Parallel VLSI Architecture and Parallel Interleaver Design for Low-Latency MAP Turbo Decoders

Reuven Dobkin, Michael Peleg and Ran Ginosar
VLSI Systems Research Center, Electrical Engineering Department
Technion—Israel Institute of Technology
Haifa 32000, Israel
[ran@ee.technion.ac.il]

Abstract - Standard VLSI implementation of turbo decoding requires substantial memory and incurs a long latency, which cannot be tolerated in some applications. A novel parallel VLSI architecture for low-latency turbo decoding is described, comprising multiple SISO elements, operating jointly on one turbo coded block, and a new parallel interleaver. The design algorithm for the parallel interleaver is presented, enhancing the error correction performance of the parallel architecture. Latency is reduced up to twenty times and throughput for large blocks is increased up to five-fold relative to sequential decoders, using the same silicon area, and achieving very high coding gain. The parallel architecture scales favorably — latency and throughput improvement with growing block size and chip area.

Index Terms: maximum a posteriori (MAP) algorithm, turbo codes, parallel architecture, VLSI architecture, decoders, interleaver.

1. Introduction

Turbo-codes with performance near the Shannon capacity limit have received considerable attention since their introduction in 1993 [1][2]. Optimal implementation approaches of turbo codes are still of high interest, particularly since turbo codes have become a standard for 3G.

VLSI *sequential architectures* of turbo decoders consist of *M* Soft-Input Soft-Output (SISO) decoders, either connected in a pipeline, or independently processing their own encoded blocks [3][4][5]. Both architectures process *M* turbo blocks simultaneously and are equivalent in terms of coding gain, throughput, latency and complexity.

For the decoding of large block sizes, sequential architectures require large amount of memory per SISO for *M* turbo blocks storage. Hence, enhancing throughput by duplicating SISOs is area inefficient. In addition, latency is high due to iterative decoding, making the sequential architecture unsuitable for latency-sensitive applications such as mobile communications, interactive video and telemedicine.

One way to lower latency is to reduce the number of required decoding iterations, but that may degrade the coding gain. An interesting tree-structured SISO approach [6] significantly reduces the latency, at the cost of an increased area requirement. Parallel decoding schemes [7][8] perform the SISO sliding window algorithm using a number of sub-block SISOs in parallel, each processing one of the sliding windows. Those

schemes trade off the number of sub-blocks for error correction performance, and are reported as having increased hardware complexity relative to sequential architectures. The designs presented in [9] and [10], and the architectures presented in [11] and [12], process sub-blocks in the similar way to ours [13], except for the definitions of the boundary metric values for the beginning and the end of the block, which, as we have shown, can be improved by use of tailbiting termination. In addition, some of them report increase in computational load. Interleaving approaches for parallel decoders were recently presented in [14], [15] and [13].

This paper presents a complete analysis of new parallel VLSI architecture, first presented by us in [13], which, unlike [7] and [8], and similarly to [9][10][11], employs the sliding window approach *inside* each sub-block that is decoded in parallel. This new approach allows choosing the number of SISO decoders independently of desired sliding window and block size. The architecture significantly reduces both latency (up to twenty-fold) and hardware complexity, and improves decoder throughput (up to five-fold) relative to sequential decoder using the same chip area. No significant coding gain degradation was observed. The VLSI architecture was analyzed using FPGA and ASIC design tools. For ASIC the synthesis was performed using 0.35μ Synopsys 2000.05, Passport Libraries tools. As for FPGA estimations, the synthesis was performed using Synplicity tool of Synplify, and Xilinx tools were used for P&R and floorplanning. The paper presents a new algorithm for Parallel Interleaver (PI) design comprising spread optimization and the elimination of low-weight error patterns. We discuss the architecture, implementation and performance of the PI for different levels of parallelism. Performance of parallel and sequential architectures is compared.

After a brief review of the decoding APP algorithm in Section 2, the sequential decoder architecture is discussed in Section 3. The novel parallel decoding architecture is presented in Section 4. Section 5 presents the Parallel Interleaver architecture and its design algorithm. In Section 6 the parallel architecture is compared with the sequential one in terms of coding gain, throughput, latency.

2. Turbo Coding – Theory of Operation

2.1. Encoder

A turbo encoder consists of convolutional encoders connected either in parallel or in series. The parallel scheme [2], shown in Figure 1, consists of an interleaver and two parallel convolutional encoders (CE1, CE2) producing redundant bits (c1, c2). One encoder receives the original information bits (u) while the other receives them interleaved. Interleaving is a crucial component of turbo coding, influencing the performance [2][16].

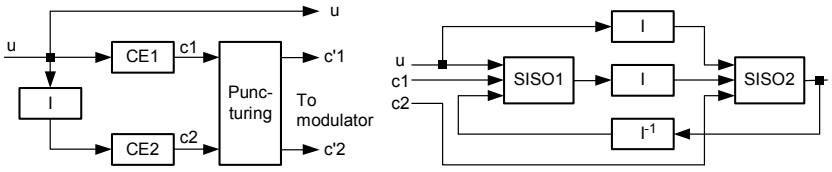


Figure 1: Turbo encoder and decoder, I denotes interleaver

For every new input block, the encoder starts in a known trellis state. In order to finish in a known trellis state, traditionally some extra input termination bits are generated. There are several configurations of termination [16] for turbo coding. However, such terminations result in changes in the block size and in some cases at least one of the added terminations is not interleaved. The *tailbiting* termination technique [17] for recursive convolutional codes keeps the block size unchanged. The technique finds a data-dependent initial state such that the initial and the final states of the encoder are identical. The parallel decoder architecture analyzed in this paper employs the tailbiting termination technique, but it is also applicable to the other termination techniques as well.

2.2. Decoder

The decoding is performed using the iterative decoding scheme (Figure 1): information from one SISO is processed by the other SISO until the desired degree of convergence is achieved. Each SISO produces an increasingly better correction term, referred to as *extrinsic information*, which is appropriately (de)interleaved and used as *a*-priory information by the next SISO [2]. According to the original BCJR algorithm and using notations from [3], the probability values obtained as SISO output are:

$$P_k(u;\,\mathrm{O}) \;=\; \mathrm{H_u} \cdot \sum_{e:\,u(e)=u} A_{k-1}[s^S(e)] \cdot P_k[c(e);\,\mathrm{I}] \cdot B_k[s^E(e)] \qquad (1)$$

where

1) $P(c;I)$ is an estimation of probability of the encoder output (c) symbols (similarly $P(u;I)$ used below for the input (u) symbols).
2) $P(u;O)$ is new refined value of the probability $P(u;I)$.
3) The index k indicates the time step and runs over the entire transmission length.
4) The symbol s represents a code state (state in trellis).
5) e is a generic trellis edge, while $c(e)$ and $u(e)$ are the output and input coder symbols, respectively, associated with the edge e.
6) $s^S(e)$ and $s^E(e)$ indicate the starting and ending states for the generic trellis edge e.
7) H_u is a normalization constant.
8) A_{k-1} and B_k are probability values accumulated in the forward and backward directions along the trellis, according to the following updating relations:

$$\begin{aligned} A_k(s) &= \sum_{e:\,\mathrm{s^E}(e)=s} A_{k-1}[s^S(e)] \cdot P_k[u(e);\,I] \cdot P_k[c(e);\,I] \\ B_k(s) &= \sum_{e:\,\mathrm{s^S}(e)=s} B_{k+1}[s^E(e)] \cdot P_{k+1}[u(e);\,I] \cdot P_{k+1}[c(e);\,I] \end{aligned} \qquad (2)$$

In practice, expensive multiplications are avoided by working in the log domain and using the E-function operator [18][19][20]. Introducing the following definitions,

$$\begin{aligned} \pi_k(c;\mathrm{I}) &\equiv \log[\mathrm{P_k(c;I)}] \\ \pi_k(u;\mathrm{I}) &\equiv \log[\mathrm{P_k(u;I)}] \\ \pi_k(u;\mathrm{O}) &\equiv \log[\mathrm{P_k(u;O)}] \\ \alpha_\mathrm{k}(s) &\equiv \log[A_k(s)] \\ \beta_\mathrm{k}(s) &\equiv \log[B_k(s)] \end{aligned}$$

the previously presented equations of forward and backward metric calculations (1) take the form of:

$$\alpha_k(s) = \underset{e:\, s^E(e)=s}{E}\left\{\alpha_{k-1}[s^S(e)] + \pi_k[u(e);\, \mathrm{I}] + \pi_k[c(e);\, \mathrm{I}]\right\}$$
$$\beta_k(s) = \underset{e:\, s^S(e)=s}{E}\left\{\beta_{k+1}[s^E(e)] + \pi_{k+1}[u(e);\, \mathrm{I}] + \pi_{k+1}[c(e);\, \mathrm{I}]\right\} \quad (3)$$

Then, the output metric of the SISO is calculated as follows:

$$\pi_k(u;\, \mathrm{O}) = \underset{e:\, u(e)=u}{E}\left\{\alpha_{k-1}[s^S(e)] + \pi_k[c(e);\, I] + \beta_k[s^E(e)]\right\} \quad (4)$$

The traditional decoding (Figure 2) is performed by computing first the β metrics for the entire block (going backwards) and storing them. Afterwards, the α values and output metrics $\pi_k(u;\mathrm{O})$ are computed (going forward). The graphical representation was adopted from [21], using the notations in Table 1.

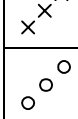
Graphical Notations	
	Input to SISO of the intrinsic (channel) information and the extrinsic information from the previous decoding stage.
	Dummy α state metrics calculation (no storage).
	Dummy β state metrics calculation (no storage).
	Valid β state metrics calculation and storage.
	Valid α metrics and SISO output metrics $\pi_k(u;\, \mathrm{O})$ calculation.

Table 1: Graphical Notations

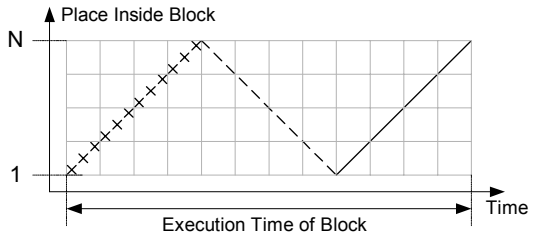


Figure 2: Standard SISO Algorithm

Latency and memory size are significantly reduced when the *sliding window* approach is used [3][22][23]. The backward (and/or forward) metrics are initialized at an intermediate point instead of the end of the block (or at the beginning, for forward metrics). The degradation due to this optimization is negligible when an appropriate intermediate point (sufficient window size) is used [3][22].

