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

Rostislav (Reuven) Dobkin, Michael Peleg, *Senior Member, IEEE*, and Ran Ginosar

***Abstract*—Standard VLSI implementations of turbo decoding require substantial memory and incur a long latency, which cannot be tolerated in some applications. A parallel VLSI architecture for low-latency turbo decoding, comprising multiple single-input single-output (SISO) elements, operating jointly on one turbo-coded block, is presented and compared to sequential architectures. A parallel interleaver is essential to process multiple concurrent SISO outputs. A novel parallel interleaver and an algorithm for its design are presented, achieving the same error correction performance as the standard architecture. Latency is reduced up to 20 times and throughput for large blocks is increased up to six-fold relative to sequential decoders, using the same silicon area, and achieving a very high coding gain. The parallel architecture scales favorably: latency and throughput are improved with increased block size and chip area.**

***Index Terms*—Decoders, interleaver, maximum *a posteriori* (MAP) algorithm, parallel architecture, turbo codes, VLSI architecture.**

I. 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 3G standard.

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]–[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 a large amount of memory per SISO for M turbo blocks' storage. Hence, enhancing throughput by duplicating SISO 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 lowering 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 SISOs in parallel, each processing one of the sliding windows. Those schemes trade off the number of SISOs 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], employ the sliding window approach *inside* each subblock that is decoded in parallel. In our approach [13], the subblocks are processed in a similar way, while the definitions of the boundary metric values for the beginning and the end of the block are improved by use of tailbiting termination. Interleaving approaches for parallel decoders were presented in [14], [15], and [13]. A recently introduced parallel interleaver architecture [16], similar to the structure presented here, enables an unconstrained implementation of any interleaver. The VLSI performance results presented here apply also to that interleaving approach.

This paper presents a complete analysis of parallel VLSI architecture, partly presented by us in [13]. Our architecture allows choosing the number of SISO decoders independently of the desired sliding window and block size. Parallel interleaving of multiple concurrent SISOs' outputs is an essential element of this architecture, affecting error correction performance, latency, and throughput of the entire decoder. The paper presents a new parallel interleaver (PI) of moderate complexity, able to achieve the error correcting performance of standard sequential architecture, and a new algorithm for PI design, comprising spread optimization and elimination of low-weight error patterns. We discuss the architecture, implementation, and performance of the PI for different levels of parallelism. The parallel decoder architecture significantly reduces both latency (up to 20-fold) and hardware complexity, and improves decoder throughput (up to six-fold) relative to sequential decoder using the same chip area. No significant coding gain degradation was observed. Performance of parallel and sequential architectures is compared.

Pertinent aspects of turbo coding and the sequential decoder architecture are described in Sections II and III, respectively. The novel parallel decoding architecture is presented in Section IV. Section V presents the parallel interleaver architecture and its design algorithm. In Section VI, the parallel and sequential architectures are compared in terms of coding gain, throughput, and latency.

II. Turbo Coding—Theory of Operation

A. Encoder

A turbo encoder consists of convolutional encoders connected either in parallel or in series. The parallel scheme [2] (Fig. 1) comprises an interleaver and two parallel convolutional

Manuscript received January 30, 2003; revised August 19, 2003; October 10, 2004.

The authors are with the Electrical Engineering Department, Technion—Israel Institute of Technology, Haifa 32000, Israel (e-mail: rostikd@tx.technion.ac.il; michael@lena.technion.ac.il; ran@ee.technion.ac.il).

Digital Object Identifier 10.1109/TVLSI.2004.842916

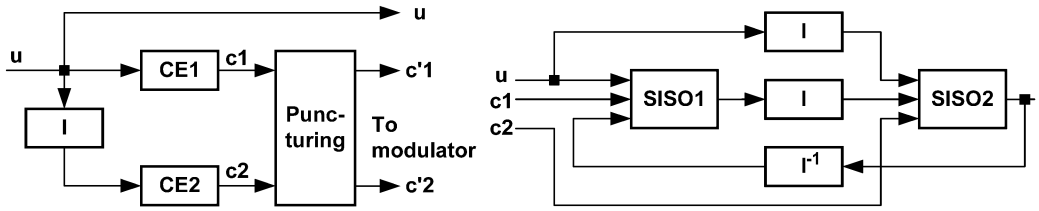

Fig. 1. Turbo encoder and decoder (I: interleaver).

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, affecting performance [2], [17].

For every new input block, the encoder starts at a known trellis state. In order to terminate at a known trellis state, traditionally some extra input termination bits are generated. Most termination techniques [17] result in changes in block size and code rate, and in some cases at least one of the added terminations is not interleaved. *Tailbiting* termination [18] for recursive convolutional codes keeps the block size unchanged. A data-dependent initial state is identified such that the initial and final states of the encoder are identical. In this paper we employ tailbiting termination.

B. Decoder

Decoding is performed iteratively (Fig. 1): information from one SISO is processed by the other SISO until convergence is achieved. Each SISO produces an increasingly better correction term *(extrinsic information),* which is (de)interleaved and used as a-priori information by the next SISO [2]. According to the original Bahl, Cocke, Jelinek, and Raviv (BCJR) algorithm [19], the SISO computes forward metrics (α), backward metrics (β), and output metrics (probability values obtained as SISO output).

BCJR decoding is performed by first computing and storing β metrics for the entire block (going backward), and then computing α and output metrics (going forward). This approach, however, suffers from very high latency and memory requirement [20].

Latency and memory size are significantly reduced by the *sliding window* approach [3], [20]–[22]. Backward (and/or forward) metrics are initialized at an intermediate point instead of the end or the beginning of the block. Degradation due to this optimization is negligible when an appropriate intermediate point is used, providing sufficient window overlap size [3], [21].

When tailbiting termination is employed, the last window length (WL) bits of the block *(tail window)* are sent to the SISO prior to the entire block [13], [20]. The SISO performs a dummy α calculation over that 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.

C. Interleaver

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

Interleaver spread optimization is desirable for both fast convergence and good distance properties of the code. Large distance lowers the point at which the bit error rate (BER) curve flattens ("error floor") and increases the slope of the BER. We adopted a "high spread" definition following [23]. Let $I(i), I(j)$ be the locations of the interleaver outputs i and j at the input of the interleaver. The spread associated with i and j is

$$S'_{\mathrm{HS}}(i,j) = |I(i) - I(j)| + |i - j|. \tag{1}$$

The minimal "high spread" S_{HS} associated with the entire interleaver is

$$S_{\mathrm{HS}} = \min_{i,j} \left[S'_{\mathrm{HS}}(i,j)\right]. \tag{2}$$

When tailbiting termination is used, distance calculations, such as $|i - j|$, are performed considering the cyclic property of the tailbiting trellis. Thus the tailbiting distance for two indexes i and j is defined as follows (N is the interleaver size)

$$D_{\mathrm{Tail}}(i,j,N) = \min[|i - j|, N - |i - j|]. \tag{3}$$

A pair of possible input–output spans is called *spreading factors* [24]. An alternative description of spreading factors is described in terms of the *displacement vector*

$$(\Delta_X, \Delta_Y) = (j - i, I(j) - I(i)), \quad i < j. \tag{4}$$

The "randomness" of the interleaver is another factor affecting performance. Regularly structured interleavers, such as classical block interleavers, perform poorly for turbo codes. The set of the interleaver displacement vectors can be used to study "randomness" [24]

$$D(I) = \{(\Delta_X, \Delta_Y) \in Z^2 | \Delta_X = j - i, \Delta_Y = I(j) - I(i),\ 0 \le i < j < N\}. \tag{5}$$

