Fine Grain Parallel Decoding of Turbo Product Codes: Algorithm and Architecture

Thierry Goubier*, Catherine Dezan†, Bernard Pottier‡ and Christophe Jégo‡

*CEA List
Embedded Real-Time System Foundations Laboratory
91191 Gif sur Yvette Cedex, France
Email: thierry.goubier@cea.fr
†Lab-STICC, CNRS UMR 3167
29200 Brest, France
Email: {Catherine.Dezan, Bernard.Pottier}@univ-brest.fr, christophe.jego@telecom-bretagne.eu

Abstract—In turbo decoding of product codes, we propose an algorithm implementation, based on the Chase-Pyndiah algorithm, which exhibits a modular, simple structure with fine grain parallelism. It is implemented into deep pipelined architectures, including an interleaving block decoding scheme, with good potential on FPGAs and MP-SoCs targets. We include an evaluation of the essential parameters of those architectures, which are situated in a different area of the block turbo decoder implementation design space.

I. INTRODUCTION

Turbo codes have appeared with Berrou’s paper[1], promising near Shannon limit decoding of transmitted data on noisy channels. Berrou introduced CTCs, or Convolutional Turbo Codes. R. Pyndiah [2] introduced iterative decoding of product codes (product codes were introduced in [3]) based on the Chase algorithm, which represent the best compromise complexity-performance for BCH and RS codes with a reasonable error correction capability (1-4 for BCH, 1-2 for RS). Turbo product codes are interesting for high code rates (over 0.95) or small blocks size.

As turbo codes have applications for use in communication systems, algorithms and architectures are further investigated to either reduce the cost of an implementation, or increase the performance of the decoder (essentially, the data rate), while maintaining a good efficiency (capability to successfully decode on a noisy channel, often measured in bit error rate, or BER, for a given signal to noise ratio). Of importance for an algorithm is to develop an appropriate architecture for its implementation on a specific target (ASIC, FPGA, system on a chip). Our primary targets are FPGAs and massively parallel systems on chip (MPSoCs).

We have developed an alternative Chase-Pyndiah decoding algorithm implementation which trades complexity and minimalism in number of operations for simplicity, regularity and fine grain parallelism. This change allows us to propose new architectures for a high data rate decoder (over 1Gbit/s), without compromising decoding efficiency, and with potential on specific implementation architectures (FPGAs, datapaths, MPSoCs). Our high-level synthesis tools were enablers for this novel algorithm and architectures.

This paper is organised as follows: first a description of turbo product codes and their decoding, followed by a related work subsection. Section II describes the Chase-Pyndiah algorithm and our Mini-Maxi implementation. Section III details our architectures for the elementary decoder, the block decoder, and the interleaved and full architectures for the block turbo decoder. Section IV contains a complexity analysis and evaluation of our architectures, and Section V concludes this paper.

A. Turbo product codes

A product code consists of a product of two linear block codes $C_1(n_1,k_1,\delta_1)$ and $C_2(n_2,k_2,\delta_2)$. In practice, $C_1$ and $C_2$ are BCH (Bose, Ray-Chaudhuri, Hocquenghem) or RS (Reed-Solomon) codes. The result product code $C_p$ has characteristics $(n_1 \times n_2, k_1 \times k_2, \delta_1 \times \delta_2)$. The product code word is a matrix of $n_1$ rows and $n_2$ columns, such that:

- The data payload is a $k_1 \times k_2$ matrix $M$ inside the code word.
- The $k_1$ rows of $M$ are encoded by $C_2$, resulting in a $k_1 \times n_2$ matrix $C$.
- The $n_2$ columns of $C$ are encoded by $C_1$, resulting in a $n_1 \times n_2$ matrix.

Each data matrix is transmitted over a noisy channel, picking up noise. It is then demodulated, with a soft output to the demodulator of $q$ bits of soft decision (see figure 1). The sign of the soft decision indicates the bit (binary) value for that symbol; the soft decision is the demodulator confidence in that bit.