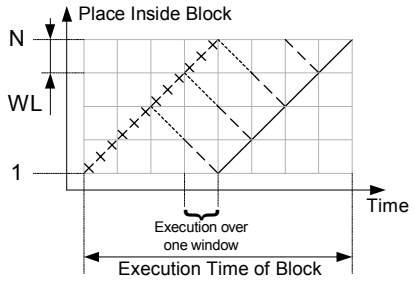


Figure 3: Sliding Window SISO Algorithm

Decoding with the sliding window (Figure 3) is performed as follows. The block is divided into windows of size WL. For each window, initial values of α and β are calculated. The initial α values are the last values of α of the previous window, and the initial β values are calculated by dummy β metrics calculation over the next window (initial values of β for the dummy β calculation are arbitrary). Dummy and valid β metrics are computed using separate hardware [3]. Note that the initial values of α could in principle be calculated by dummy α metrics calculation over the previous window.

When tailbiting termination is employed, the last WL bits of the block (*tail window*) are sent to the SISO prior to the entire block send. The SISO performs dummy α calculation over this window in order to get initial α values for the first window of the block. The initial β values for the last window are calculated and stored during the valid β state metrics calculation over the first window. Note that the cost of using tailbiting is additional latency of WL cycles per decoding iteration. The decoding with sliding window and tailbiting is shown in Figure 4.

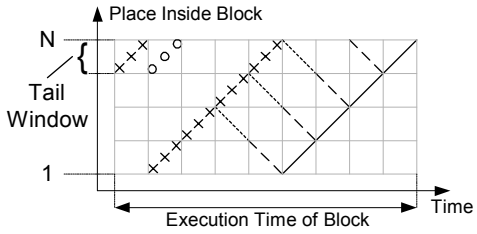


Figure 4: Sliding Window with Tailbiting SISO Algorithm

2.3. Interleaver

2.3.1. Interleaver Parameters

The interleaver size and structure considerably affect the turbo code error performance. The purpose of the interleaver in turbo codes is to ensure that the information patterns, which cause low-weight words for the first encoder, are not interleaved to similar patterns for the second encoder, thus improving the weight spectrum of the code.

The interleaver spread is one of the main interleaver parameters. Spread optimization is desirable for both fast convergence and good distance properties of the code. Large

distance leads to lowering the point at which the BER curve flattens ("error floor") and for increasing the slope of the BER.

When J is the one-sided span of the output indices, and $I(i)$, $I(j)$ are the locations of the interleaver outputs i and j at the input of the interleaver, the spread definition, usually defined for the S-random interleaver [24] is:

$$S''(J,i,j) = \left|I(i) - I(j)\right|, \quad for \ \left|i - j\right| < J$$

The minimal spread associated with index i is then defined as:

$$S'(J,i) = \min_{j}\left[S''(J,i,j)\right]$$

The minimal spread, $S(J)$ associated with the entire interleaver is:

$$S(J) = \min_{i}\left[S'(J,i)\right] \tag{5}$$

The pair (S, J) is called also the *spreading factors* of an interleaver [25]. Another alternative description of the spreading factors is described in terms of the *displacement vector*:

$$(\Delta_X, \Delta_Y) = \left(\, j - i,\ I(j) - I(i) \,\right), \quad i < j \tag{6}$$

A new and more effective definition of the interleaver spread was introduced in [24] and was adopted in this work. The spread definition associated with two output indices i and j is defined as:

$$S''_{HS}(i,j) = \left|I(i) - I(j)\right| + \left|i - j\right|$$

The minimal spread associated with index i is then defined as:

$$S'_{HS}(i) = \min_{j}\left[S''_{HS}(i,j)\right]$$

The minimal "High-Spread", $\boldsymbol{S_{HS}}$, associated with the entire interleaver is:

$$S_{HS} = \min_{i}\left[S'(i)\right] \tag{7}$$

When tailbiting termination is used, the distance calculations, like $\left|\boldsymbol{i} - \boldsymbol{j}\right|$ above, are performed considering the tailbiting cyclic trellis feature, thus the tailbiting distance for two indices i and j is defined as follows (N is the interleaver size):

$$D_{Tail}(i,j,N) = \min\left[\ \left|\, i \ - j \,\right|,\ N \ - \left|\, i \ - j \,\right| \ \right] \tag{8}$$

The "randomness" of the interleaver also is of high influence on the performance. Interleavers having regular structure, such as classical block interleavers, perform poorly for turbo codes. The set of displacement vectors of the interleaver can be used to study the "randomness" [25]:

$$D(I) = \{(\Delta_X, \Delta_Y) \in Z^2 \mid \Delta_X = j - i, \Delta_Y = I(j) - I(i),\ 0 \le i < j < N\} \tag{9}$$

The largest set of displacement vectors occurs for *Costas* permutation [25][26]. In this case the number of displacement vectors is $\boldsymbol{N}\cdot(\boldsymbol{N}-1)/2$. The normalized dispersion, γ, is defined then as follows:

$$\gamma = \frac{2 \cdot |D(I)|}{N \cdot (N-1)}, \quad \text{obtaining: } \frac{2}{\mathrm{N}} \le \gamma \le 1 \tag{10}$$

where $|\boldsymbol{D}(\boldsymbol{I})|$ is the size of the set of the displacement vectors *D(I)*.

In this work, that definition of dispersion could not be used, due to the use of tailbiting. Therefore, we consider the ratio of the number of parallel interleaver displacement vectors ($|D(I_{PI})|$, as proposed in this paper) to the dispersion of a random interleaver:

$$\Gamma = \frac{|D(I_{PI})|}{|D(I_{Random})|} \tag{11}$$

2.3.2. Error Patterns

With recursive systematic codes, single 1's yield codewords of semi-infinite weight, and low-weight words appear with patterns of 2, 3 and 4 errors in information bits [27]. While for a single convolutional code a 3-bit error pattern may cause an error event, it rarely happens in turbo codes, thanks to interleaving. The low-weight patterns elimination contributes drastically to the code performance at the error floor region [27]. In this work the proposed algorithm eliminates 2- and 4-bit error patterns. The structure of the error patterns and the corresponding search algorithm are detailed in Section 5.3.

3. Sequential Decoder Architecture

The detailed scheme of an iterative decoder is depicted in Figure 5.

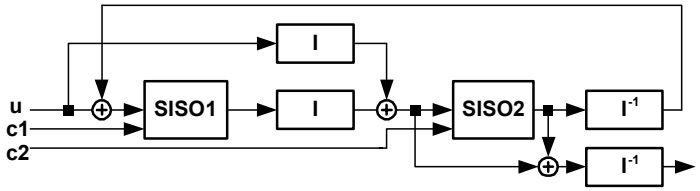


Figure 5: Iterative Decoder Scheme (I denotes interleaver)

An iteration through the decoder can be divided into two stages:

a. "*Interleaving Stage*". The result of the previous iteration plus *u*, and *c1* bits are processed by SISO1 and passed through interleaver I.

b. "*DeInterleaving Stage*". The extrinsic data from Interleaving Stage plus an interleaved version of *u*, and *c2* are processed by SISO2 and de-interleaved.

Both stages perform similar operations in the same order: Add, compute (SISO) and (de)interleave. When CE1 and CE2 are identical, SISO1 is identical to SISO2. In addition, the same memory unit can perform the interleaving and de-interleaving processes while suitable addresses are provided. Therefore, the above two stages can be implemented by the same hardware block, used twice for each iteration.

To decode a block, a *decoding unit* consists of a SISO, interleaver memory, adder, memories for channel data (*u, c1, c2*), interleaving address memory and control logic. When parallel processing of, say, *n* blocks is required to achieve higher data rate, the entire decoding unit is duplicated *n* times. Alternatively, a single interleaving address memory can be shared, using appropriate FIFOs.

The maximal input rate, $F_{in,seq}^{Uncoded}$, for the sequential architecture (with input double buffer) is:

$$F_{in,seq}^{Uncoded} = \frac{NDU}{2 \cdot NI} \cdot F_{\text{int},seq}^{eff} \tag{12}$$

$$\text{where } F_{\text{int},seq}^{eff} = \frac{F_{\text{int}} \cdot N}{N + c \cdot WL} \tag{13}$$

NDU: number of decoding units,
NI: number of iterations,
N: block size,
WL: window length,
c: SISO delay in window lengths (for the algorithm of Figure 4, c=5),
$F_{\text{int},seq}^{eff}$: effective processing rate, and
F_{int}: internal clock rate.

The $\frac{N}{N + c \cdot WL}$ ratio in Eq. (13) is the number of metrics processed per an internal clock cycle, while performing an internal iteration of the decoder. The fraction is less than one due to the SISO delay of $c \cdot WL$ cycles. The maximal possible processing rate F_{int} is degraded by this fraction. The input rate in Eq. (12) reflects *NDU* concurrent independent decoding units and $2 \cdot NI$ internal operations executed sequentially.

For a given silicon area, the throughput of the decoder $F_{in,seq}^{Uncoded}$ depends on the number of decoding units that can be placed on that area. The area efficiency of sequential and parallel architectures is discussed below.

Latency of the sequential architecture is that of the decoding unit:

$$D_{Seq} = \frac{2 \cdot NI \cdot (N + c \cdot WL)}{F_{\text{int}}} \tag{14}$$

The latency consists of the delay of the practical SISO due to prior input of five windows (c=5 in Figure 4) in addition to processing *N* metrics of the block, all multiplied by the number of iterations.

4. Parallel Decoder Architecture

The parallel decoding architecture applies *m* SISOs in parallel to one incoming block. The block is decomposed into *m* sub-blocks. The decomposition of processing to sub-blocks is facilitated by applying the sliding window principle, which allows independent decoding of sub-blocks without degradation in error-correction performance [22]. Dummy α and β metrics are calculated in order to determine the initial values of α and β for each sub-block *i.* An example of block decomposition to *m*=5 sub-blocks is shown in Figure 6.