The largest set of displacement vectors occurs for *Costas* permutation [24], [25], in which the number of displacement vectors is $N \cdot (N-1)/2$. The normalized dispersion γ is then defined as follows:

$$\gamma = \frac{2 \cdot |D(I)|}{N \cdot (N-1)}, \text{ obtaining } \frac{2}{N} \le \gamma \le 1 \tag{6}$$

where $|D(I)|$ is the size of the set of the displacement vectors $D(I)$. However, we cannot use that definition of dispersion, due to tailbiting. Therefore, we consider the ratio of the number of

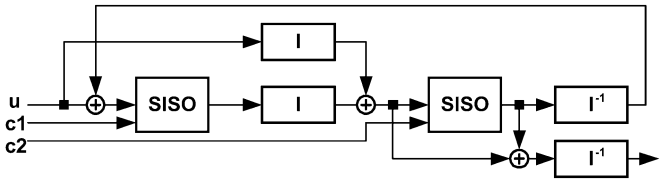

Fig. 2. Iterative decoder scheme (I: interleaver).

parallel interleaver displacement vectors (see Section VI-A) and the dispersion of a random interleaver

$$\Gamma = \frac{|D(I_{\mathrm{PI}})|}{|D(I_{\mathrm{Random}})|}. \tag{7}$$

Using recursive systematic codes, single 1s yield codewords of semi-infinite weight, and low-weight words appear with patterns of two, three, and four errors in the information bits [26]. 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 pattern elimination contributes drastically to the code performance at the error floor region [26]. We define the 2-bit error pattern spread [26] $S_{\min}^{2\text{-}bit}$ involving interleaver outputs i and j, according to (1). The 4-bit error pattern [26] spread $S_{\min}^{4\text{-}bit}$, is defined analogously [20]. Our algorithm eliminates 2- and 4-bit error patterns (Section V-C).

III. Sequential Decoder Architecture

An iterative decoder detailed scheme is shown in Fig. 2. An iteration through the decoder can be divided into two stages: the *interleaving stage,* where the result of the previous iteration plus u, and $c1$ bits are processed by SISO1 and passed through interleaver I; and the *deinterleaving stage,* where the extrinsic data from the interleaving stage plus an interleaved version of u, and $c2$ are processed by SISO2 and deinterleaved. 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 interleaving and deinterleaving while suitable addresses are provided. Therefore, these two stages can be implemented by the same hardware block, used twice for each iteration.

A *decoding unit* consists of a SISO, an interleaver memory, an adder, memories for channel data (u, $c1$, $c2$), (u, $c1$, $c2$), (u, $c1$, $c2$) u, $c1$, $c2$), an 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 first in, first outs (FIFOs).

The maximal input rate $F_{in,seq}^{\mathrm{Uncoded}}$ for the sequential architecture (with input double buffer) is [13], [20]

$$F_{\mathrm{in,seq}}^{\mathrm{Uncoded}} = \frac{NDU}{2 \cdot NI} \cdot F_{\mathrm{int,seq}}^{\mathrm{eff}} \tag{8}$$

$$F_{\mathrm{int,seq}}^{\mathrm{eff}} = \frac{F_{\mathrm{int}} \cdot N}{N + c \cdot WL} \tag{9}$$

where

NDU	number of decoding units;
NI	number of iterations;
N	block size;

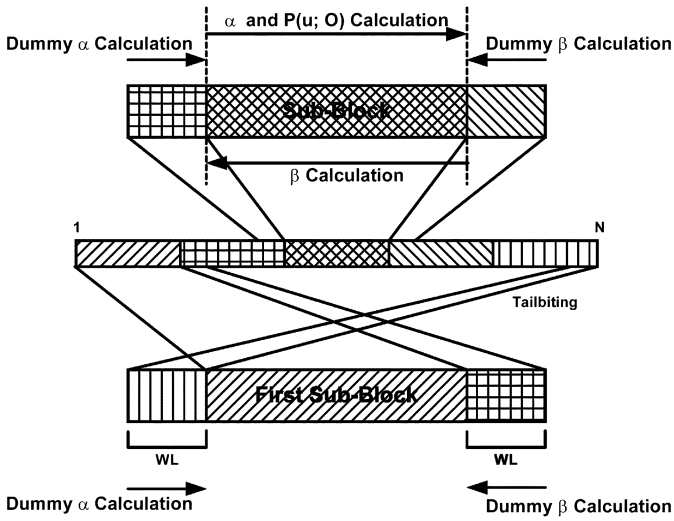

Fig. 3. Decomposition to subblocks example (WL: window length).

WL	window length;
C	SISO delay in window lengths;
$F_{\mathrm{int,seq}}^{\mathrm{eff}}$	effective processing rate;
F_{int}	internal clock rate.

For a given silicon area, the throughput of the decoder $F_{\mathrm{in,seq}}^{\mathrm{Uncoded}}$ 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.

The latency of the sequential architecture is that of the decoding unit

$$D_{\mathrm{Seq}} = \frac{2 \cdot NI \cdot (N + c \cdot WL)}{F_{\mathrm{int}}}. \tag{10}$$

The latency consists of the delay of the practical SISO due to prior input of five windows ($c = 5$ for the algorithm with tailbiting termination) in addition to processing N metrics of the block, all multiplied by the number of iterations.

IV. Parallel Decoder Architecture

The parallel decoding architecture applies m SISOs in parallel to one incoming block. The block is decomposed into m subblocks. The decomposition of processing to subblocks is facilitated by applying the sliding window principle, which allows independent decoding of subblocks without degradation in error correction performance [21]. Dummy α and β metrics are calculated in order to determine the initial values of α and β for each subblock i. An example of block decomposition to $m = 5$ subblocks is shown in Fig. 3.

