has been simulated and compared with that of the decoder with S-random interleaver. The maximum number of iterations for each case is 10.

Fig. 9 illustrates the performance comparison. The curves corresponding to the performance of the decoder with S-random interleaver are named with S suffix and the proposed two-dimensional S-random interleaver with S2 suffix. As we see the S-random interleaver has a slightly better error floor. For the decoders with size 4096 the difference between the error floors is more noticeable. However, the codes have equal threshold in both cases. The error floor can be reduced with a more powerful constraint in the interleaver design algorithm.

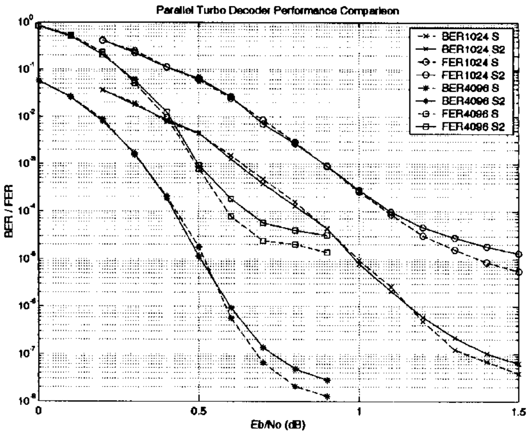

Figure 9. Performance comparison for B=1024

VI. GRAPHICAL VIEW

In this section we look at this problem from a graphical point of view. The parallel turbo decoder comprises M identical sub-graphs that are running in parallel. We put the partitions in parallel planes and then look at the projected graph. The projected graph for turbo code for the example of Table III is shown in Figure 10.

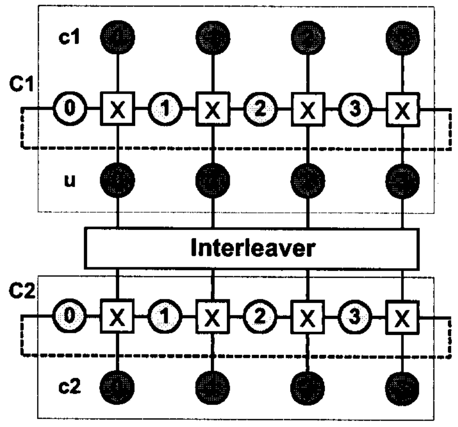

Figure 10. Projected graph

The projected graph can be viewed as the vectorized version of the actual graph. In other words, there is a message vector associated with every edge in the projected graph. There are three types of edges in the projected graph. First, the edges that are inside the partitions, which are shown in solid lines. The associated messages are internal messages that are not communicated with other partitions. In PCCC case, these are alpha and beta variables. Second, the edges that connect adjacent partitions, which are shown in black dashed lines. The associated messages are communicated between adjacent partitions across their border; we call them border messages. Initial values for alpha and beta variable at each iteration are border messages in the parallel PCCC. Third, the edges that connect two constituent codes. These edges are interleaved before reach the other constituent code. The extrinsics are associated with these edges. The structure of memories for extrinsics is such that only one extrinsic vector is accessible at a time. In order to solve the memory access problem, the interleaver preserves the extrinsic vectors in its entirety, but the permutation is allowed within a vector. The permutation within a vector is the permutation among the window processors or different planes in the overall graph. In other words, the interleaver consists of several independent permutations within the extrinsic vectors.

The interesting property about the projected graph is that the proposed interleaver is viewed as some disjoint edges. The projected graph for turbo code for the example of Table III is shown in Figure 11. The red dashed edges indicate that the permutation is allowed within a vector. Therefore, projected graph not only shows the structure of the code, but also its interleaver.

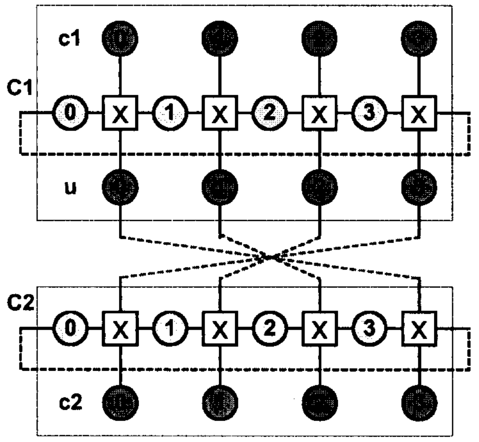

Figure 11. Projected graph with the proposed interleaver

The way edges are connected between two constituent codes in the projected graph is another flexibility in the interleaver design. It is very instructive to note that the reverse interleaver, used for decreasing the latency, is clearly shown in the figure- connecting the edges of two component graphs in reverse order.