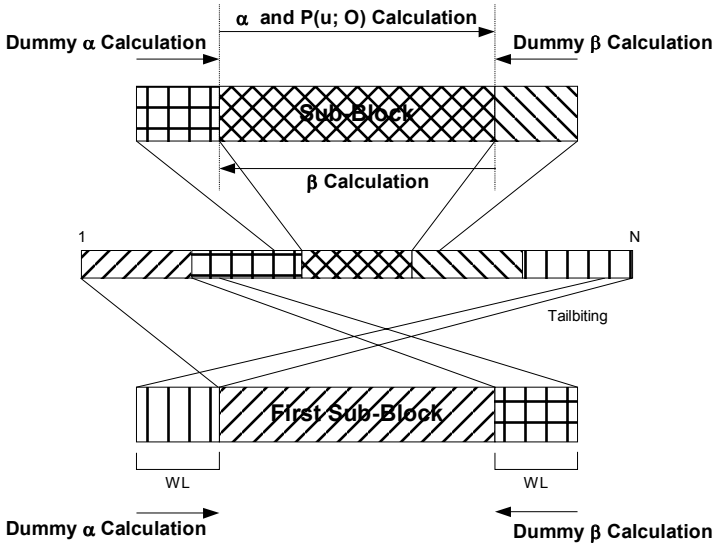


Figure 6: Sub-Block Decomposition Example (WL denotes Window Length)

For each sub-block *i* the initial α metrics are calculated over the "tail" window of sub-block *i*-1, incurring a slight increase of computational load but no increase of latency. For the first sub-block (*i*=1), the tail window is taken from sub-block *m*, thanks to tailbiting. Similarly, initial β values for the last window of sub-block *i* are β values received for the first window in sub-block *i*+1, incurring a slight decrease of computational load, which compensates for the increase, mentioned above, due to α dummy metrics computation. The sub-block is decoded according to the sliding window SISO algorithm (as in Figure 4).

The related works [7] and [8] have same approach of dividing the block into sub-blocks. However, the proposed in that works algorithms do not apply sliding window inside sub-blocks. In addition, proposed here technique for boundary metrics computation is more generic, than the techniques proposed in [10][11][12], and causes no performance degradation or increase of computational load.

Parallel versus sequential decoding are shown in Figure 7. At the beginning the "tail" windows are supplied to the SISOs. Subsequently, the sub-block itself is sent. Parallel decoding significantly reduces the processing latency relative to sequential decoding as evident from Figure 7. The latency in this case becomes equal to the sub-block processing latency.

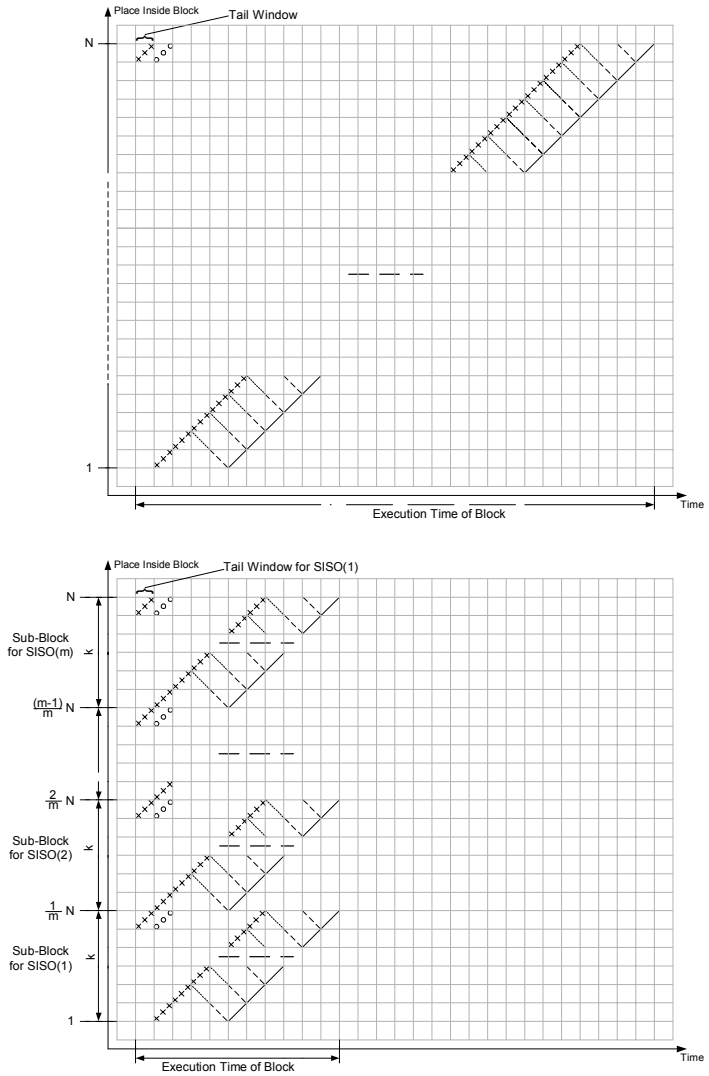


Figure 7: Sequential and Parallel Decoding (with Tailbiting)

Alternatively, sub-block decomposition could be performed in the encoder, encoding each sub-block separately with its own tailbiting termination [28]. Thus, there would be no need to exchange dummy metrics between the parallel data flows in the decoder. Such a scheme incurs the same decoder complexity and achieves the same throughput and latency as the previous approach.

The parallel processing of Figure 7 executes one SISO path, which is only one half of the decoding iteration (referred as an internal iteration). The results are interleaved, divided again into sub-blocks and then sent for the next parallel processing. Iterative parallel processing is executed by the architecture shown in Figure 8, as follows.

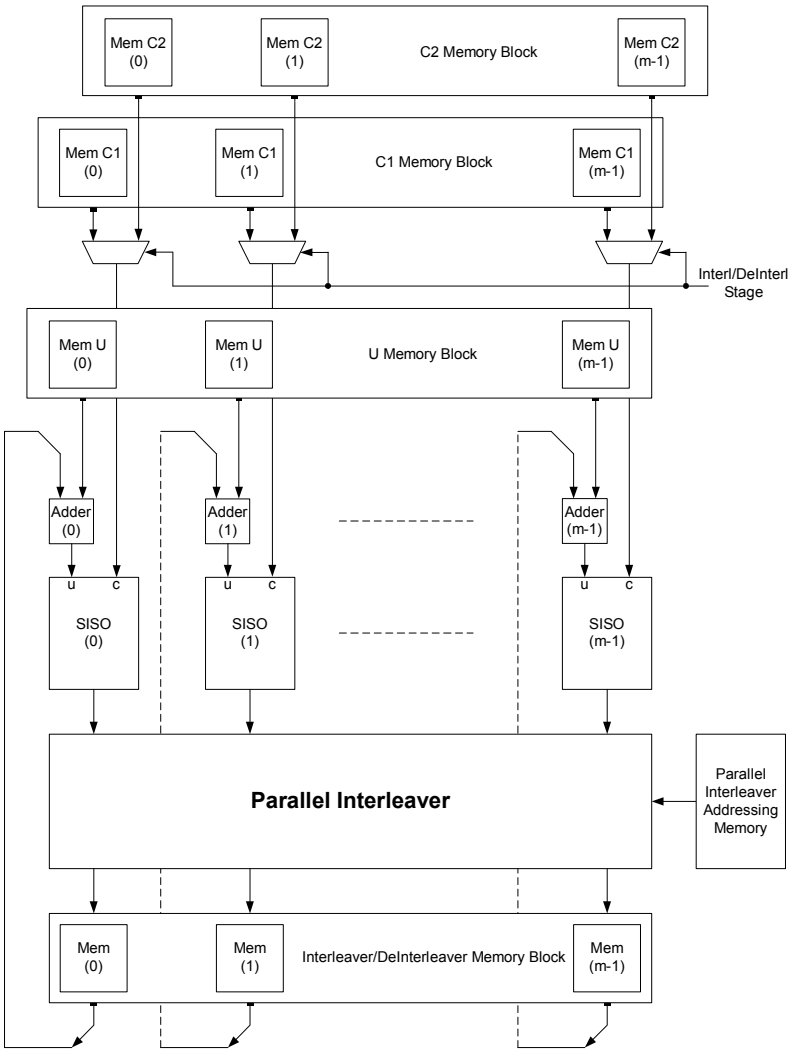


Figure 8: Parallel Decoder Architecture

The incoming block is divided into m sub-blocks and each sub-block is sent to a separate SISO. The operations are the same as in the sequential architecture: Add, compute (SISO) and de/interleave. After SISO processing, the extrinsic metrics are sent (in sets of m metrics) to the *Parallel Interleaver* where they are permuted and stored in the interleaver memory (Interleaver / Deinterleaver Memory Block in Figure 8). The interleaver memory and the channel data memories (U, C1, and C2 memory blocks) consists each of an array of m memories of depth N/m. The Parallel Interleaver (PI) performs interleaving according to the addressing supplied by its addressing memory. PI architecture is discussed in Section 5.

The maximal input rate $F_{in,par}^{Uncoded}$ for the parallel architecture is:

$$F_{in,par}^{Uncoded} = \frac{m}{2 \cdot NI} \cdot F_{\text{int},par}^{eff} \tag{15}$$

$$\text{where } F_{\text{int},par}^{eff} = \frac{F_{\text{int}} \cdot \frac{N}{m}}{\frac{N}{m} + c \cdot WL} \tag{16}$$

and m is the number of SISOs, defining the parallelism level of the decoder.
The expression for sequential effective processing rate $F_{\text{int},seq}^{eff}$ (see Eq. (13)) is changed here to a sub-block effective processing rate $F_{\text{int},par}^{eff}$, using sub-block length N/m instead of total block length N (Eq. (16)). Similar to Eq. (12), the decoder processing rate reflects performing $2 \cdot NI$ internal operations and processing m sub-blocks in parallel.

The latency also depends on m: Each SISO now operates on only N/m metrics. The latency consists of the delay of the practical SISO due to prior input of c windows in addition to processing N/m metrics of the sub-block, all multiplied by the number of iterations:

$$D_{Par} = \frac{2 \cdot NI \cdot (\frac{N}{m} + c \cdot WL)}{F_{\text{int}}} \tag{17}$$

5. Parallel Interleaver

5.1. Architecture

The parallel interleaver (PI) plays a key role in the performance of the entire parallel decoder. Its task is to permute in parallel at least m metrics coming simultaneously from m SISOs and send the permuted metrics to an array of memories for storage and further permuting (Mem(1)-Mem(m) in Figure 9).