Decoding is done iteratively, with soft-input, soft-output decoders (see Figure 2). Each decoding iteration consists of two halves: first decoding all rows with code $C_2$, then decoding all columns with code $C_1$. The decoding of a vector (a row or a column) is done by an elementary soft-input, soft-output decoder for the code. It returns a reliability vector, the extrinsic information, which is added to the received vector scaled by

---

1This work was supported by Région Bretagne under the PRIR VALMADEO project.
the elementary decoder in three stages (reception, processing and emission), and pipeline it. Parallel decoding is possible given that all the rows may be decoded in parallel; the same with the columns. Thus a parallel decoder may be able to use a large number of elementary decoders in parallel to reach high performance ([5][6][7]). A recurrent problem is the use of intermediary buffers between half-iterations, which increases latency and the amount of logic; recent work has a trend of replacing this by a carefully designed interconnection network [6]. Another optimisation consists of storing multiple input symbols at the same address, and decoding them in parallel in a single, larger, elementary decoder [5].

Implementations may also be optimised by carefully tuning the Chase-Pyndiah algorithm parameters; a good breakdown of all parameters and their impact on a fast implementation is given in [7]. Alternative algorithms also exist, including changing the generation of test vectors such as in the Fast Chase algorithm [8]. Switching from BCH codes to RS codes also has benefits, particularly at very low error rates [9] while maintaining a relatively low complexity [10]. A similar algorithm to ours is described in [11], using the Cartesian product of the test and received vector.

Our algorithm builds upon these works, by being an algebraic equivalent to the Chase-Pyndiah algorithm; hence parameter scaling will give exactly the same results in terms of decoder efficiency (however, implementation complexity influence will differ). Our algorithm is very modular, hence switching codes or test vector generations are possible. In the architecture, we keep buffers between half-iterations, and we do not duplicate elementary decoders, choosing to deep pipeline them instead. Finally, our high-level language to logic synthesis approach and tools can be compared to similar approaches using System-C [12].

II. ALGORITHM

The core algorithm for turbo product code decoding is the block decoding algorithm used to decode the two linear block codes used in the product code. The turbo decoding principle is to use a SISO (Soft Input, Soft Output) linear block decoding algorithm.

The first algorithm used is based on the Chase algorithm, a soft-input, hard-output algorithm, combined with a soft-output stage in [2], hence the Chase-Pyndiah algorithm.

A. The Chase-Pyndiah algorithm

The Chase [13] algorithm is a linear block code decoding algorithm which uses soft decision bits (soft input) to be able to, at high signal to noise ratio, correct errors that are $2\delta$ in squared Euclidean distance from the transmitted code word for a code $C(n, k, \delta)$, rather than $(\delta - 1)/2$ as done by straight algebraic decoding. It is in fact a class of algorithms, with three variants.

The principle of the Chase algorithm is to find the codeword the closest to the received sequence, using the soft decisions to measure this distance. With $R$ the received sequence and $Y$ the binary of $R$ (the sign of $r_i$ is $y_i$: 0 if $r_i < 0$, 1...
This is done by computing the reliability of all the components $D$ coded vector output decoding algorithm, returning us a decision: the decoded codeword exists, then the reliability is excluding the decision of candidate keywords is the list of the $C$ sequence, the list of candidate codewords (where the list of the previous one.

The algorithm is then the following (as per [13]):

- Search for the $\delta/2$ least reliable positions in $R$ (for Chase-2 and Chase-3).
- Compute the set of error patterns.
- For each error pattern $T$:
  - Form $V_t = Y \oplus T$.
  - Binary decode $V_t$ in $C_t$. If this fails, process the next error pattern.
  - Compute $W_\alpha(C_t) = \sum_{i=1}^{n} |r_i|(y_i \oplus c_i)$.
  - If $W_\alpha(C_t)$ is the lowest metric computed, then store $C' = Y \oplus C_t$.
- If an error pattern $C'$ was stored, decode as $D = Y \oplus C'$, else decode as $D = Y$.

The Chase decoding algorithm is then a soft-input, hard-output decoding algorithm, returning a decision: the decoded vector $D$.

The Chase-Pyndiah algorithm [2] is a turbo product code decoding algorithm. The elementary decoding step requires a soft-input, soft-output decoder out of the Chase algorithm. This is done by computing the reliability of all the components of the decision $D$ of the Chase algorithm. For each bit $j$ in the sequence, the list of candidate codewords (where the list of candidate keywords is the list of the $C'$ generated above, excluding the decision $D$) is searched for the closest (by metric) competing codeword $C$ such that $c_j \neq \hat{d}_j$. If such a codeword exists, then the reliability is

$$r_j' = \frac{||R - C'||^2 - ||R - D||^2}{4}$$

else

$$r_j' = r_j + \beta$$

Where $\beta$ is dependent on the iteration and the channel characteristics.