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

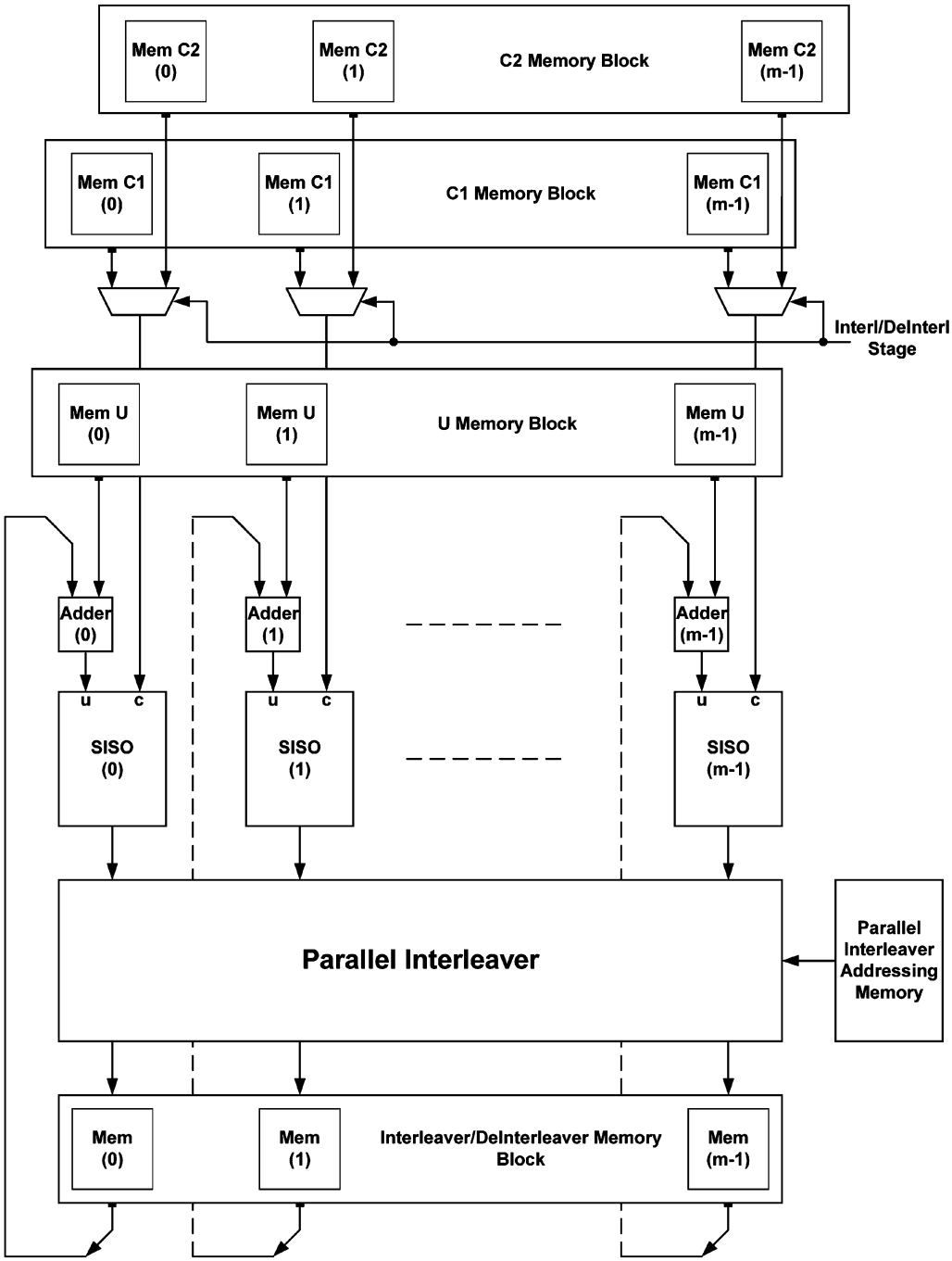

Fig. 4. Parallel decoder architecture.

Other works have also divided the block into subblocks [7], [8]. However, the algorithms proposed there did not apply a sliding window inside subblocks. The technique proposed here creates boundary metrics for the first and last subblocks using the *same* process as for the other subblocks, thus differing from [10]–[12], and causing no performance degradation nor increasing the computational load.

Alternatively, subblock decomposition could be performed in the encoder, encoding each subblock separately with its own tailbiting termination [27]. Thus, there would be no need to exchange dummy metrics between the parallel data flows in the decoder. Incurring the same decoder complexity and achieving similar throughput and latency as the previous approach, this scheme may be a subject for future research.

Iterative parallel processing is executed by the architecture shown in Fig. 4, as follows. The incoming block is divided into m subblocks, and each subblock 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 Fig. 4). The interleaver memory and the channel data memories (U, C1, and C2 memory blocks) consist each of an array of m memories

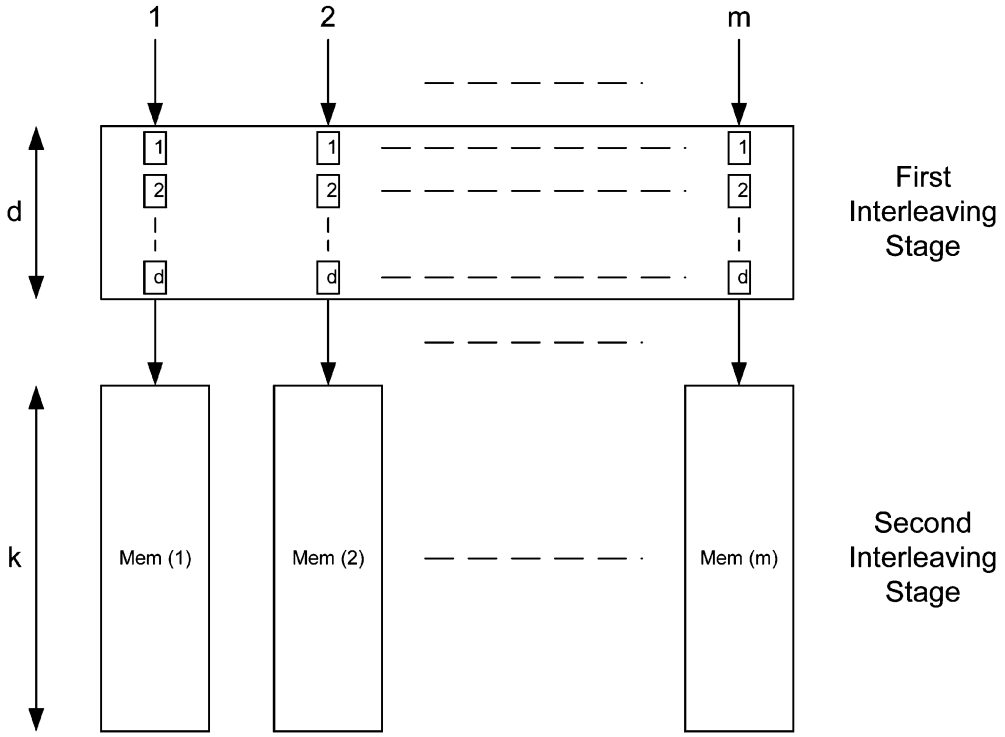

Fig. 5. The parallel interleaver.

of depth N/m. The PI performs interleaving according to addresses supplied by its addressing memory. The PI architecture is discussed in Section V.

The maximal input rate $F_{in,par}^{\text{Uncoded}}$ for the parallel architecture is [13], [20]

$$F_{in,par}^{\text{Uncoded}} = \frac{m}{2 \cdot NI} \cdot F_{\text{int},par}^{\text{eff}} \tag{11}$$

where

$$F_{\text{int},par}^{\text{eff}} = \frac{F_{\text{int}} \cdot \frac{N}{m}}{\frac{N}{m} + c \cdot WL} \tag{12}$$

and m is the number of SISOs, defining the parallelism level of the decoder.

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

$$D_{\text{Par}} = \frac{2 \cdot NI \cdot \left(\frac{N}{m} + c \cdot WL\right)}{F_{\text{int}}}. \tag{13}$$

V. Parallel Interleaver

A. Architecture

The PI plays a key role in the performance of the parallel decoder. It comprises the first interleaving stage (FIS) and the second interleaving stage (SIS). The FIS, which can hold up to $m \times d$ metrics, permutes the m metrics coming simultaneously from m SISOs (FIS depth d is termed also as FIS delay in the following). The m outputs of the FIS are directed into the m memories that constitute the SIS (Fig. 5). Each SIS memory can hold and permute at least $k = N/m$ metrics.

At each calculation cycle, m metrics (from m SISOs) enter the FIS. At the beginning, the FIS accumulates $m \cdot d$ metrics for d cycles and then outputs sets of m metrics at 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*. FIS permutations are designated to be a permutation of these indexes, turning them into the *SIS memory destination index* and the *output cycle number*.