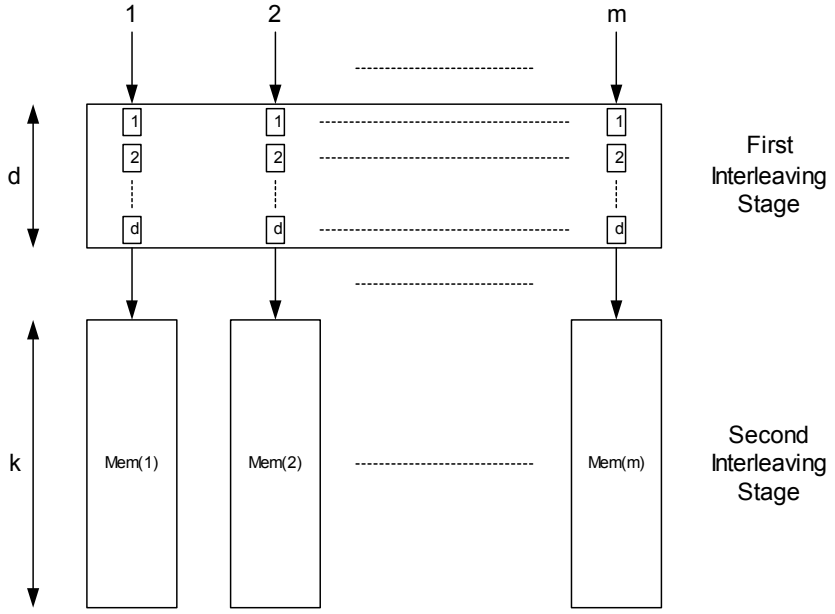


Figure 9: The Parallel Interleaver

The interleaving consists of two stages:

a. First Interleaving Stage (FIS).
b. Second Interleaving Stage (SIS).

The parallel interleaver architecture is parameterized by *m,* the number of inputs/outputs of the interleaver, and *d*, the FIS delay (Figure 9). The minimal depth of each storage memory in SIS is denoted by *k*, where:

$$k = N / m \tag{18}$$

The incoming sets of metrics are first permuted by the First Interleaving Stage, and are subsequently permuted again in the target memories (Mem(1)-Mem(*m*)) of the Second Interleaving Stage.

At each calculation cycle, *m* metrics (from *m* SISOs) enter FIS. At the beginning, FIS accumulates $m \cdot d$ metrics for *d* cycles and then starts ejecting sets of *m* metrics for each calculation cycle. Each output set contains *m* out of the $m \cdot d$ metrics. Each metric is identified by its *SISO source index* and *input cycle number*, as in the example of Table 2. FIS permutations are affected by permuting these indices, turning them into the *memory destination index* and the *output cycle number.*

Input Sets:		SISO Source Index			
		1	2	3	4
Input Cycle	2	m5	m6	m7	m8
	1	m1	m2	m3	m4

Output Sets:		Memory Destination Index			
		1	2	3	4
Output Cycle	2	m5	m7	m2	m4
	1	m1	m6	m8	m3

Table 2: FIS Interleaving Example with m=4, d=2. The input on the left is permuted into the table on the right.

After FIS interleaving, the data arrive in the Second Interleaving Stage (SIS), where intra-sub-block permutation is performed by proper addressing of each of the *m* memories. For the sake of efficient hardware implementation, $N_{\max}$ and *m* should be chosen so that $rem(N,m) = 0$ and *k* is a power of two (see Eq. (18)). All *k!* permutations can be achieved on the metrics within the same SIS memory since there is no restriction on the order of metrics inside the memory.

5.2. Possible Permutations

Due to the structure of PI, the number of possible permutations is limited relative to the turbo interleaver used in the sequential decoder. The standard interleaver of size *N*, implemented as a memory, can perform *N!* permutations. We considered three alternative FIS architectures: *A cross-bar, an infinite permutation network and a finite permutation network* (Figure 10). Wide input and output memories are combined with cross-bar switches. Regardless of which FIS architecture is employed, the output is always directed into the SIS.

In the *crossbar* architecture, a single set of *m* metrics is permuted by a $m \times m$ crossbar switch, facilitating *m!* possible permutations. Note that two metrics which arrive in the same input cycle can *never* end up in the same target SIS memory (Mem(i) in Figure 9).

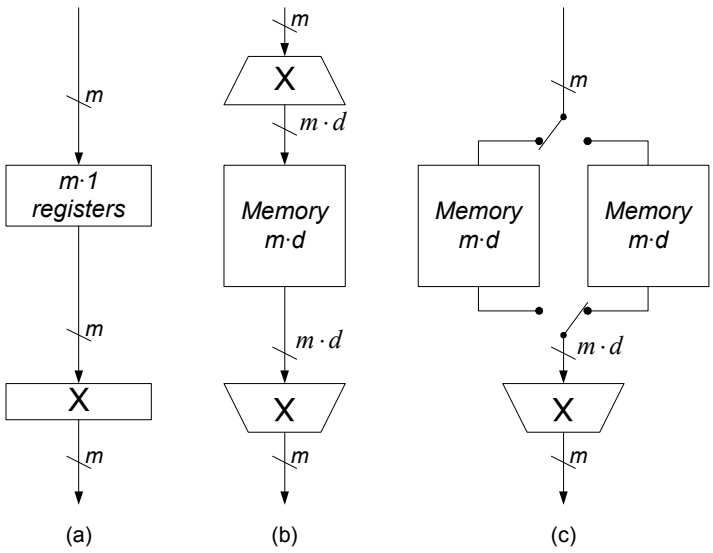


Figure 10: Alternative FIS Architectures: (a) cross-bar (b) infinite permutation network (c) finite permutation network. ***X*** *represents a cross-bar switch*

In the (most powerful) *infinite permutation network* architecture, an $m \times (m \cdot d)$ crossbar spreads the incoming m metrics set into m free spaces in the $m \cdot d$ memory. A different set of m metrics is concurrently extracted from the memory and permuted by the second crossbar. All metrics stored in the memory are considered as one big set, out of which the next set of m metrics can be selected without any constraints. The total effect in the FIS thus consists of three permuting steps (in the two crossbar switches and by memory addressing). Note that any single metric may enter memory early on and stay there for a long time, hence the name of this architecture. Note further that, unlike the *crossbar* architecture, the entire set of d metrics (namely, metrics arriving simultaneously at the FIS) can be channeled into the same SIS memory.

The *finite permutation network* architecture is a simpler version, employing a double-buffered memory in lieu of the first crossbar switch. d metric sets are stored in one memory (during d cycles) and are permuted and extracted in full during d subsequent cycles, when the other memory is being filled. Thus, any single metric may be delayed by at most $d-1$ cycles, hence the name of that architecture. The full content of a buffer ($m \cdot d$ metrics) is termed a *delay packet* in the following. The allowed delay, d, impacts the total number of possible permutations. For an unlimited d value, FIS resolves all blocking and provides all $N!$ permutations.

A parallel interleaver may perform any of $N!$ possible permutations, when FIFOs of an appropriate length are used. A form of parallel interleaver, which optimizes the FIFOs sizes, was recently proposed in [14]. Multiple multi-input FIFO solution becomes very expansive with block size and parallelism level growth.

The *crossbar* architecture can be considered a special case of the *finite permutation network*, whereas the latter is a special case of the infinite one. Their implementation is progressively more complex; we have opted for the medium complexity *finite permutation network,* and we analyze its area requirements in Section 5.4 below. We now consider the impact of the level of parallelism m and the FIS delay d on *NPerm,* the number of possible PI permutations.

Let's assume that $m \cdot d$ divides N ($N \mid m \cdot d = 0$). The depth of the SIS memory (and the number of output cycles of the SISOs array) is $k=N/m$. In other words, k sets of m metrics ($N/(m \cdot d)$ packets) enter PI during one internal decoding iteration.

The interleaving process is depicted in Figure 11. For each delay packet the FIS can perform $(m \cdot d)!$ permutations. However, this number includes permutations performed by SIS for the metrics entering the same memory, and thus for each set of d metrics sent by FIS to the same memory, $d!$ permutations should be eliminated. This results in $\frac{(m \cdot d)!}{(d!)^m}$ permutations for a single delay packet.

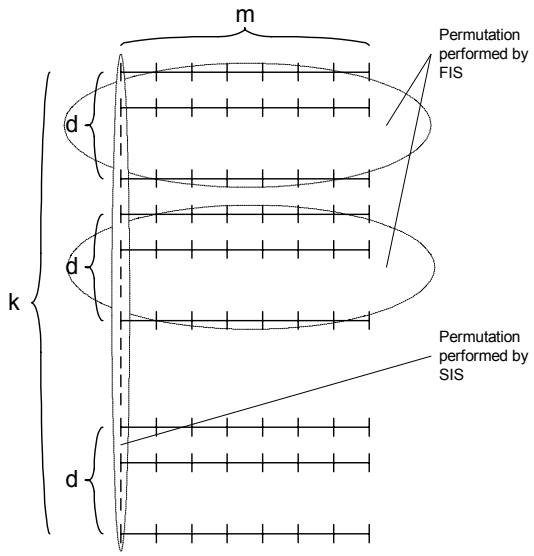


Figure 11: PI (Delay Packet Approach) Interleaving

Each SIS memory can perform $k!$ permutations. The total number of possible permutations for the entire PI is:

$$NPerm = \left[\frac{(m \cdot d)!}{(d!)^m} \right]^{N/(m \cdot d)} \cdot (k!)^m \tag{19}$$

FIS can perform $\frac{(m \cdot d)!}{(d!)^m}$ permutations on one delay packet, there are $N/(d \cdot m)$ delay packets, and the entire SIS can perform $(\boldsymbol{k}!)^{\boldsymbol{m}}$ permutations. For example,

For d=1: $NPerm = (m!)^{N/m} \cdot (k!)^m = (m!)^k \cdot (k!)^m$ (*crossbar* architecture).

For m=1: $NPerm = \left[\frac{d!}{d!} \right]^{N/d} \cdot (k!) = k! = (N/m)! = N!$ (one SISO, sequential architecture).

Working in the log-domain, Eq. (19) takes the form of:

$$\ln(NPerm) = \frac{k}{d} \cdot \{\ln[(m \cdot d)!] - m \cdot \ln[d!]\} + m \cdot \ln[k!] \tag{20}$$

As can be seen from Eq. (19) the number of possible permutations depends on three parameters: N, m and d. The following charts demonstrate those dependencies.

Stirling's approximation was used in the computation. In both charts (Figure 12 and Figure 13), ln(*NPerm*) is normalized by the logarithm of the maximal number of permutations, *N!*.