**B. The Mini-Maxi algorithm**

The Mini-Maxi algorithm use mini-maxi arrays (hence the name) to store the best metrics of all decoded test vectors. This allows the algorithm to simplify the reliability computations, and to even forego completely an explicit choice of the decided word and its concurrent words.

The inputs of the Mini-Maxi algorithm are as in Figure 3, with $Y_k$ the binary vector of $R_k$ (the signs of the symbols of $R_k$). Mini and Maxi are two arrays of length $n$, initialised at an arbitrary high value. The process is:

1. Least reliable search on $R_k$.
2. Test vectors processing: this part is a loop, done for each test vector.
   a) Generate the test vector $V_t$ out of an error pattern, the least reliable positions and $Y_k$.
   b) Binary decode the test vector. It is now a codeword $C_t$. If the decoding fails, process the next test vector.
   c) Compute the metric of the test vector.

$$m_t = \sum_{i=1}^{n} |r_k|(y_k \oplus c_i)$$

d) Update each element of the Mini/Maxi arrays using the metric $m_t$ and the test vector.

$$c_i = \begin{cases} 0 & \text{if } \text{Mini}_i < \text{Maxi}_i \\ 1 & \text{if } \text{Maxi}_i < \text{Mini}_i, m_t \end{cases}$$

3. Reliability computation $f_i = \text{Mini}_i - \text{Maxi}_i$. If one of $\text{Maxi}_i$ or $\text{Mini}_i$ is arbitrary high, then no concurrent was found and the reliability is updated to $f_i = (2u(f_i) - 1)^2$. The extrinsic information is computed as $w_{k+1,i} = f_i - r_k$, and $r_{k+1,i} = r_i + \alpha_k w_{k+1,i}$. If needed, the decision $D$ is computed as $d_i = u(f_i)$.

By construction, the test vector with the smallest metric $m_d$ recorded in the Mini-Maxi arrays is the decision word $D$. The concurrent word $C$ for position $i$ is the lowest metric test vector with $c_i \neq \hat{d}_i$. The concurrent word metric $m_c$ is stored in the other array (Maxi if $m_d$ is in Mini, else the reverse). $f_i$ is signed, and remembering that $m_d < m_c$, $d_i = 0$ means that $f_i = \text{Mini}_i - \text{Maxi}_i = m_d - m_c$. $f_i$ is equal to the normalisation of the log-likelihood ratio of the decision $d_i$ (as explained in [14]). As such, this is an algebraic equivalent of the Chase-Pyndiah algorithm.

This Mini-Maxi implementation is simple, wide and regular, allowing for parallel or vector operations: Mini-Maxi updates, reliability and output computations. Our synthesis tools shows that the binary decoder can be implemented in the same way: wide and fast.

**III. ARCHITECTURE**

In implementing this algorithm, we have chosen to maintain the overall shape of the algorithm, with well divided steps that can be done in parallel, and wide, vector type operations.

The overall block turbo decoder is built out of one or more block decoders, i.e. a decoder which takes a complete data code block (a rectangular matrix of transmitted data) and operates one half-iteration on this block. The block decoder is itself built out of the elementary decoder.

$\beta$ where $u(f_i)$ is either 0 (for $f_i < 0$) or 1 ($f_i \geq 0$).
A. The elementary decoder

The elementary decoder directly applies the algorithm to an input vector (which may already have been adjusted by previous iterations) and a received vector (the vector as is out of the demodulator), and returns a modified vector (along the received vector). It is designed along the stages described in II-B (see Figure 4) where full vectors \((V_i, C_i)\) are propagated between stages.

\[ \text{Swap after each half-iteration} \]

![Diagram of elementary decoder](image)

This decoder is implemented as a pipeline, sequenced along the three main stages. Stage 2 is itself internally a pipeline. Overall, the cycle time \(C\) of the pipeline is equal to \((T_v + 4)c\), where \(c\) is the stage 2 internal pipeline cycle time and \(T_v\) the number of test vectors. This architecture makes plain the ability to change code by changing the binary decoder stage b), and the fact that full vectors are passed between the stages.

![Diagram of pipelined elementary decoder](image)

Acceleration of this architecture is done by duplicating stage \(2K\) times, and shifting each duplicate \(T_v/K\) cycles later than the previous one (see Figure 5). In this way, a new decoding sequence can start on stage 2 every \(T_v/c\). The maximum performance is achieved for \(K = T_v\), where a new test vector processing sequence can start every \(c\). The benefit of this is that we can increase the performance of the elementary decoder to a point similar to putting multiple decoders in parallel, but with only the surface duplication of stage 2 and limited to the number of test vectors.