After FIS interleaving, the data arrive in the SIS, where intrasubblock permutation is performed by properly addressing each of the m memories. In order to perform deinterleaving, an additional FIS unit with reversed FIS addresses is incorporated in the design. In [16], the deinterleaving stage is supplied with different addressing allowing unconstrained interleaver implementation. Thanks to the very similar structure, the results presenting in this paper (Section VI) are applicable to the architecture employing the approach of [16].

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. 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.

B. Possible Permutations

Due to the structure of PI, the number of possible permutations is limited, in comparison with the turbo interleaver used in the sequential decoder. The standard interleaver of size N, implemented as a memory, can perform $N!$ permutations.

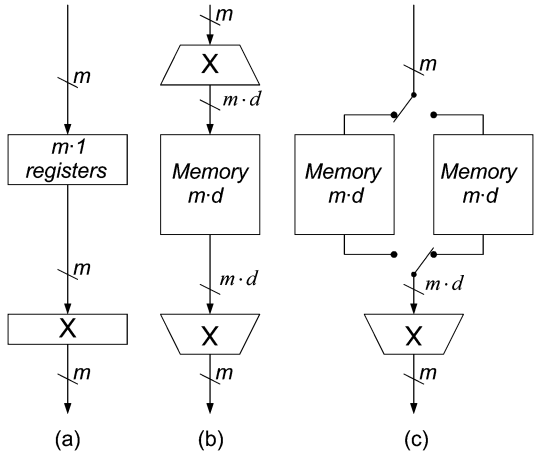

Fig. 6. Alternative FIS architectures: (a) crossbar, (b) infinite permutation network, and (c) finite permutation network. X represents a crossbar switch.

We considered three alternative FIS architectures: a *crossbar,* an *infinite permutation network,* and a *finite permutation network* (Fig. 6). Wide input and output memories are combined with crossbar 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 an $m \times m$ crossbar switch, facilitating $m!$ permutations. Note that two metrics that arrive in the same input cycle can *never* end up in the same target SIS memory [Mem(i) in Fig. 5].

In the (most powerful) *infinite permutation network* architecture, an $m \times (m \cdot d)$ crossbar spreads the incoming m metrics 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 m 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, while 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 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 FIFO sizes, was recently proposed in [14]. The multiple multi-input FIFO solution becomes very expensive with growing block size and level of parallelism.

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 V-D 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 us assume that $m{\cdot}d$ divides $N (N | m{\cdot}d = 0)$. The depth of the SIS memory (and the number of output cycles of the SISO's 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 total number of possible permutations for the entire PI is [20]

$$NPerm = \left[\frac{(m \cdot d)!}{(d!)^m}\right]^{N/(m \cdot d)} \cdot (k!)^m. \tag{14}$$

The FIS can perform $(m \cdot d)!/(d!)^m$ permutations on one delay packet; there are $N/(d \cdot m)$ delay packets, and the entire SIS can perform $(k!)^m$ permutations. For example, for $d = 1$ (*crossbar* architecture)

$$NPerm = (m!)^{N/m} \cdot (k!)^m = (m!)^k \cdot (k!)^m.$$

For $m = 1$ (one SISO, sequential architecture)

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

As can be seen from (14), the number of possible permutations depends on three parameters: N, m, and d. Increasing d expands the delay packet size, resulting in a growing number of possible permutations. Fig. 7 shows the results for $N = 4$ K and four different delay values in the log-domain. Stirling's approximation was used in the computation. Thus $d > 1$ compensates for most of the decrease resulting from partitioning. In Fig. 7, $\ln(NPerm)$ is normalized by the logarithm of the maximal number of permutations $N!$. 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 level of parallelism grows, the degradation levels off thanks to the growing size of the delay packet $(m \cdot d)$.

C. Parallel Interleaver Design

1) Interleaver Design Algorithm: Given a PI with certain N, m, d parameters and *finite permutation network* FIS, the high-performance PI design algorithm (Figs. 8 and 9) generates the required permutations according to the high-spread criteria [(2)], while eliminating the 2- and 4-bit error patterns (Section II-C) and considering the tailbiting restrictions of (3). The algorithm comprises the following stages (stage numbers correspond to the markings in the figures).

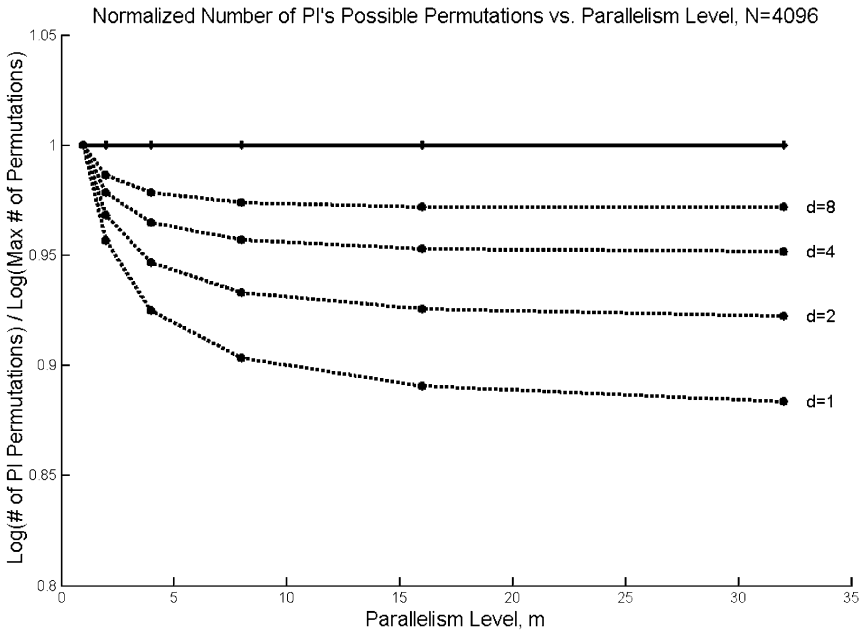

Fig. 7. Normalized number of permutations for various delays versus the level of parallelism.

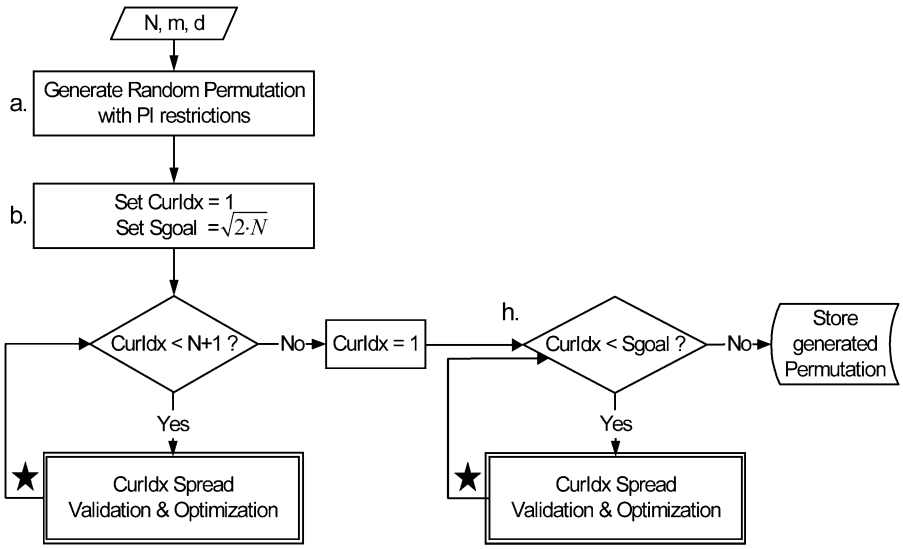

Fig. 8. Parallel interleaver design algorithm—main flow.