Figure 12 represents the permutations attainable by *finite permutation networks* for different block sizes (*N)*. In this example *d*=1 (a *crossbar* architecture). The decrease shown on the log-ratio scale actually reflects a decrease by many orders of magnitude in the permuting power relative to the maximum number of permutations. In general, as the parallelism level grows, the degradation levels off thanks to the growing size of the delay packet (*m·d*).

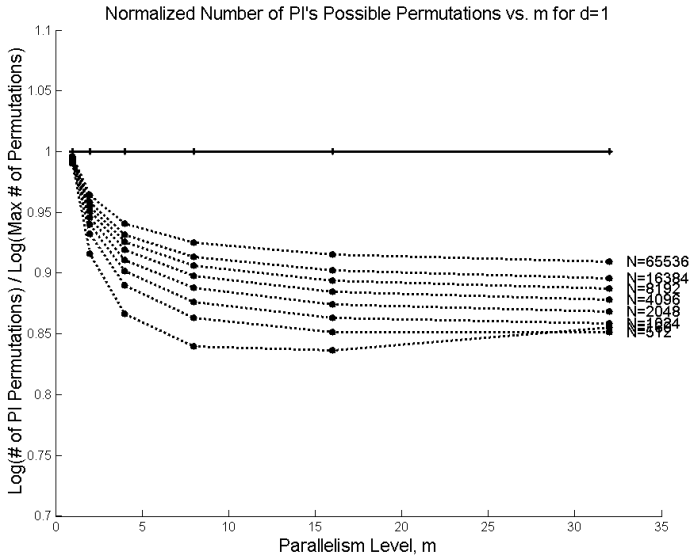


Figure 12: Normalized Number of Permutations for Different N vs. m

Increasing *d* also expands the delay packet size, resulting in a growing number of possible permutations. For example, Figure 13 shows the results for *N=4K* and for different delays, *d*. Thus *d*>1 compensates for most of the decrease resulting from partitioning.

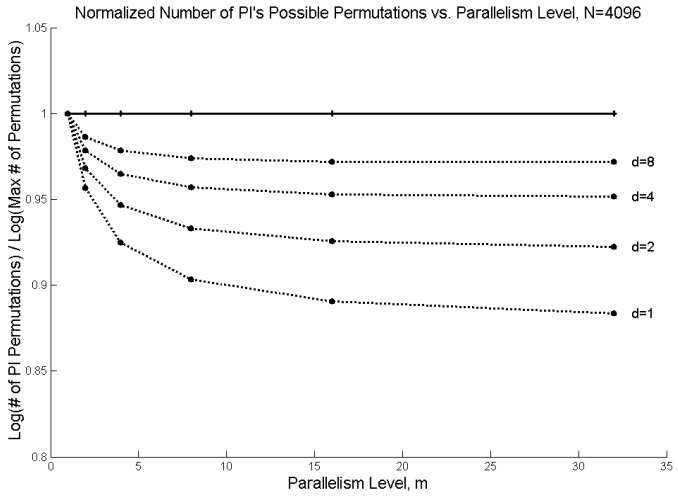


Figure 13: Normalized Number of Permutations for Different Delays vs. m

5.3. Parallel Interleaver Design

5.3.1. Interleaver Design Algorithm

Given a PI with certain *N, m, d* parameters and *finite permutation network* FIS, the high performance PI design algorithm (Figure 14 and Figure 15) generates the required permutations according to the high-spread criteria (Eq. (7)), while eliminating the 2- and 4-bit error patterns (Section 2.3.2) and considering the tailbiting restrictions of Eq. (8).

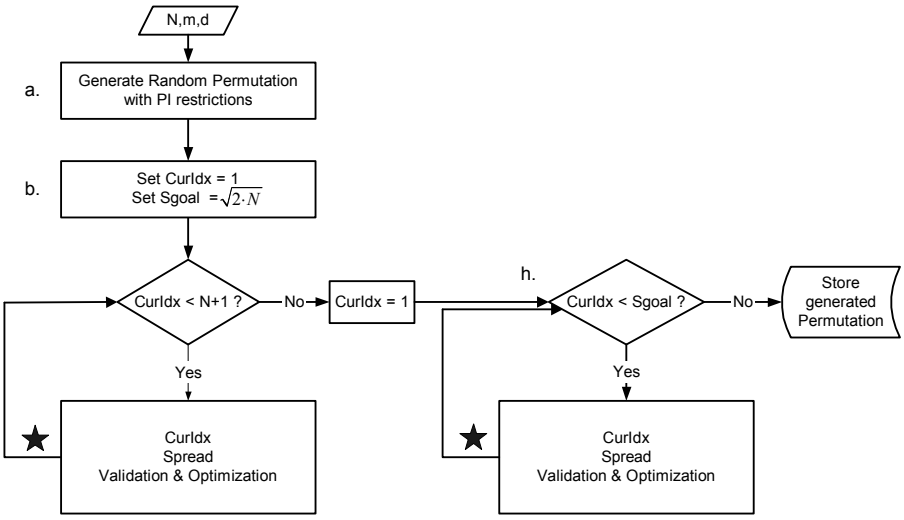


Figure 14: Parallel Interleaver Design Algorithm - Main Flow

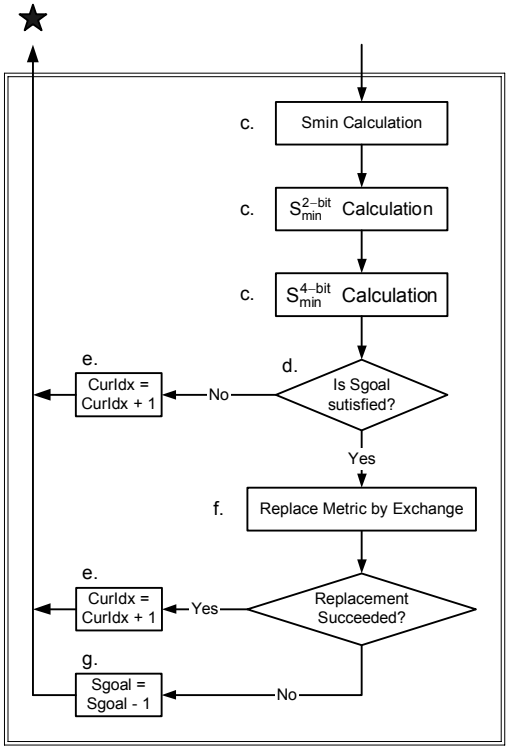


Figure 15: PI Design Algorithm – CurIdx Spread Validation & Optimization

The algorithm comprises the following stages:

a. A random permutation P[1..N] according to the *N*, *m* and *d* parameters is generated by passing an ordered sequence through the PI, using random addresses at the First and Second Interleaving Stages (FIS and SIS). Thus, the permutation is guaranteed realizable.

b. The current spread value, Sgoal, is initialized to the maximal possible value. While according to [24] $S_{goal}^{Max} = \sqrt{2 \cdot N}$, in practice that bound is unreachable and a lower initial value results in faster convergence of the algorithm.

c. For each permutation index CurIdx, the algorithm performs minimal spread (Smin) calculation according to Eq. (7). The spread is calculated for CurIdx (Smin) and for the two- and four-bit patterns related to the CurIdx ($S_{\min}^{2-bit}$ and $S_{\min}^{4-bit}$, Eq. (22) and Eq. (23) below).

d. In order to satisfy total permutation spread, Sgoal (the minimal spreads computed at stage c) should satisfy all the following inequalities:

$$\begin{aligned} &\text{i. } S_{\min} \geq S_{goal} \\ &\text{ii. } S_{\min}^{2-bit} \geq 2 \cdot S_{goal} \\ &\text{iii. } S_{\min}^{4-bit} \geq 2 \cdot S_{goal} \end{aligned} \tag{21}$$

e. When Eq. (21) is satisfied, the algorithm accepts CurIdx, and begins treating the next index, CurIdx+1.

f. If Eq. (21) is not satisfied, P[CurIdx] is rejected and replaced as follows:

   i. The algorithm searches for a set of indices, which can be swapped with P[CurIdx]. The suitable indices must belong to the same *delay packet* or to the same *SIS memory* as CurIdx, and satisfy Eq. (21) after the swap.

   ii. If such a set is found, P[CurIdx] is exchanged with a randomly selected index from the set, and CurIdx is incremented. Otherwise, replacement cannot be performed.

g. If the replacement cannot be performed, the Sgoal constraint is reduced.

h. Due to tailbiting, when the algorithm reaches index *N*, the first Sgoal indices of the permutation should be recomputed.

When in stage g, there are two approaches that can be applied. The search could be restarted (CurIdx could be reset to 1 in stage g), or the search can be continued with the same CurIdx and reduced Sgoal. We have found that the latter option yields better results in terms of final spread and execution time. We believe that the resulting interleaver regularity is lower, improving the total dispersion of the permutation.

Convergence of this algorithm was found to be fast. It converges very close to the highest spread value within a few dozens of search iterations. The algorithm was developed based on results presented in [16] [24][27]. A similar approach (for the design of a sequential interleaver) has recently been presented in [29], achieving a

performance very close to the application of our algorithm to the sequential case (m=1).

5.3.2. 2-bit Error Patterns

A 2-bit error pattern is defined by CurIdx and three additional indices (Figure 16) that sustain the following conditions:

a. An index at the input sequence (In) that sustains the tailbiting distance of $n \cdot (L-1)$ forward or backward relative to CurIdx (L is the maximal code generator length).

b. Two output indices which are the interleaving targets of CurIdx and the input index from a, and for which the tailbiting distance between the output indices is $m \cdot (L-1)$.

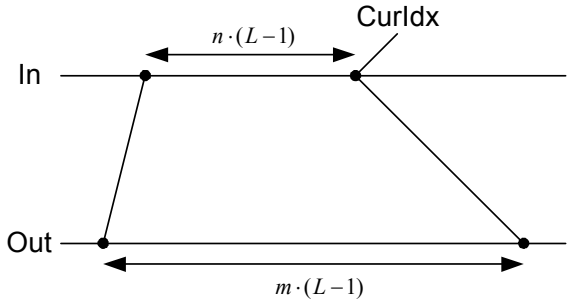


Figure 16: Two-Bit Error Pattern; n, m are integers

According to Eq. (7), 2-bit error pattern spread, $S_{\min}^{2-bit}$, is:

$$S_{\min}^{2-bit} = n \cdot (L-1) + m \cdot (L-1) \tag{22}$$

When $S_{\min}^{2-bit} \geq 2 \cdot S_{goal}$, the *error event* is long enough and the pattern is not eliminated. Otherwise, the CurIdx is exchanged with another index, such that no error patterns occur for either CurIdx or the other index.