VII. Design based on projected graph

In this section design methodology for turbo-like codes based on the projected graph is presented. This approach is very appealing because the resulting code is definitely is parallelizable and the performance of the code can be analyzed very efficiently by its component graphs using density evolution technique. The following section explains how to design LDPC codes with parallel decoding capabilities.

There are two ways that could be pursued in order to design codes based on projected graphs. First approach is the one that was used so far to parallelize the decoder. This is based on partitioning an existing code graph into some sub-graphs. This method works on any regular or semi-regular graph. The projected graph includes one partition of each component code, which is called component graph. The component graphs are connected with edges that represent independent interleavers. To show the broadness of this approach we have illustrated the projected graphs for a PCCC with three constituent codes, a simple SCCC, repeat-accumulate (RA) and irregular-repeat-accumulate (IRA) codes in Figure 12.

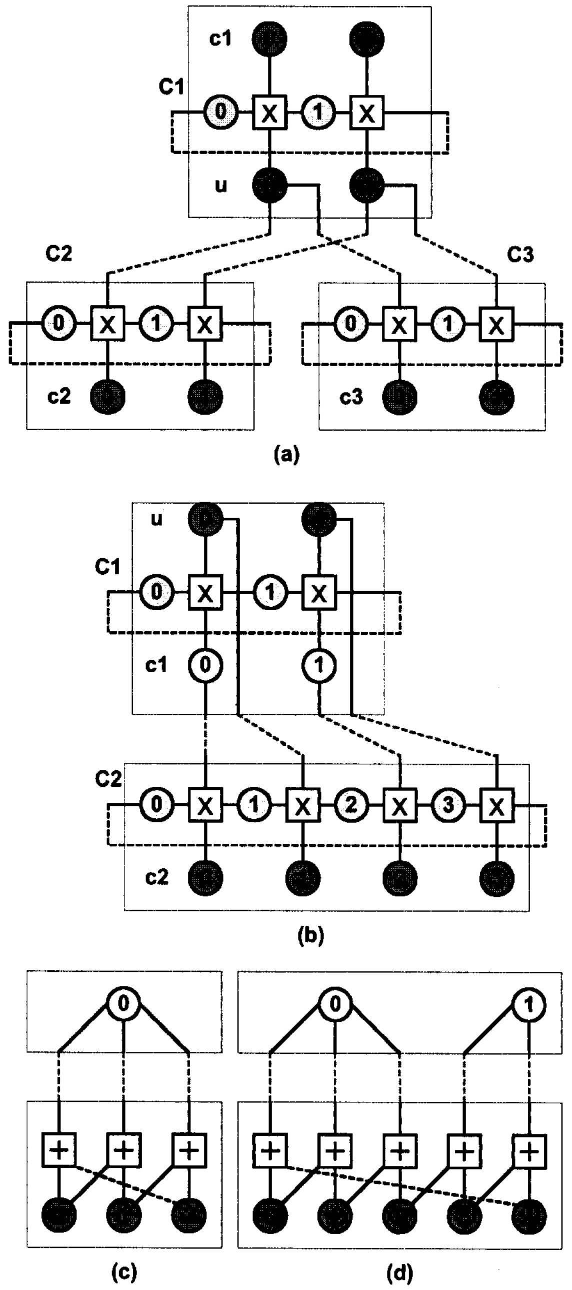

Figure 12. Projected graphs (a) PCCC with 3 constituent codes (b) a simple SCCC (d) RA code (q=3) (d) IRA code (q=2,3)

The second approach is to design the code by designing its projected graph. In this method we should design some component graphs and the connections between them in order to have good performance. In other words, the partitions are designed first and then put together to create constituent codes.

The most important example of this approach is LDPC codes with projected graphs. There are two component graphs: one contains only single parity check codes (variable degrees) and the other has only repetition codes (variable degrees). One example of such a code is shown in Figure 13.