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

b) The current spread value S_{goal} is initialized to the maximal possible value. While according to [23] $S_{\text{goal}}^{Max} = \sqrt{2 \cdot N}$, 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 $(S_{\min})$ calculation according to (2). The spread is calculated for CurIdx $(S_{\min})$ and for the 2- and 4-bit patterns related to the CurIdx $\left(S_{\min}^{2\text{-}bit} \text{ and } S_{\min}^{4\text{-}bit}\right)$.

d) In order to satisfy total permutation spread, S_{goal} [the minimal spreads computed at stage c)] should satisfy the following inequalities:

$$\begin{aligned} &\text{i. } S_{\min} \geq S_{\text{goal}} \\ &\text{ii. } S_{\min}^{2\text{-}bit} \geq 2 \cdot S_{\text{goal}} \\ &\text{iii. } S_{\min}^{4\text{-}bit} \geq 2 \cdot S_{\text{goal}}. \end{aligned} \tag{15}$$

e) When (15) is satisfied, the algorithm accepts CurIdx and begins treating the next index CurIdx+1.

f) If (15) is not satisfied, $P[\text{CurIdx}]$ is rejected and replaced as follows.

i) The algorithm searches for a set of indexes, which can be swapped with $P[\text{CurIdx}]$. The suitable indexes

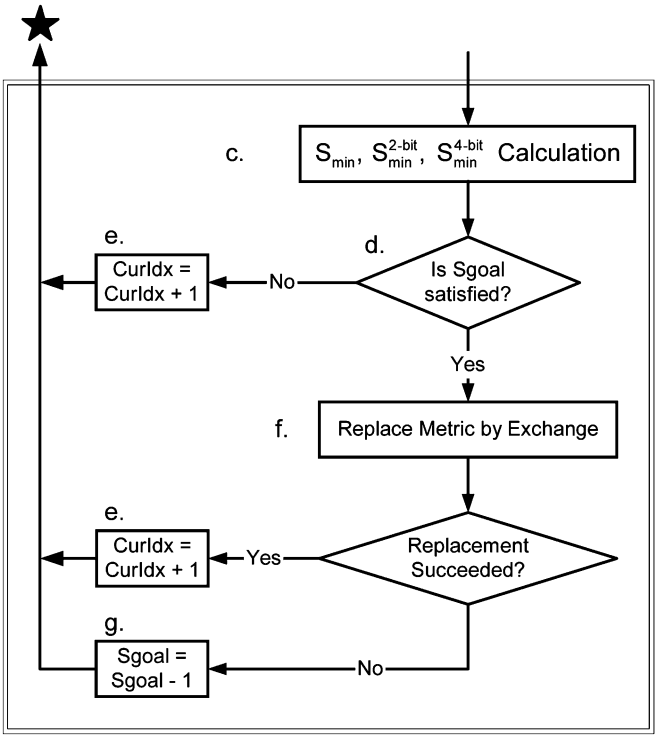

Fig. 9. Parallel interleaver design algorithm—CurIdx spread validation and optimization.

must belong to the same *delay packet* or to the same *SIS memory* as CurIdx, and satisfy (15) after the swap.

ii) If such a set is found, $P[\text{CurIdx}]$ 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 S_{goal} constraint is reduced.

h) Due to tailbiting, when the algorithm reaches index N, the first S_{goal} indexes of the permutation should be recomputed.

This algorithm converges fast in practice. It arrives very close to the highest spread value within a few dozens of search iterations. The algorithm was developed based on results presented in [17], [23], and [26]. A similar approach (for the design of a sequential interleaver) has recently been presented in [28], achieving performance very close to the application of our algorithm to the sequential case $(m = 1)$.

D. VLSI Implementation

The PI consists of two stages: FIS and SIS (Fig. 5). SIS is implemented by an array of memories that perform interleaving using addresses generated by the algorithm of Section V-C. The following FIS implementation is optimized for the finite permutation network architecture (Fig. 6).

The 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 (Fig. 6) in a double-buffer setup; only one array is shown in Fig. 10.

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 every time d sets are read and written.

Application-specific integrated circuit (ASIC) and field-programmable gate array (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. Fig. 11 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. Fig. 12 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 achievable when using a shorter metric representation (thanks to SISO internal optimizations).

VI. Performance Analysis

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

A. Spread and Dispersion

The spread and dispersion performance of the PI design algorithm for various configurations of the parallel decoder (N, m, d) are presented in Figs. 13 and 14.

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

The interleaver dispersions (Fig. 14) are very close to that of a random interleaver (where $\Gamma = 1$). With such dispersion and high spread characteristics, the decoder achieves a high error correction performance, as shown in Section VI-C.

B. 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 the number of decoding units NDU and level of parallelism m; see (8)–(13). 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 various block sizes is charted in Fig. 15 as a function of chip area. It can be seen that for larger area, the parallel architecture can handle larger blocks more efficiently, and higher input data rates are accommodated.