5.3.3. 4-bit Error Patterns

For 4-bit error pattern there are two low weight error events at each component code [27], even though each pair exhibits a high spread separately (Figure 17).

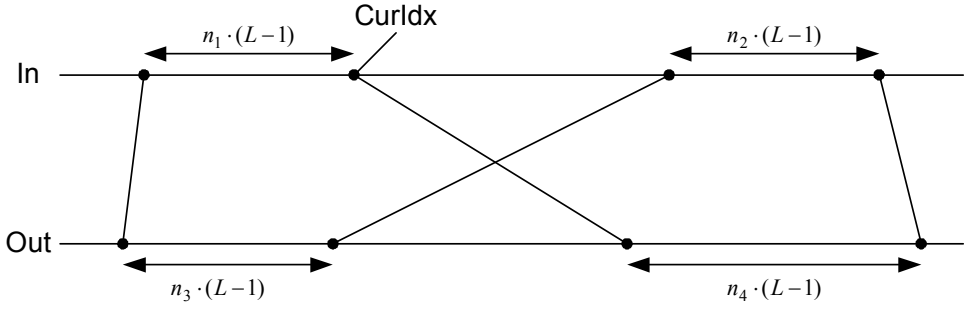


Figure 17: Four-Bit Error Pattern; $\boldsymbol{n}_1, \boldsymbol{n}_2, \boldsymbol{n}_3, \boldsymbol{n}_4$ *are integers*

As in Section 5.3.2, the distances between the indices are computed according to Eq. (8). The 4-bit error pattern spread, $S_{\min}^{4-bit}$, is:

$$S_{\min}^{4-bit} = \sum_{i=1}^{4} n_i \cdot (L-1) \tag{23}$$

When $S_{\min}^{4-bit}$ does not sustain Eq. (21), the pattern is eliminated by exchanging the CurIdx with other index of the permutation, according to the conditions listed in stage f of the PI design algorithm.

5.4. VLSI Implementation

The Parallel Interleaver (PI) consists of two stages: FIS and SIS (Figure 9). SIS is implemented by an array of memories that perform SIS interleaving using addresses generated by the algorithm of Section 5.3. The following FIS implementation is optimized for the Finite Permutation Network architecture (Figure 10).

FIS consists of an array of $m \cdot d$ memory elements (flip-flops) followed by a $(m \cdot d) \times m$ crossbar switch (an interconnection matrix and selection multiplexers). The Finite Permutation Network comprises two memory arrays (Figure 10) in a double-buffer setup; only one array is shown in Figure 18.

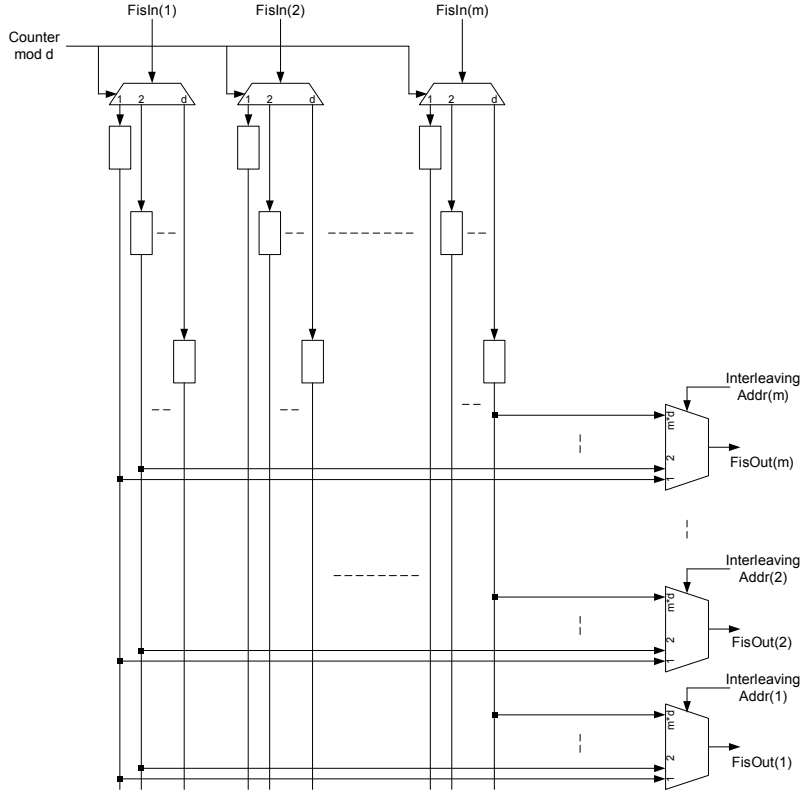


Figure 18: Finite Permutation Network Architecture

The write operation is performed sequentially by rows. For each output at each read cycle, an address is supplied indicating from which of the $m \cdot d$ memory elements the metric is taken. The two buffers are switched each time d sets are read and written.

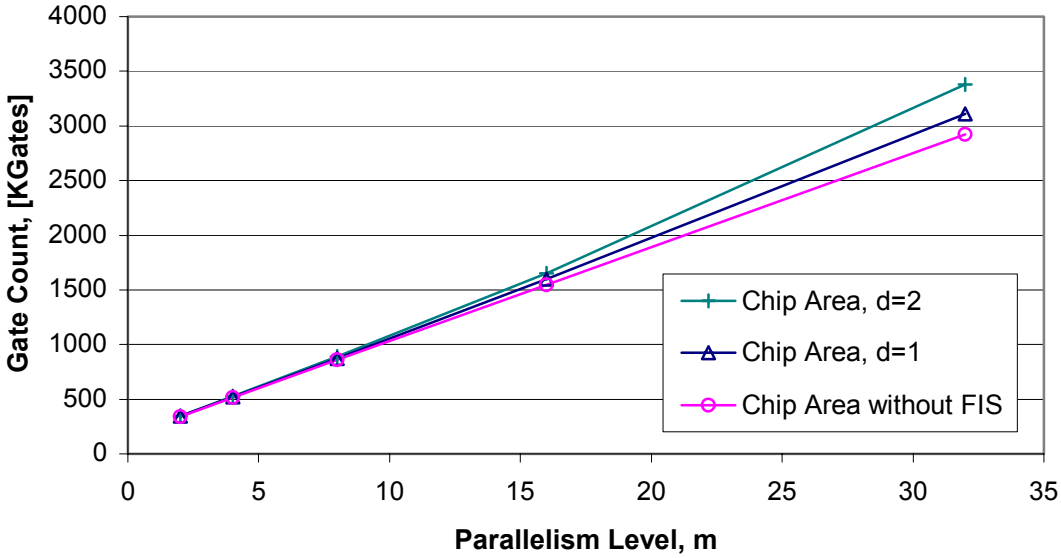


Figure 19: Parallel Turbo Decoder ASIC Chip Area vs. Parallelism Level

ASIC and FPGA FIS implementations were designed and compared for area requirements. An eight-bit data width was selected for inputs, outputs and the memory elements. The total parallel decoder chip area (for parallelism level of m) is approximately m times the area of a single SISO. Figure 19 shows ASIC chip area for d=1,2 and a baseline without a FIS. Evidently, FIS requires an insignificant silicon area on the parallel decoder chip. Note the highly linear growth in chip area with m; below we show also a linear speedup in return for this linear increase in cost.

Figure 20 shows similar results for an FPGA implementation. In addition, it should be noted that at least twice lower total gate counts and FIS gate counts are achieved, when a shorter metric representation (thanks to SISO internal optimizations) is used.

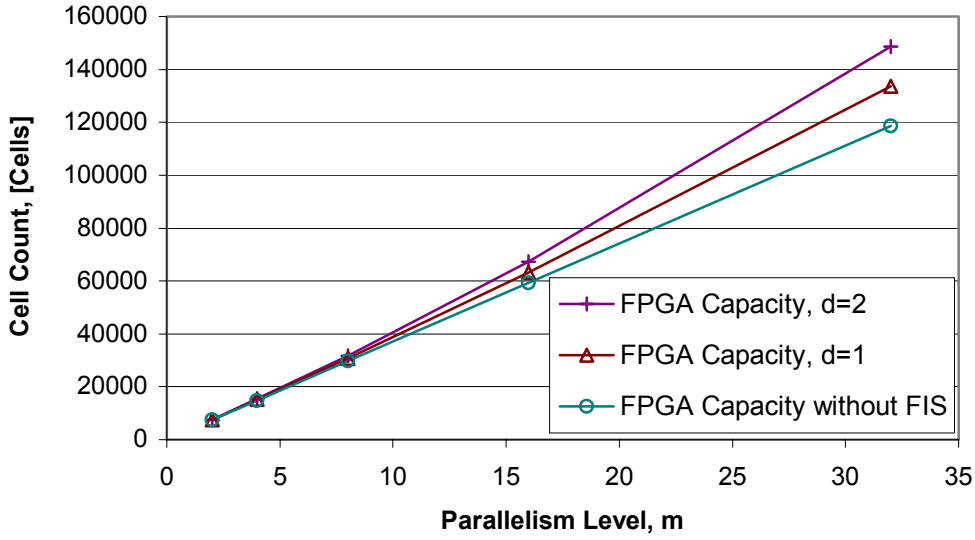


Figure 20: Parallel Turbo Decoder FPGA Chip Capacity vs. Parallelism Level

6. Performance Analysis

This section contains the analysis and simulation results for the parallel decoder.

6.1. Spread and Dispersion

The spread and dispersion performance of the algorithm for different configurations of parallel decoder (*N, m, d*) are presented in Figure 21 and Figure 22.