B. The block decoder

The block decoder is based on the elementary decoder, but focusing on a full matrix of data, not just a vector. We design around the following idea: the internal memory of the block decoder is able to keep a full matrix (received and being decoded) inside. For simplicity, we suppose that the two linear block codes be the same and the data matrix be square; adaptation to two different codes \(C_1\) and \(C_2\) are shown later.

![Diagram of pipelined block decoder](image)

Our solution is to make a pipelined block decoder with a number of steps equal to the number of rows in the block. Hence a full block can be kept in the intermediary registers in the pipeline. The block decoder in Figure 6 reads its input blocks row by row. We make the block decoder write in columns into its output buffers. Once a full block has been fully decoded (last vector written), the output buffers can be used as input for the next half-iteration, reading them by rows.

The full pipeline for the block decoder is built on a fully pipelined version of the elementary decoder, with a large number of steps (subdividing stages 1 and 3 of the elementary decoder, and as needed sub stages a, b, c, d in stage 2) and \(T_v\) stage 2’s. Overall, we balance the length of each stage to be able to reach the number of necessary stages in the block decoder pipeline: 32 stages for an \(eBCH\)(32,26), 128 stages for an \(eBCH\)(128,120). Latency is of course 32 or 128 cycles, but the sustained throughput of this decoder is one pair \(R_{k+1}, R\) per cycle and is equal to its input data rate.
C. Interleaved architecture

The interleaved architecture makes full use of a single block decoder, by interleaving the decoding of two blocks. While block \( i \) is being read from the input buffers and processed by the block decoder (one pair of decoded vector/received vector at a time) block \( i - 1 \) is being decoded and written to the output buffers (one pair of decoded vector/received vector at a time).

Fig. 7. Interleaved block decoding. The block decoder can do one \( W_k \), \( R_j \) simultaneously due to the deep pipeline (\( W_k \)). Write vectors to output from half-iteration \( k \), \( R_j \) read vectors from input from iteration \( j \), \( D_q \) write decision vectors at the end)

Once the decoding of a block is finished, the output buffers are exchanged with the input buffers, and the block decoder is ready to start the next half-iteration for this block (see Figure 6). In this scheme, by interleaving, we maintain a full pipeline while coping with the necessary rebuild of the matrix between two half-iterations (Figure 7).

The two blocks are separated by 4.5 half-iterations\(^3\), and the overall decoding of a block takes 9 half-iterations. The ninth half-iteration is needed for the pipeline equilibrium; a common approach is to use the decision after 8 half-iterations, disregarding the ninth. Overall, the complete decoder outputs a fully decoded block every 4.5 half-iterations. Each half-iteration takes \( 2n \) cycles for an \((n, k, d)\) elementary code.

When the two linear block codes are different (with a rectangular data matrix, not square), our interleaved architecture requires a minimum of two block decoders with the adequate characteristics (number of steps in each pipeline).

D. Full architecture

Fig. 8. Full architecture: 8 block decoders in a pipeline, start reading on \( B_i \) while outputting the decision \( B_{D_{i-1},15} \)

The fastest version of our architecture consists of throwing the maximum of hardware resources at it, that is eight block decoders in sequence (see Figure 8). We maintain the block decoder as is, with the same characteristics as before, except that, instead of swapping output buffers with input buffers at the end of decoding a block, we exchange the output buffers of block decoder \( i \) with the input buffers of block decoder \( i+1 \).

This turbo decoder produces one block every half-iteration. With a code sufficiently large, and a good frequency \( f \) for each pipeline step, the data rate can be very large.

IV. SYNTHESIS AND EVALUATION

A. Methodology

Our algorithms and architectures were developed with a high-level language, Smalltalk, implemented in parts on an FPGA target with the Madeo tool chain [15]. Algorithms and architectures are written in Smalltalk, including finite field arithmetic for the eBCH(32, 26)/eBCH(128, 120) binary decoders, and synthesised for FPGAs (logic and physical synthesis) [16].

To characterise the architectures and implementation of our algorithm, we use the following set of parameters:

- \( n \) the length of the code
- \( q \) the number of bits for symbol quantification (soft decision bits)
- \( L_v \) the number of least reliable symbols
- \( T_v \) the number of test vectors

We analyse first in time and space each basic element of the algorithm, as implemented in the elementary decoder. We then give an evaluation in terms of latency, throughput and memory size of the elementary decoder and of the full architecture.