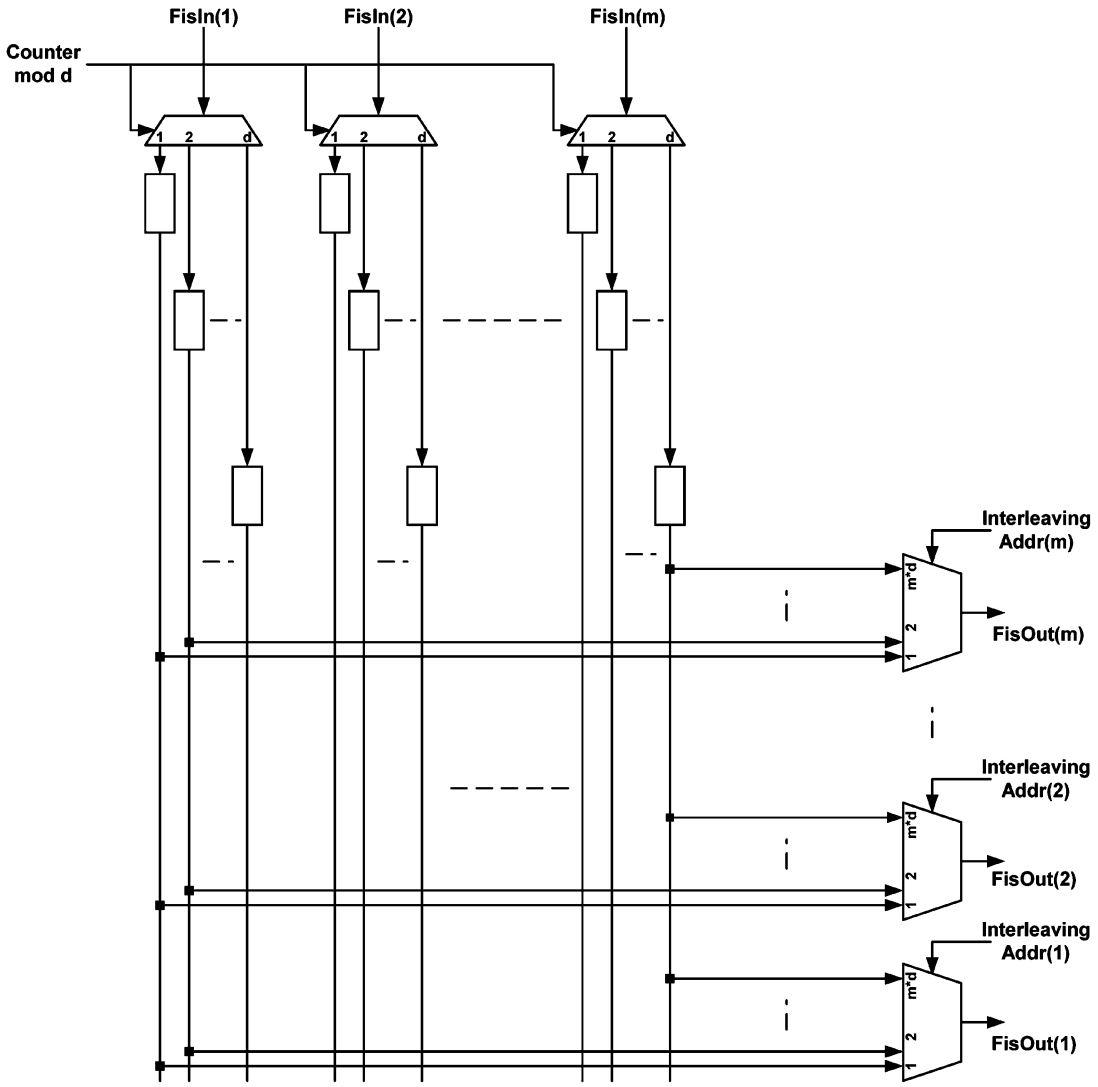

Fig. 10. Finite permutation network architecture.

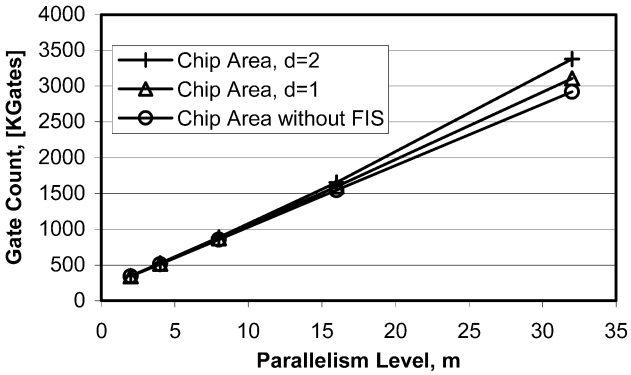

Fig. 11. Parallel turbo decoder ASIC chip area versus parallelism level.

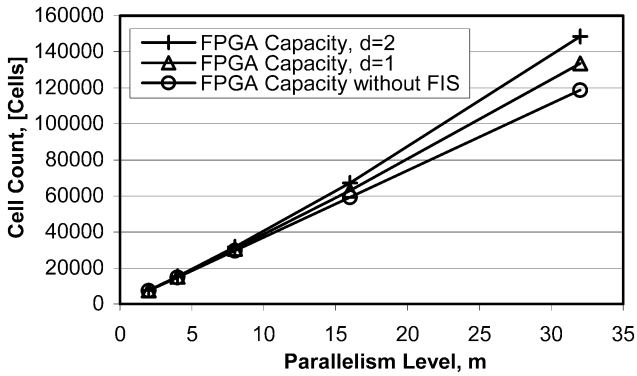

Fig. 12. Parallel turbo decoder FPGA chip capacity versus parallelism level.

The nonmonotonic changes in Fig. 15 are due to employing an integral number of decoding units and SISOs.

The latency reduction is summarized in Fig. 16. A linear speedup with chip area increase is evident in the chart. Recall that the level of parallelism is also linear in chip area (Fig. 11), thus achieving an attractive cost/performance ratio.

C. BER Performance

The parallel decoder was simulated over a AWGN channel using binary phase-shift keying modulation. The results refer to rate 1/3, 2/3, and 3/4 turbo code with two identical eight-state convolutional encoders with $g_0 = 13$, $g_1 = 17$ generator [29].

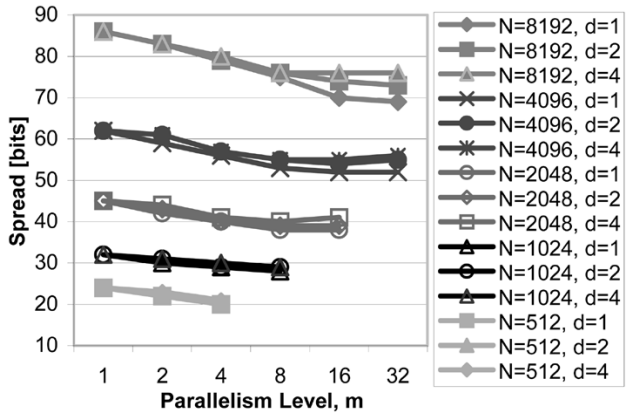

Fig. 13. PI spread.

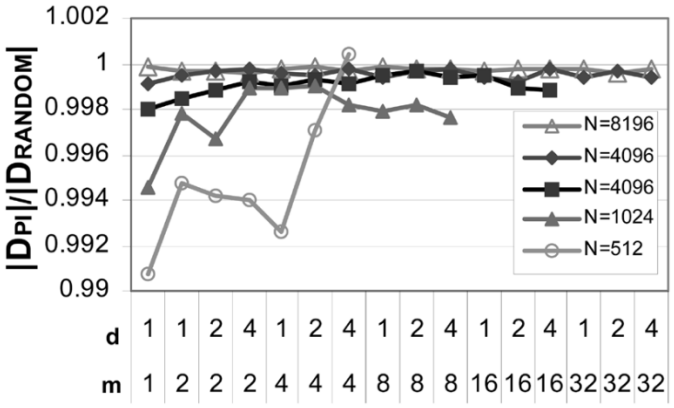

Fig. 14. PI dispersion Γ.

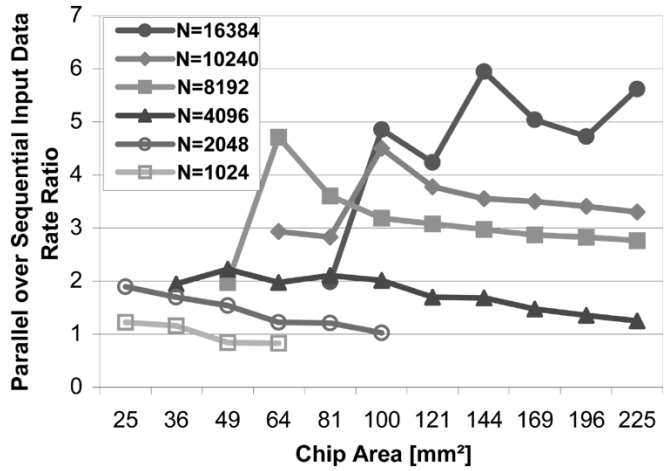

Fig. 15. The ratio of parallel to sequential throughput.