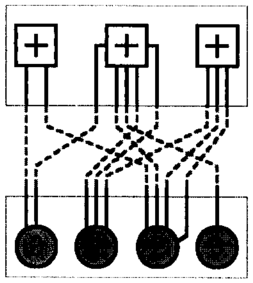

Figure 13. A LDPC with projected graph

There are some noticeable facts about this projected graph. All variable nodes are in one component graph, which means that all the observations is stored and processed in one kind of window processor. Variable and check node can have different degrees. Therefore this structure is capable of implementing regular and irregular LDPC codes. The degree distribution of variable and check nodes is known from the projected graph. There are no local messages and no border messages are passed between adjacent sub-graphs. Therefore, the projected graph can be represented graphically as a simple tanner graph. The graphical representation for above example is shown in Figure 14. The number of interleavers needed for this code is equal to the number of edges of the projected graph.

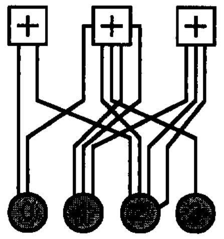

Figure 14. Simple graphical representation of a LDPC projected graph

The only disadvantage of this method for code design is that it does not provide an efficient encoder. Sometimes simple encoding is not possible for codes designed this way.

VIII. General Hardware Architecture

In this section we present general hardware architecture for implementation of parallel turbo-like decoders. Without any loss of generality we focus on turbo-like codes with two constituent codes. This can be easily extended to codes with several constituent codes, see appendix A. The general hardware architecture is shown in Figure 15. EXT_n denotes the external memory for nth window processor.

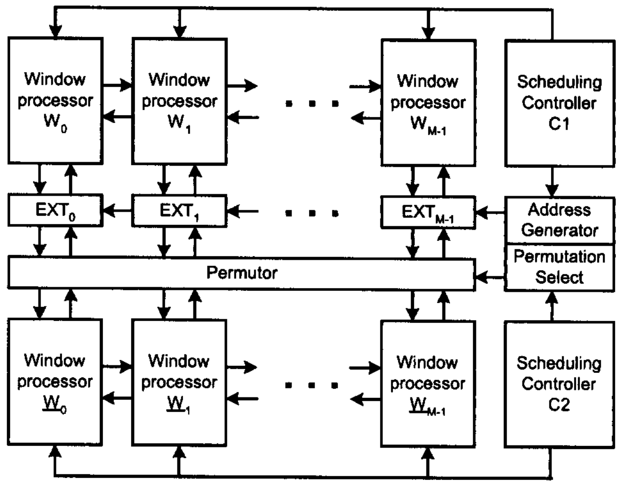

Figure 15. Parallel decoder hardware architecture

Since the processors are identical and are running in parallel, the scheduling is the same for all of them. Therefore, there is only one scheduling controller needed for each constituent codes. The scheduling controller determines which message vector is accessed and what permutation is used. The permuter is a memory-less block that permutes the message vector on the fly. Since the message vectors are permuted differently, the permuter should be programmable. If M, the number of window processors, is large, the permuter can be the bottleneck of the hardware design.

The architecture of one window processor is depicted in Figure 16.

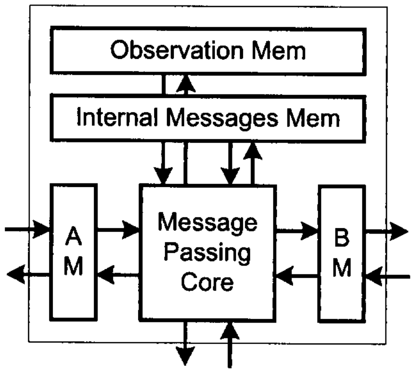

Figure 16. Window processor hardware architecture

AM and BM denote the registers which contains border messages. The observation memory is loaded at the beginning of the decoding and remains intact until end. This memory is not necessary for all window processors.

IX. Conclusion

We have proposed an efficient architecture for parallel processing of turbo decoding. Non-overlapping windows were used that reduces the processing load. Interleavers are designed to have collision free memory access. The tradeoff between speed gain and efficiency was studied. Simulation results showed that the more the speed gain is, the lower is the efficiency. In other words, serial processing is more efficient that parallel processing. However, with proper window-size this structure not only can achieve some orders of magnitude in speed gain, but also maintains the efficiency in processing.