B. Basic cells of the Elementary Decoder

For the elementary decoder, we consider the following basic cells:

- **Least reliable search**(stage 1): this search of the least \( L_v \) values among \( n \) uses a parallel bitonic sort. The complexity depends on \( L_v \), but also on \( q \) since each value uses \( q \) bits.
- **Test vector generation**(stage 2a): Combines a test pattern, the least reliable positions and the input vector \( Y_k \). Complexity is driven by \( L_v \).
- **Test vector binary decoder**(stage 2b): binary decode the vector, recompute parity. In the case of an eBCH(32, 26) it consists of a syndrome calculation in \( GF(2^5) \) followed by 31 comparisons and an eventual bit flip, with in parallel a parity computation. Syndrome computations have been shown to be reduction trees [16].
- **Metric**(stage 2c): compute \( \sum |r_{k,i}|c_{i,j} \). Optimised due to the fact at most \( L_v + 1 \) (parity) + 1 (binary error correction) have changed between \( C_t \) and \( Y_k \). Adders start at \( q - 1 \) bits wide, and it’s a reduction tree in shape (end at \( q + 1 \) or \( q + 2 \) bits).
- **Mini-Maxi update**(stage 2d): The arrays Mini and Maxi are updated with the metric computation. Vector operator: \( n \) operations in parallel, with \( q + 3 \) bits wide comparators.
- **Reliability and output computations**(stage 3): vector parallel operations, \( n \) wide, with (excluding \( \beta \) and \( d \))

\[ r_{k+1} = r_i + \alpha_k(Min-i - Max_i - r_{k+1}) \]

The impact of the parameters are summarised in Table I.
This implementation is modular in regard to the linear block wide, fine-grain parallel execution targets and deep pipelines. Fine-grain parallelism and simplicity, adapted for execution on algorithm for decoding turbo product codes which exhibit complexity.

Overall, the architectures presented have a quite high memory count; we expect this to be efficient however on FPGAs, where most of the memory will be latches in logic cells, and on MP-SoCs, where local memory won’t be a limiting factor for this algorithm, but computational power. Our objectives are now complete synthesis of the interleaved architecture on FPGA, and adaptation to MP-SoCs without reconfigurable nor dedicated IPs.

C. Architectural evaluation

- Elementary decoder using $T_v$ cells for stage 2:
  - Latency: $\sum_{allbasiccells} Latency$
  - Throughput: one vector per cycle : $n$ symbols of $q$ bits per cycle
  - Memory (bits):
    \[ 2q\cdot(T_v+6)+L_v\cdot\log_2(n)+T_v\cdot(3n+q+2+2n(q+2)) \]

- Interleaved architecture with one block decoder
  - Latency: $\frac{16n}{T_v}$
  - Information throughput: $\frac{k^2}{9n} f$ bit/s
  - Memory (bits):
    \[ M_I \approx 3(2n^2q) + T_v \times n(2(q+2) + 3) + n^2 \]

- Full architecture with eight block decoders
  - Latency: $\frac{16n}{T_v}$
  - Information throughput: $\frac{k^2}{9n} f$ bit/s
  - Memory (bits): $8M_I - \frac{7}{2}(n^2) - (2n^2 \times q)$

For a frequency $f$ of 100MHz, $q = 5$ bits, $T_v = 16$, reasonable values on our synthesis results, we can extrapolate throughput and memory amounts for the interleaved and full architectures (table II).

<table>
<thead>
<tr>
<th>Code</th>
<th>Throughput(bit/s)</th>
<th>Memory(bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Interleaved</td>
<td>Full</td>
</tr>
<tr>
<td>eBCH(72, 26)</td>
<td>2.34G</td>
<td>1.38G</td>
</tr>
<tr>
<td>eBCH(128, 120)</td>
<td>1.25G</td>
<td>11.2G</td>
</tr>
</tbody>
</table>

V. Conclusion

We have presented an implementation of the Chase-Pyndiah algorithm for decoding turbo product codes which exhibit fine-grain parallelism and simplicity, adapted for execution on wide, fine-grain parallel execution targets and deep pipelines. This implementation is modular in regard to the linear block code used and the method used to generate test vectors. Presented architectures for the elementary decoder and block decoder show the interest of using a deep pipeline: an interleaved architecture shows how the deep pipeline is fully used with a single decoder. Our high-level language synthesis approach has allowed us to explore different areas in the turbo product code decoder implementation problem space.

REFERENCES