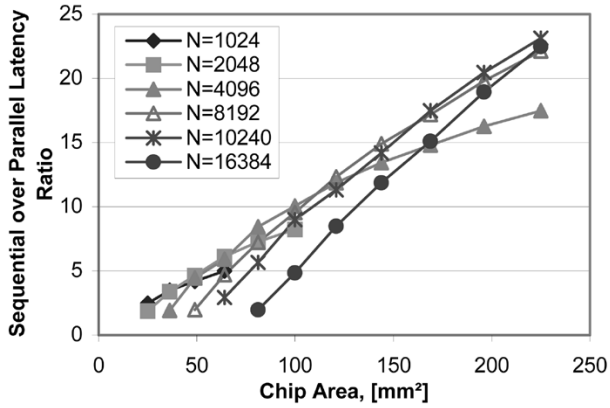

Fig. 16. The ratio of sequential to parallel latency.

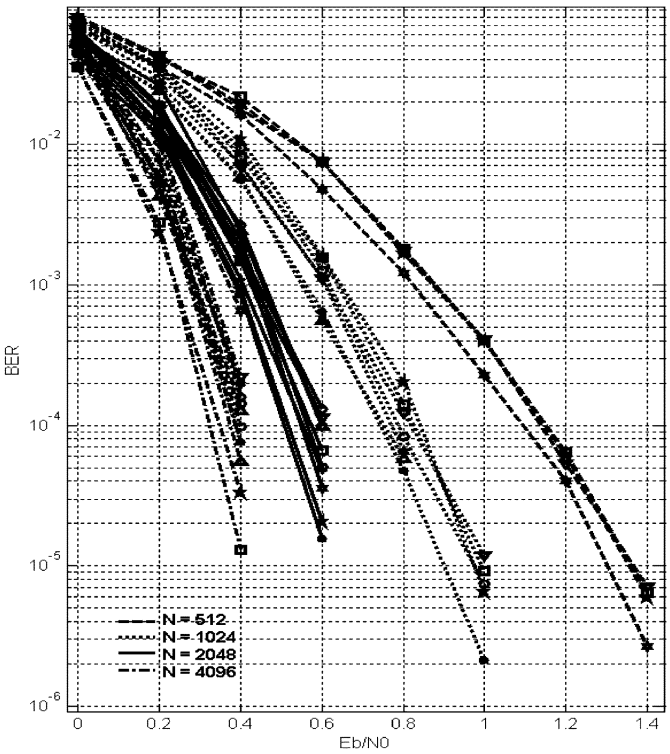

Fig. 17. BER results, CR= $1/3$, $N = 0.5, 1, 2, 4$ K.

The decoder performed ten decoding iterations, using a 32-bit sliding window.

Each line in Figs. 17 and 18 refers to a different (m, d) configuration for a given block size and code rate. We used $d = 1, 2, 4$ and $m = 1, 2, 4, \ldots, N/256$ values. The number of error events that occurred for the last computed BER point for the given (m, d) configuration is at least five. For other points the number of the error events is in the range 10–100.

The results in Fig. 17 refer to code rate 1/3. The results show only slight deviations relative to the sequential decoder ($m = 1$, $d = 1$). For all different block lengths and for different m values, performance is within 0.05 dB of the sequential turbo decoder. Since the lines for the same N are so close together, marking (m, d) values per line is ineffective. Generally, for the larger block lengths, the small degradation is compensated by applying $d = 2$.

For some configurations of the decoder, these results outperform marginally 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 (m, d) permutations as well.

The results in Fig. 18 refer to code rates 1/3, 2/3 and 3/4 for a fixed block length of 2048. These code rates were obtained by puncturing the outputs of the convolutional encoders (see Fig. 1). The results are similar to those of Fig. 17.

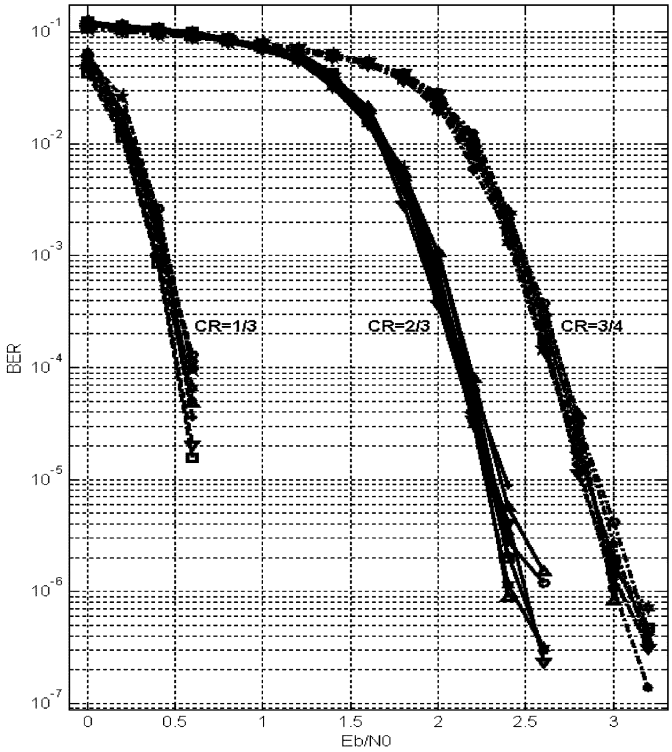

Fig. 18. BER results, $N = 1024$, CR= $1/3, 2/3, 3/4$.

TABLE I
IMPLEMENTATION EXAMPLE

Design Characteristic	Sequential	Parallel
Active Area	100 mm^2	100 mm^2
Technology	0.35µm	0.35µm
Total Gate Count	1400 KGates	1400KGates
Maximal Block Size	4096	4096
Number of SISOs	5	14
SISO gate count	78 KGates	78 KGates
Internal Clock Rate (F_{int})	60 MHz	60 MHz
Latency (one iteration)	140.8 µs	14.02µs
Max Throughput (5 iterations)	29.09 Mb/s	58.43 Mb/s

D. Implementation Example

Table I lists the results for parallel and sequential decoder ASIC implementations. ASIC synthesis was performed using Synopsys synthesizer for a 0.35μ technology and Passport standard cell libraries. For comparison, we have also described an FPGA implementation; Synplicity was used for FPGA synthesis, and Xilinx physical design tools were used for FPGA floor-planning, place, and route.

For an area-optimized SISO implementation, such as reported in [9] and [10], SISO gate count can be reduced to only 53 KGates. Then we can place more SISOs on the die, having 6 SISOs for the sequential and 21 SISOs for the parallel case in the implementation example. In addition, thanks to a shorter critical path, a higher internal clock rate of 90 MHz can be used. In that case, a throughput of 114.11 Mb/s is achieved (52 Mb/s for the sequential case, see Fig. 15). Latency is reduced to only 7.18 μs per iteration. Scaling to a more advanced technology (e.g., 90 nm) will allow further throughput and latency improvements.

VII. 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. The error correction performance was within 0.05 dB of that of the sequential turbo decoder. 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 six for the same chip area. 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.

ACKNOWLEDGMENT

The authors are grateful to the anonymous referees for many helpful and constructive comments.