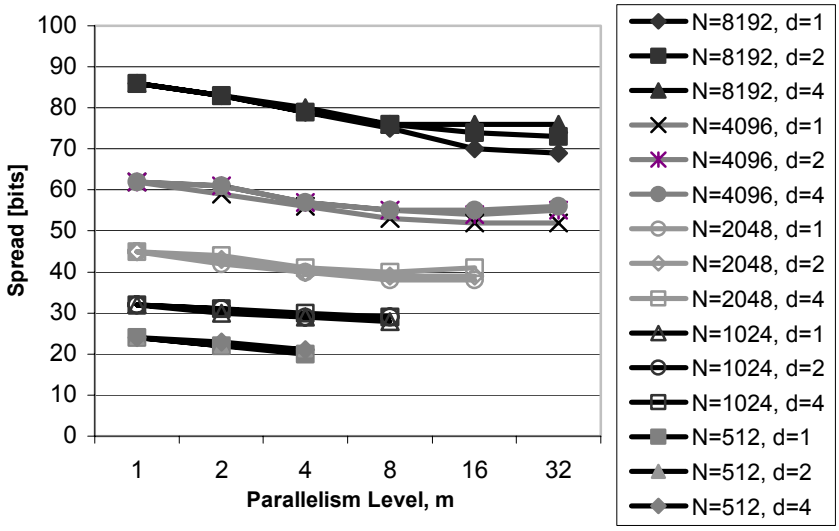


Figure 21: PI Spread Results for Different N, m and d

Slight spread degradation relatively to *sequential* interleaver (*m*=1, *d*=1) is observed as parallelism level, m, goes higher. A slight spread improvement is achieved when *d* (the PI delay) is increased, thanks to larger delay packets. Most of the improvement occurs when *d* is increased from *d*=1 to *d*=2.

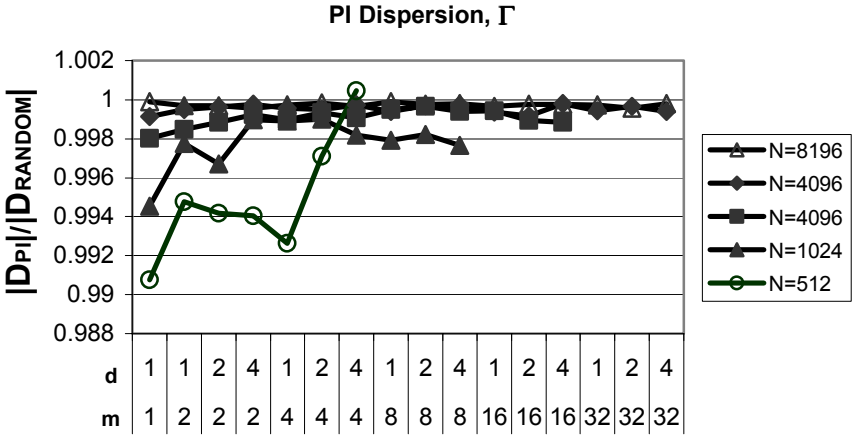


Figure 22: PI Dispersion results for different N, m and d

The interleaver dispersions (Figure 22) are very close to that of a random interleaver (where Γ=1). With such dispersion and high spread characteristics, the decoder achieves a high error correction performance, as shown in Section 6.3 below.

6.2. Throughput and Latency

Parallel and sequential architectures were compared in terms of latency and throughput for a given silicon area. Performance is highly correlated to *NDU* (number of decoding units) and *m* (level of parallelism; see Eq. (12)-(17)). The higher *NDU* and *m* are, the more efficient the area utilization; parallel architectures are more area efficient thanks to the fact that, as we add more SISOs, no additional memories and almost no additional logic are required. When, on the other hand, we wish to add more SISOs to a sequential architecture, the entire decoding unit must be duplicated.

The ratio of throughput $F_{in}^{Uncoded}$ of the parallel architecture to that of the sequential one for different block sizes vs. area is shown in Figure 23. It can be seen that for larger chip area, the parallel architecture can handle larger blocks more efficiently, and higher input data rates are accommodated.

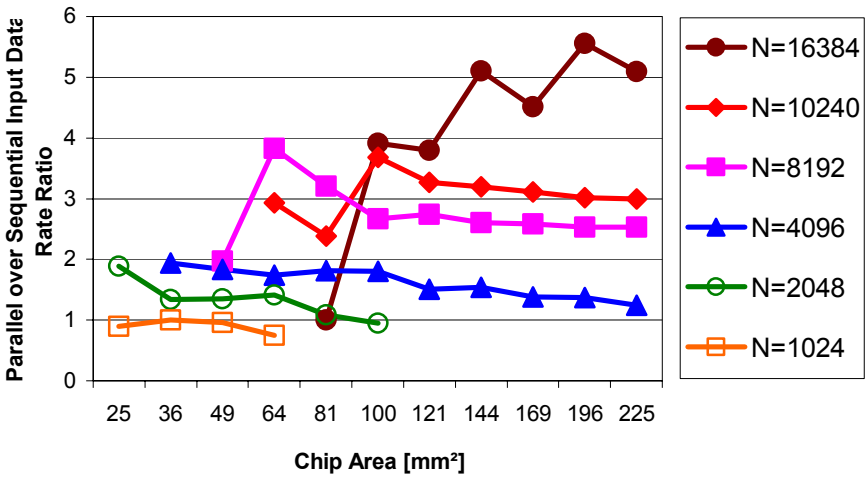


Figure 23: Parallel over Sequential Throughput Ratio vs. Area

The latency reduction is summarized in Figure 24. A linear speedup with chip area increase is evident in the chart. Recall that the level of parallelism is also linear in chip area (Figure 19), thus achieving an attractive cost/performance ratio.

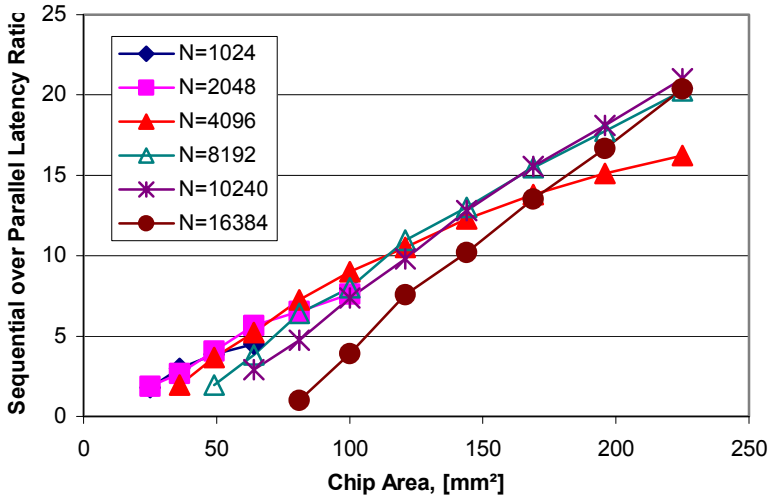


Figure 24: Sequential over Parallel Latency Ratio vs. Chip Area

6.3. BER Performance

The parallel decoder was simulated over AWGN channel using BPSK modulation. The results refer to rate 1/3, 2/3 and 3/4 turbo code with two identical 8-state convolutional encoders with g_0=13, g_1=17 generator [30]. The decoder performed 10 decoding iterations, using a 32-bit sliding window.

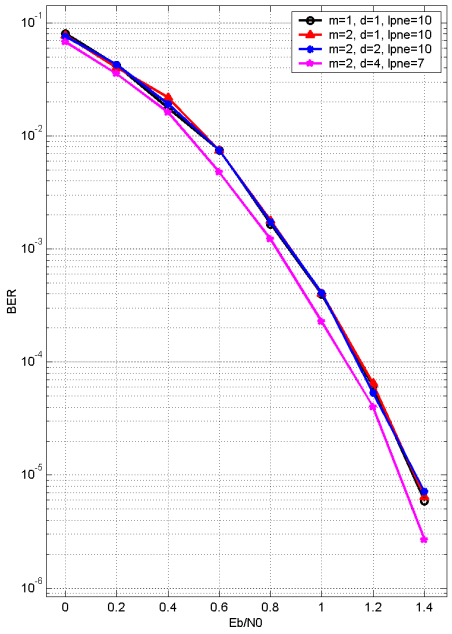


Figure 25: BER Results, N=512, CR=1/3

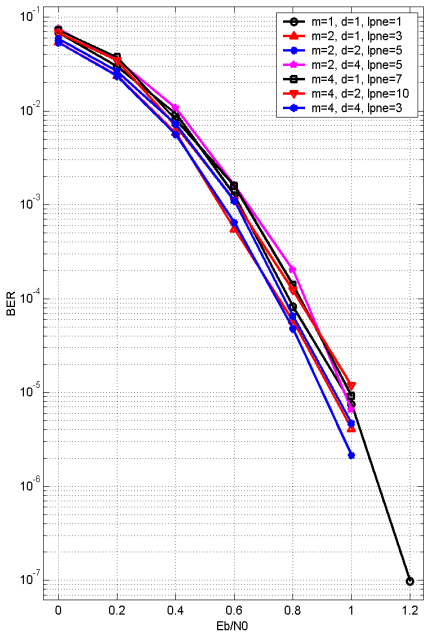


Figure 27: BER Results, N=1024, CR=1/3

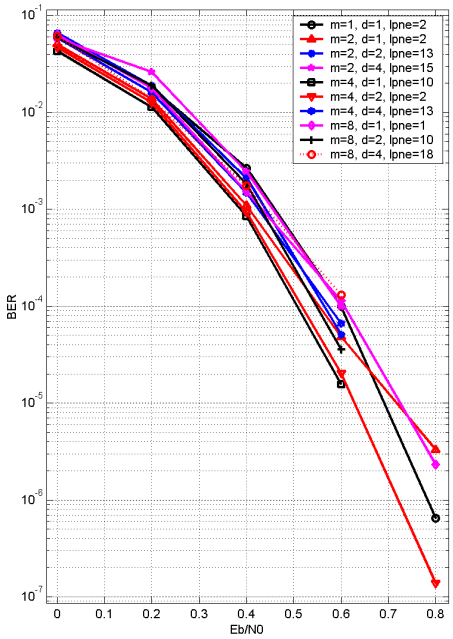


Figure 26: BER Results, N=2048, CR=1/3

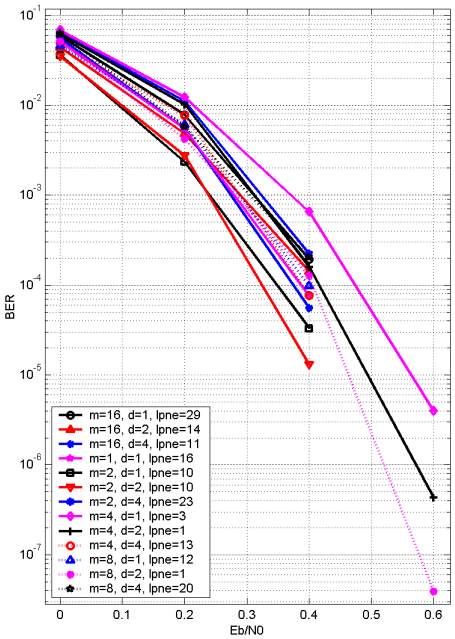


Figure 28: BER Results, N=4096, CR=1/3

The results in Figure 28 refer to the code rate 1/3. “lpne” in the legend stands for **l**ast **p**oint **n**umber of **e**rrors, corresponding to the number of error events that occurred for the last computed BER point for the given (*m*, *d*) configuration. For other points the number of the error events is in range 10-100. The “lpne” is given in order to provide the reader the confidence level for the last measurements at a very low BER.