The memory access problem was addressed by designing a special interleaver. Simulation results show that the performance of code with the proposed interleaver structure is as good as other interleavers.

The proposed algorithm was explained based on message passing algorithm and the graphical interpretation of the code structure was presented. This led to introduction of a new class of turbo-like codes that can be decoded very fast, which are the codes with projected graph. This classification provides an alternative method to design turbo-like codes for high-speed decoding.

At the end, a general architecture for decoding this class of codes was also presented. The regularity and simplicity of the proposed architecture make it the architecture of choice for VLSI implementation of high-speed turbo-like codes.

Acknowledgment

The authors would like to thank Dr. F. Lorenzelli for his helpful comments.

Appendix A

Turbo-like codes consist of constituent codes connected with random or psuedo-random interleavers. The iterative decoding is performed by passing messages between these constituent codes.

In most cases there are only two constituent codes. However, it can be shown that we can almost always rearrange the code such that a turbo-like code with only two constituent codes is obtained.

The main idea is to align the interleavers in one line and divide the constituent codes into two sets, which comprise the new component codes. This is explained with some examples in Figure 17.

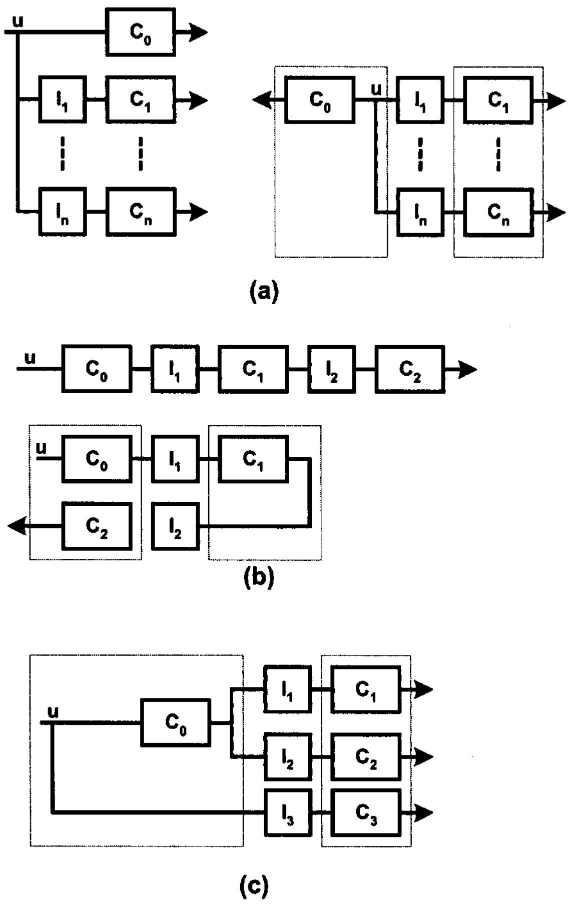

Figure 17. (a) Parallel Concatenated codes (b) Serial Concatenated codes (c) Hybrid concatenated codes.

References

[1] F.R. Kschischang and B.J. Frey, "Iterative decoding of compound codes by probability propagation in graphical models," IEEE JSAC. pp.219-230, Vol.16, No2, Feb. 98.

[2] B.J. Frey, F.R. Kschischang, and P.G. Gulak, "Concurrent turbo-decoding," Proc. of IEEE International Symp. on Info. Theory. p.431. July 97.

[3] J. Hsu and C.H. Wang, "A parallel decoding scheme for turbo codes," ISCAS'98, vol.4, June 98, pp. 445-448.

[4] C. Berrou, A. Glavieux, and P. Thitimasjshima, "Near Shannon limit error correcting coding and decoding: Turbo codes (1)," in Proc. IEEE Int. Conf. Comm., Geneva, Switzerland, May 1993, pp. 1064-1070.

[5] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Soft-input Soft-output APP module for iterative decoding of concatenated codes," IEEE Commu. Letters vol. 1, pp.22-24, Jan. 97.

[6] L.R. Bahl, J. Cocke, F. Jelinek,, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," IEEE Trans. Inform. Theory, vol. IT-20, pp.284-287, Mar. 1974.