REFERENCES

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes," in *Proc. ICC'93*, Geneva, Switzerland, 1993, pp. 379–427.
[2] C. Berrou and A. Glavieux, "Near optimum error correcting coding and decoding: Turbo-codes," *IEEE Trans. Commun.*, vol. 44, no. 10, pp. 1261–1271, Oct. 1996.
[3] G. Masera, G. Piccinini, M. R. Roch, and M. Zamboni, "VLSI architectures for turbo-codes," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 7, no. 3, pp. 369–379, Jun. 1999.
[4] Z. Wang, Z. Chi, and K. K. Parhi, "Area-efficient high speed decoding schemes for turbo/MAP decoders," in *Proc. 2001 IEEE Int. Conf. Acoustics, Speech and Signal Processing*, Salt Lake City, UT, May 2001, pp. 2633–2636.
[5] G. Park, S. Yoon, C. Kang, and D. Hong, "An implementation method of a turbo-code decoder using a block-wise MAP algorithm," presented at the VTC Fall 2000, Boston, MA, Sep. 2000.
[6] P. A. Beerel and K. M. Chugg, "A low latency SISO application to broadband turbo decoding," *IEEE J. Select. Areas Commun.*, vol. 19, May 2001.
[7] J. Hsu and C. Wang, "A parallel decoding scheme for turbo codes," in *Proc. ISCAS'98*, vol. 4, Jun. 1998, pp. 445–448.
[8] S. Yoon and Y. Bar-Ness, "A parallel MAP algorithm for low latency turbo decoding," *IEEE Commun. Lett.*, vol. 6, no. 7, pp. 288–290, Jul. 2002.
[9] B. Bougard, A. Giulietti, V. Derudder, J. W. Weijers, S. Dupont, L. Hollevoet, F. Catthoor, L. V. der Perre, H. De Man, and R. Lauwereins, "A scalable 8.7 nJ/bit 75.6 MB/s parallel concatenated convolutional (TURBO-) CODEC," in *ISSCC 2003*, San Francisco, CA, Feb. 2003, pp. 152–153.
[10] A. Giulietti, B. Bougard, V. Derruder, S. Dupont, J. W. Weijers, and L. V. der Perre, "A 80 Mb/s low-power scalable turbo codec core," in *CICC'02*, Orlando, FL, May 2002.
[11] F. Gilbert, M. J. Thul, and N. When, "A scalable system architecture for high throughput turbo-decoder," in *SIPS'02*, San Diego, CA, Oct. 2002, pp. 152–158.
[12] Z. Wang, Z. Chi, and K. K. Parhi, "Area-efficient high-speed decoding schemes for turbo decoders," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 10, no. 6, pp. 902–912, Dec. 2002.
[13] R. Dobkin, M. Peleg, and R. Ginosar, "Parallel VLSI architecture for MAP turbo decoder," in *Proc. PIMRC'2002*, vol. 1, Sept. 2002, pp. 384–388.
[14] M. J. Thul, F. Gilbert, and N. When, "Concurrent interleaving architectures for high-throughput channel coding," in *ICASSP'03*, vol. 2, Hong Kong, Apr. 2003, pp. 613–616.
[15] A. Giulietti, L. van der Perre, and M. Strum, "Parallel turbo coding interleavers: Avoiding collisions in accesses to storage elements," *Electron. Lett.*, vol. 38, no. 5, pp. 232–234, Feb. 2002.

[16] A. Tarable, G. Montorsi, and S. Benedetto, "Mapping interleaving lows to parallel turbo decoder architectures," in *Proc. 3rd Int. Symp. Turbo Codes and Related Topics*, Brest, France, 2003, pp. 153–156.
[17] S. Crozier, "Turbo-code design issues: Trellis termination methods, interleaving strategies, and implementation complexity," in *Proc. ICC'99*, Vancouver, Canada, 1999.
[18] C. Berrou, "Additional information on the EUTELSAT/ENST-bretagne proposed channel turbo coding for DVD_RCS," in *6th Meeting Ad Hoc Group on Return Channel Over Satellite*, Geneva, Switzerland, 1999.
[19] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inf. Theory*, vol. IT-20, no. 2, pp. 284–287, Mar. 1974.
[20] R. Dobkin, M. Peleg, and R. Ginosar, "Parallel VLSI architecture and parallel interleaver design for low-latency MAP turbo decoders," in *CCIT TR436*, Jul. 2003.
[21] A. Hunt, S. Crozier, M. Richards, and K. Gracie, "Performance degradation as a function of overlap depth when using sub-block processing in the decoding of turbo codes," in *Proc. IMSC'99*, Ottawa, Canada, 1999, pp. 276–280.
[22] H. Dawid and H. Meyr, "Real-time algorithms and VLSI architectures for soft output MAP convolutional decoding," in *PIMRC'95*, Toronto, Canada, Sep. 1995, pp. 193–197.
[23] S. N. Crozier, "New high-spread high-distance interleavers for turbo codes," in *20th Biennial Symp. Communications*, Kingston, Canada, 2000, pp. 3–7.
[24] C. Heegard and S. B. Wicker, *Turbo Coding*. Norwell, MA: Kluwer Academic, 1999, pp. 50–52.
[25] S. W. Golomb and H. Taylor, "Construction and properties of Costas arrays," *Proc. IEEE*, vol. 72, no. 9, pp. 1143–1163, Sep. 1984.
[26] J. D. Andersen, "Selection of code and interleaver for turbo coding," in *1st ESA Workshop Tracking, Telemetry and Command Systems ESTEC*, The Netherlands, Jun. 1998.
[27] D. Weinfeld, "Symbol-wise implementation of turbo/MAP decoder," Technion—Israel Institute of Technology, Electrical Engineering Dept., Communications Lab. (ISIS Consortium), Internal Rep., 2002.
[28] W. Feng, J. Yuan, and B. S. Vucetic, "A code-matched interleaver design for turbo codes," *IEEE Trans. Commun.*, vol. 50, pp. 926–937, Jun. 2002.
[29] M. S. C. Ho, S. S. Pietrobon, and T. Giles, "Improving the constituent codes of turbo encoders," in *IEEE Globecom'98*, vol. 6, Nov. 1998, pp. 3525–3529.

Rostislav (Reuven) Dobkin received the B.Sc. and M.Sc. degrees in electrical engineering from the Technion—Israel Institute of Technology, Haifa, in 1999 and 2003, respectively.

During 1997–2000, he worked within the RAFAEL ASIC Experts design group and during 2001–2002 led a very large-scale integration (VLSI) Design Group at IC4IC Ltd., developing family of chips for communications. In parallel, he was a Teaching Assistant at the Technion Electrical Engineering Department. His research interests are VLSI architectures, parallel architectures, asynchronous logic, high-speed interconnect, synchronization, GALS systems, SoC, and NoC.

Michael Peleg (M'87–SM'98) received the B.Sc. and M.Sc. degrees from the Technion—Israel Institute of Technology, Haifa, in 1978 and 1986, respectively.

Since 1980 he has been with the communication research facilities of the Israel Ministry of Defense. He is associated with the Electrical Engineering Department of the Technion, where he is collaborating in research in communications and information theory. His research interests include wireless digital communications, iterative decoding, and multiantenna systems.

Ran Ginosar received the B.Sc. degree in electrical engineering and computer engineering from the Technion—Israel Institute of Technology, Haifa, in 1978 and the Ph.D. degree in electrical engineering and computer science from Princeton University, Princeton, NJ, in 1982.

He was with AT&T Bell Laboratories in 1982–1983 and joined the Technion Faculty in 1983. He was a Visiting Associate Professor with the University of Utah in 1989–1990 and Visiting Faculty with Intel Research Labs in 1997–1999. He cofounded four companies in the areas of electronic imaging, medical devices, and wireless communications. He is Head of the VLSI Systems Research Center at the Technion. His research interests include VLSI architecture, asynchronous logic, electronic imaging, and neuroprocessors.