The results show slight deviations relative to the sequential decoder (*m=1, d=1*). As evident from the results for all different block lengths and for different examined *m* values, performance is within 0.05 dB of the sequential turbo decoder. For the larger block lengths the small degradation is compensated by applying *d*=2. For some configurations of the decoder, the obtained results outperform marginally over the (*m=1, d=1*) configuration. This is a result of variations of the interleaver search algorithm. In any case, the sequential architecture, which can implement any of *N*! possible permutations, can implement the obtained (*m, d*) permutations as well.

The results in Figure 29 and Figure 30 refer to the code rates 2/3 and 3/4 respectively. The mentioned rates were obtained by puncturing the outputs of the convolutional encoders (see Figure 1). The results are similar to those for code rate 1/3.

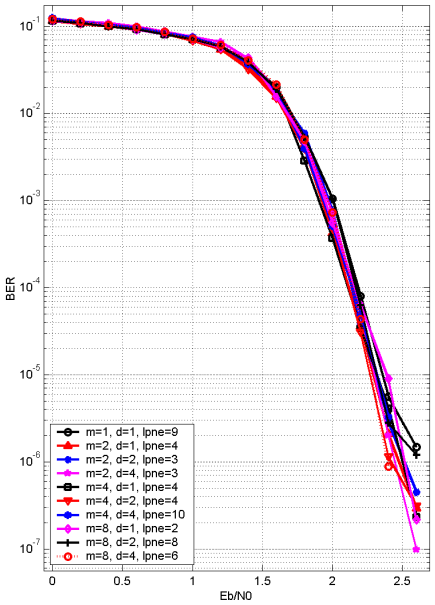


Figure 29: BER Results, N=2048, CR=2/3

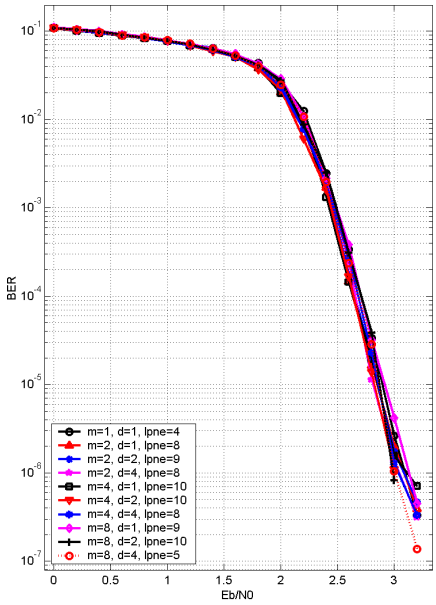


Figure 30: BER Results, N=2048, CR=3/4

7. Conclusions

A new parallel turbo decoder VLSI architecture was presented. The architecture of the parallel interleaver was detailed and a new interleaver design algorithm was introduced. A significant linear reduction of latency was achieved (up to a factor of 20) in comparison with a sequential turbo decoder. In addition, it was found that for large blocks the parallel architecture is more area efficient, improving throughput up to a factor of 5 for the same chip. The error correction performance was within 0.05 dB of that of the sequential turbo decoder. The parallel architecture and the parallel interleaver design algorithm achieved an attractive cost/performance ratio and an attractive performance in terms of BER, latency and throughput.

References

[1] C. Berrou, A. Glavieux, P. Thitimajshima, "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes," Proc. of ICC'93, pp. 379-427, Geneva, Switzerland, 1993.

[2] C. Berrou, A. Glavieux, "Near Optimum Error Correcting Coding And Decoding: Turbo-Codes," IEEE Transactions on Communications, vol. 44, no. 10, pp. 1261-1271, 1996.

[3] G. Masera, G. Piccinini, M. R. Roch, M. Zamboni, "VLSI Architectures for Turbo-Codes," IEEE Transactions on VLSI Systems, vol. 7, no. 3, pp. 369-379, 1999.

[4] Z. Wang, Z. Chi, K. K. Parhi, "Area-Efficient High Speed Decoding Schemes for Turbo/MAP Decoders," Proc. of 2001 IEEE Int. Conf. on Acoustics, Speech and Signal Processing, Salt Lake City, Utah, pp. 2633-2636, May 2001.

[5] G. Park, S. Yoon, C. Kang and D. Hong, "An Implementation Method of a Turbo-code Decoder using a Block-wise MAP Algorithm," VTC Fall 2000, Boston, USA, Sept. 2000.

[6] Peter A. Beerel, and Keith M. Chugg, "A Low Latency SISO Application to Broadband Turbo Decoding," IEEE Journal on selected areas in communications, Vol.19, No.5, May 2001.

[7] J. Hsu, C. Wang, "A Parallel Decoding Scheme for Turbo Codes," Proc. of ISCAS'98, vol.4, June 1998, pp. 445-448.

[8] S. Yoon, Y. Bar-Ness, "A Parallel MAP Algorithm for Low Latency Turbo Decoding," IEEE Communications Letters, Vol. 6, No. 7, July 2002.

[9] B. Bougard, A. Giulietti, V. Derudder, J. W. Weijers, S. Dupont, L. Hollevoet, F. Cathhoor, L. V. der Perre, H. De Man, R. Lauwereins, "A Scalable 8.7nJ/bit 75.6 MB/s Parallel Concatenated Convolutional (TURBO-) CODEC," ISSCC 2003, San Francisco, USA, pp. 152-153, Feb. 2003.

[10] A. Giulietti, B. Bougard, V. Derruder, S. Dupont, J. W. Weijers, L. V. der Perre, "A 80 Mb/s Low-power Scalable Turbo Codec Core," Orlando, USA, CICC'02, May 2002.

[11] F. Gilbert, M. J. Thul, N. When, "A Scalable System Architecture for high throughput turbo-decoder," SIPS'02, San Diego, USA, pp. 152-158, Oct. 2002.

[12] Z. Wang, Z. Chi, and K. K. Parhi, "Area-Efficient High-Speed Decoding Schemes for Turbo Decoders," IEEE Transaction on VLSI, vol. 10, No. 6, pp. 902-912, Dec. 2002.

[13] R. Dobkin, M. Peleg, R. Ginosar, "Parallel VLSI Architecture for MAP Turbo Decoder," Proc. of PIMRC'2002, vol. 1, Sept. 2002, pp. 384-388.

[14] M. J. Thul, F. Gilbert, N. When, "Concurrent Interleaving architectures for high-throughput channel coding," ICASSP'03, Hong Kong, Vol. 2, pp. 613-616, Apr. 2003.

[15] A. Giulietti, L. van der Perre, M. Strum, "Parallel turbo coding interleavers: avoiding collisions in accesses to storage elements," Electronic Letters, Vol. 38, No. 5, pp. 232-234, Feb. 2002.

[16] S. Crozier, "Turbo-Code design Issues: Trellis Termination Methods, Interleaving Strategies, and Implementation Complexity," Proc. of ICC'99, Vancouver, Canada, 1999.

[17] C. Berrou, "Additional information on the EUTELSAT/ENST-Bretagne proposed channel turbo coding for DVD_RCS," Ad Hoc Group on Return Channel over Satellite, 6th meeting, Geneva, Switzerland, 1999.

[18] C. Schurgers, F. Catthoor, M. Engles, "Optimized MAP Turbo Decoder," The 2000 IEEE Workshop on Signal Processing Systems, Lafayette, Louisiana, USA, 2000.

[19] J. A. Erfanian, S. Pasupathy and G. Gulak, "Reduced Complexity Symbol Detectors with Parallel Structures," Globecom'90, San Diego, USA, pp. 704-708, Dec. 1990.

[20] S. S. Pietrobon, A. S. Barbulescu, “A Simplification of the Modified Bahl Decoding Algorithm for Systematic Convolutional Codes,” Int. Symposium on Information Theory & Its Applications, Sydney, Australia, pp. 1073-1077, Nov. 1994.
[21] C. Schurgers, F. Catthoor, M. Engels, “Memory Optimization of MAP Turbo Decoder Algorithms,” IEEE Transactions on VLSI Systems, vol. 9, no.2, pp. 305-312, 2001.
[22] A. Hunt, S. Crozier, M. Richards, K. Gracie, “Performance Degradation as a Function of Overlap Depth when using Sub-Block Processing in the Decoding of Turbo Codes,” Proc. of IMSC’99, pp. 276-280, Ottawa, Canada, 1999.
[23] H. Dawid and H. Meyr, “Real-Time Algorithms and VLSI Architectures for Soft Output MAP Convolutional Decoding,” PIMRC'95, Toronto, Canada, pp 193-197, Sept. 1995.
[24] S. N. Crozier, “New High-Spread High-Distance Interleavers for Turbo Codes,” 20-th biennial Symposium on Communications, pp. 3-7, Kingston, Canada, 2000.
[25] C. Heegard, S. B. Wicker, “Turbo Coding.” Kluwer Academic Publishers, 1999, pp. 50-52.
[26] Solomon W. Golomb, and Herbert Taylor, “Construction and properties of Costas Arrays,” Proceedings of the IEEE, Vol. 72, No. 9, Sep. 1984, pp.1143-1163.
[27] J. D. Andersen, “Selection of code and interleaver for turbo coding,” First ESA Workshop on Tracking, Telemetry and Command systems ESTEC, The Netherlands**,** June 1998.
[28] D. Weinfeld, “Symbol-wise Implementation of Turbo/MAP Decoder,” Internal report, Technion – Israel Institute of Technology, EE Department, Communications Laboratory (ISIS Consortium), July 2002.
[29] W. Feng, J. Yuan, B. S. Vucetic, “A Code-Matched Interleaver Design for Turbo Codes,” IEEE Transactions on Communications, Vol. 50, No.6, June 2002, pp.926-937.
[30] M. S. C. Ho, S. S. Pietrobon, T. Giles, “Improving the Constituent Codes of Turbo Encoders,” IEEE Globecom’98, Vol. 6, pp. 3525-3529, November 1998.